

O'REILLY®

Third Edition  
Covers Java 8-11

# Head First

# Java

A Learner's Guide  
to Real-World  
Programming

---

Kathy Sierra,  
Bert Bates &  
Trisha Gee

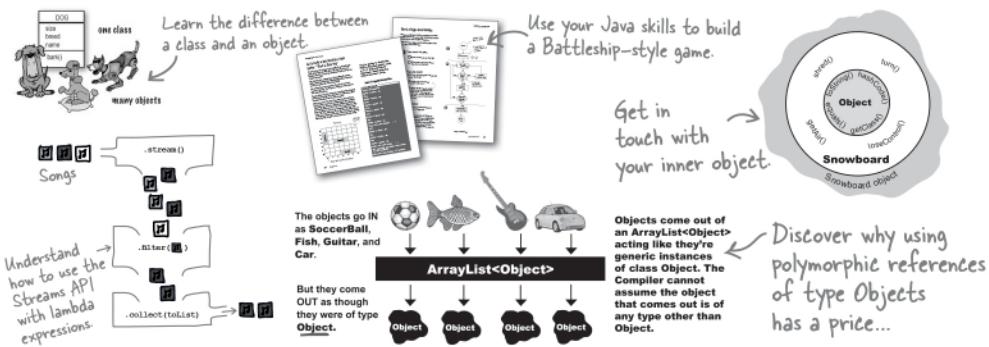


A Brain-Friendly Guide

# Head First Java

## What will you learn from this book?

*Head First Java* is a complete learning experience in Java and object-oriented programming. With this book, you'll learn the Java language with a unique method that goes beyond how-to manuals and helps you become a great programmer. Through puzzles, mysteries, and soul-searching interviews with famous Java objects, you'll quickly get up to speed on Java's fundamentals and advanced topics including lambdas, streams, generics, threading, networking, and the dreaded desktop GUI. If you have experience with another programming language, *Head First Java* will engage your brain with more modern approaches to coding—the sleeker, faster, and easier to read, write, and maintain Java of today.



## What's so special about this book?

If you've read a Head First book, you know what to expect—a visually rich format designed for the way your brain works. If you haven't, you're in for a treat. With *Head First Java*, you'll learn Java through a multisensory experience that engages your mind, rather than by means of a text-heavy approach that puts you to sleep.

JAVA

US \$69.99

CAN \$87.99

ISBN: 978-1-491-91077-1



5 6 9 9 9

9 7 8 1 4 9 1 9 1 0 7 7 1

"What a fun and quirky book! I've taught Java for many years and can honestly say this is the most engaging resource I've ever seen for learning to program. It makes me want to learn Java all over again."

—Angie Jones  
Java Champion

"The only way to decide the worth of a tutorial is to decide how well it teaches. *Head First Java* excels at teaching."

—slashdot.org

"It's definitely time to dive in—Head First."

—Scott McNealy  
former Sun Microsystems Chairman, President, and CEO

O'REILLY®

## **Other books in O'Reilly's Head First series**

*Head First Android Development*

*Head First C#*

*Head First Design Patterns*

*Head First Git*

*Head First Go*

*Head First HTML and CSS*

*Head First JavaScript Programming*

*Head First Kotlin*

*Head First Learn to Code*

*Head First Object-Oriented Analysis and Design*

*Head First PMP*

*Head First Programming*

*Head First Python*

*Head First Software Development*

*Head First SQL*

*Head First Swift*

*Head First Web Design*

## Praise for *Head First Java*, 3rd Edition

“What a fun and quirky book! I’ve taught Java for many years and can honestly say this is the most engaging resource I’ve ever seen for learning to program. It makes me want to learn Java all over again!”

— **Angie Jones, Java Champion**

“HFJ has the clearest explanation of Java Streams and Lambdas I have ever read—without the hype! It teaches important functional programming concepts with humor and style. And it was so fun I wanted to learn Java all over again. If only everyone programmed Java like they teach in this book.”

— **Eric Normand, Clojure instructor and author of *Grokking Simplicity***

“Oh how I wish I had had this book when I was learning Java! It is such fun to read, one forgets that it is a serious Java learning book. The third edition is a great step forward. It covers everything that a Java programmer needs to know in 2022+ to become proficient in the Java language. To me the best though are the illustrations, which made me chuckle quite a few times. Very well done to the Java Champion authors: Kathy, Bert, and Trisha!”

— **Dr. Heinz M. Kabutz (*The Java Specialists’ Newsletter*, [www.javaspecialists.eu](http://www.javaspecialists.eu))**

“I love *Head First Java*’s style of teaching. It is a ‘technical’ book but feels like fiction—hard to stop reading once you start with any chapter. It has fun and unconventional images, great analogies, fireside chats between a developer and compiler/runtime and so many more such features. It has a completely different and great way of teaching concepts that makes readers question their assumptions and beliefs, which I believe is crucial to letting any learner realize the power of their own curiosity. The authors of this book are no less than magicians. This is a must-read book for all Java developers to get started learning Java or to level up their existing skills in a fun way.”

— **Mala Gupta, Developer Advocate @ JetBrains, Author and Java Champion**

“I often get asked by folks new to the Java programming ecosystem, ‘What book should I start with?’ I always tell them *Head First Java*! The original editions by Kathy Sierra and Bert Bates flipped the old way of learning a programming language on its head, with a learner-centric way of teaching. It was simply revolutionary. Trisha Gee is one of the finest Java engineers and educators on the planet, and my go to person when I need something gnarly about the language explained in clear detail! The third edition brought a huge smile to my face, not only for the trip down memory lane but because once more I learned new things about Java despite having spent over 20 years with it :-).”

— **Maritijn Verburg aka “The Diabolical Developer”; Java Champion and Principal Group Manager for Java @ Microsoft**

“The *Head First Java* book is legendary, and I can’t think of a better person than Trisha Gee to update it. I already knew Trisha was awesome, but I didn’t know she was funny. Now I do! The third edition is authoritative, entertaining, clear, and current. If there’s a better way to learn Java, I don’t know it.”

— **Holly Cummins, Senior Principal Quarkus Software Engineer, Red Hat**

“This book is a riot! It’s got curly braces, humor, objects, metaphors, syntax, fun, code, and a proper acknowledgment that the reader is human. It takes the process of learning seriously, but does so playfully and memorably. I love the metacognitive tips, the invitations to play the role of compiler, the stories, the visuals, and the sense that learning a programming language—like any learning—is something that is open to anyone.”

— **Kevlin Henney, co-editor of *97 Things Every Java Programmer Should Know***

“I wish I’d known about this book when I’d been learning Java! For those looking to learn Java in a fun, humorous and captivating way (who knew that was possible?), and especially those who have not come from a traditional computer science background like myself, this is definitely the book for you. Never before have I laughed out loud at a programming book. It’s brilliantly written, witty, engaging, interactive, easy to follow and highly educational.”

— **Grace Jansen, Developer Advocate, IBM**

“If you’re just starting your journey learning how to program in Java, you’re faced with an overwhelming choice of books and courses ready to get you to your destination. The problem is most focus purely on the technical information, making you frequently ask, “are we there yet?” *Head First Java* takes an altogether different approach making the adventure of learning both entertaining and educational. Using arrays of dogs, pool puzzles, five-minute mysteries and sharpen your pencil (who’d have thought you need a pencil to program?), this book makes learning fun, yet making sure you absorb all the essential details you’ll need. I wish this had been available when I started learning Java!”

— **Simon Ritter, Deputy CTO at Azul and Java Champion**

“This book never disappoints. I still remember it from my early days at university and I am quite pleased to see this new improved version. *Head First Java* is very well put together with simple to understand English and coding examples. I highly recommend it to every Java developer.”

— **Nelson Djalo, Tech Entrepreneur, founder of Amigoscode.com learning platform and Amigoscode YouTube channel**

“*Head First Java* was the first book I had my son read when he wanted to learn Java. And there’s a reason: I knew the fun cartoons would captivate his attention like no other Java book I have seen before. Reading *Head First Java* was more like being on the playground than stuck in the classroom.”

— **Kevin Nilson, Google Software Engineer and Leader of the Silicon Valley Java User**

“I can only envy programmers who want to learn Java today, as they have this great book. I learned Java twenty years ago, and it was quite boring. But you’ll never be bored with this book. I’ve never seen a Java book that has a battle between Java compiler and virtual machine. This is mind-blowing!”

— **Tagir Valeev, Java Champion and Technical Lead in IntelliJ IDEA, JetBrains**

“Nearly 20 years ago after I read *Head First Java*, 1st edition, as a junior developer entering the Java world, the third edition still amazes me. Much has changed since then, including how people present tutorials. *Head First Java*, 3rd edition, is a valid alternative to today’s excellent video materials: It allows learners—junior and senior alike—to quickly grasp concepts without losing them in details, but also without dumbing down the material, and with enough of the correct references for further reading. I especially enjoyed the material on Java streams, concurrency and NIO.”

— **Michael Simons, Java Champion and engineer at Neo4j, author of the German Spring Boot Book reference**

“If you want to develop software using Java, this book is for you. *Head First Java* designs lots of straightforward and elegant examples to help the readers understand and learn how to use Java to create software. This is a great first book for anyone who wants to be a Java programmer.”

— **Sanhong Li, Alibaba Cloud**

## More praise for *Head First Java*, 3rd Edition

“*Head First Java* is a technical book that doesn’t feel like a technical book: it’s fun, casual, and full of worldly analogies that introduce complex concepts in a very smooth way. It’s the perfect introduction to the rich and vast Java ecosystem.”

— **Abraham Marin-Perez, Principal Consultant, Cosota Team**

“For those who like a little whimsy and humor with their “work”, I can think of no better book for learning Java than *Head First Java*, 3rd edition. Practical and playful, educational and engaging, it’s the perfect guide for new developers looking to hit the ground running.”

— **Marc Loy, trainer, author of *Smaller C*, and co-author of *Learning Java and Java Swing***

## Praise for earlier editions of *Head First Java*, and for other books coauthored by Kathy and Bert

“Kathy and Bert’s *Head First Java* transforms the printed page into the closest thing to a GUI you’ve ever seen. In a wry, hip manner, the authors make learning Java an engaging ‘what’re they gonna do next?’ experience.”

— **Warren Keuffel, Software Development Magazine**

“...the only way to decide the worth of a tutorial is to decide how well it teaches. *Head First Java* excels at teaching. OK, I thought it was silly...then I realized that I was thoroughly learning the topics as I went through the book.... The style of *Head First Java* made learning, well, easier.”

— **slashdot (honestpuck's review)**

“Beyond the engaging style that drags you forward from know-nothing into exalted Java warrior status, *Head First Java* covers a huge amount of practical matters that other texts leave as the dreaded “exercise for the reader...” It’s clever, wry, hip and practical—there aren’t a lot of textbooks that can make that claim and live up to it while also teaching you about object serialization and network launch protocols.”

— **Dr. Dan Russell, Director of User Sciences and Experience Research IBM Almaden Research Center (and teaches Artificial Intelligence at Stanford University)**

“It’s fast, irreverent, fun, and engaging. Be careful—you might actually learn something!”

— **Ken Arnold, former Senior Engineer at Sun Microsystems and coauthor (with James Gosling, creator of Java) of *The Java Programming Language***

“Java technology is everywhere. If you develop software and haven’t learned Java, it’s definitely time to dive in—Head First.”

— **Scott McNealy, former Sun Microsystems Chairman, President, and CEO**

“*Head First Java* is like Monty Python meets the gang of four...the text is broken up so well by puzzles and stories, quizzes and examples, that you cover ground like no computer book before.”

— **Douglas Rowe, Columbia Java Users Group**

“Read *Head First Java* and you will once again experience fun in learning..For people who like to learn new programming languages, and do not come from a computer science or programming background, this book is a gem...This is one book that makes learning a complex computer language fun.”

— **Judith Taylor, Southeast Ohio Macromedia User Group**

“If you want to learn Java, look no further: welcome to the first GUI-based technical book! This perfectly-executed, ground-breaking format delivers benefits other Java texts simply can’t...Prepare yourself for a truly remarkable ride through Java land.”

— **Neil R. Bauman, Captain and CEO, Geek Cruises**

“I was ADDICTED to the book’s short stories, annotated code, mock interviews, and brain exercises.”

— **Michael Yuan, author of *Enterprise J2ME***

“*Head First Java* gives new meaning to their marketing phrase ‘There’s an O’Reilly for that.’ I picked this up because several others I respect had described it in terms like ‘revolutionary’ and described a radically different approach to the textbook. They were (are) right...In typical O’Reilly fashion, they’ve taken a scientific and well considered approach. The result is funny, irreverent, topical, interactive, and brilliant...Reading this book is like sitting in the speakers lounge at a conference, learning from—and laughing with—peers...If you want to UNDERSTAND Java, go buy this book.”

— **Andrew Pollack, [www.thenorth.com](http://www.thenorth.com)**

“This stuff is so fricking good it makes me wanna WEEP! I’m stunned.”

— **Floyd Jones, Senior Technical Writer/Poolboy, BEA**

“I feel like a thousand pounds of books have just been lifted off of my head.”

— **Ward Cunningham, inventor of the Wiki and founder of the Hillside Group**

“I laughed, I cried, it moved me.”

— **Dan Steinberg, Editor-in-Chief, [java.net](http://java.net)**

“My first reaction was to roll on the floor laughing. After I picked myself up, I realized that not only is the book technically accurate, it is the easiest to understand introduction to design patterns that I have seen.”

— **Dr. Timothy A. Budd, Associate Professor of Computer Science at Oregon State University; author of more than a dozen books including *C++ for Java Programmers***

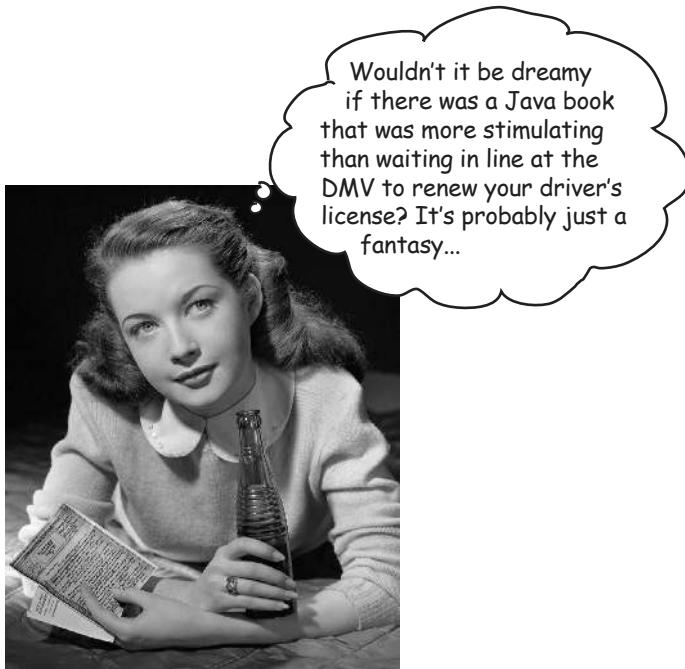
“Just the right tone for the geeked-out, casual-cool guru coder in all of us. The right reference for practical development strategies—gets my brain going without having to slog through a bunch of tired stale professor-speak.”

— **Travis Kalanick, founder of Scour and Red Swoosh, member of the MIT TR100**



# Head First Java™

Third Edition



Wouldn't it be dreamy  
if there was a Java book  
that was more stimulating  
than waiting in line at the  
DMV to renew your driver's  
license? It's probably just a  
fantasy...

Kathy Sierra  
Bert Bates  
Trisha Gee

Beijing • Boston • Farnham • Sebastopol • Tokyo

**O'REILLY®**

# **Head First Java™**

## **Third Edition**

by Kathy Sierra, Bert Bates, and Trisha Gee

Copyright © 2022 by Kathy Sierra and Bert Bates. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly Media books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles ([oreilly.com](http://oreilly.com)). For more information, contact our corporate/institutional sales department: (800) 998-9938 or [corporate@oreilly.com](mailto:corporate@oreilly.com).

**Editor for 1st and 2nd Editions:** Mike Loukides

**Editors for 3rd Edition:** Suzanne McQuade, Nicole Taché

**Cover Design:** Susan Thompson, based on a series design by Ellie Volckhausen

**Cover Illustration:** José Marzan, Jr.

**Production Editor:** Kristen Brown

**Original Interior Designers:** Kathy Sierra and Bert Bates

**3rd Edition Design Support:** Ron Bilodeau

**Java Whisperer:** Trisha Gee

**Series Advisors:** Eric Freeman, Elizabeth Robson

## **Printing History:**

May 2003: First Edition.

February 2005: Second Edition.

May 2022: Third Edition

(You might want to pick up a copy of *all* the editions...for your kids. Think eBay™)

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc., in the United States and other countries. O'Reilly Media, Inc. is independent of Sun Microsystems.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks.

Where those designations appear in this book, and O'Reilly Media, Inc. was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher and the authors assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

In other words, if you use anything in *Head First Java™* to, say, run a nuclear power plant or air traffic control system, you're on your own.

978-149-191077-1

[LSI]

[2022-05-11]

**From Kathy and Bert:**

To our brains, for always being there

(despite shaky evidence)

**From Trisha:**

To Isra, for always being there

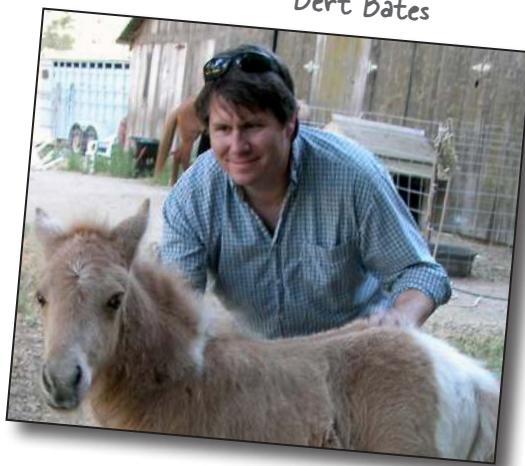
(with a surfeit of evidence)

## the authors

# Authors of *Head First Java* and Creators of the Head First series



Kathy Sierra



Bert Bates

**Kathy** has been interested in learning theory since her days as a game designer for Virgin, MGM, and Amblin', and a teacher of New Media Authoring at UCLA. She was a master Java trainer for Sun Microsystems, and she founded JavaRanch.com (now CodeRanch.com), which won Jolt Cola Productivity awards in 2003 and 2004.

In 2015, she won the Electronic Frontier Foundation's Pioneer Award for her work creating skillful users and building sustainable communities.

Kathy's recent focus has been on cutting-edge, movement science and skill acquisition coaching, known as ecological dynamics or "Eco-D." Her work using Eco-D for training horses is ushering in a far, far more humane approach to horsemanship, causing delight for some (and sadly, consternation for others). Those fortunate (autonomous!) horses whose owners are using Kathy's approach are happier, healthier, and more athletic than their fellows who are traditionally trained.

You can follow Kathy on Instagram:  
@pantherflows.

Before **Bert** was an author, he was a developer, specializing in old-school AI (mostly expert systems), real-time OSs, and complex scheduling systems.

In 2003, Bert and Kathy wrote *Head First Java* and launched the Head First series. Since then, he's written more Java books and consulted with Sun Microsystems and Oracle on many of their Java certifications. These days, he works with coaches, teachers, professors, authors, and editors, helping them create more bad-ass training for their students.

Bert's a Go player, and in 2016 he watched in horror and fascination as AlphaGo trounced Lee Sedol. Recently he's been using Eco-D (ecological dynamics) to improve his golf game and to train his parrotlet Bokeh.

Bert has been privileged to know Trisha Gee for more than eight years now, and the Head First series is extremely fortunate to count Trisha as one of its authors.

You can email Bert at bertbates.hf@gmail.com.

## Co-author of *Head First Java, 3rd Edition*



Trisha Gee

**Trisha** has been working with Java since 1997, when her university was forward-looking enough to adopt this “shiny new” language to teach computer science. Since then, she’s worked as a developer and consultant, creating Java applications in a range of industries including banking, manufacturing, nonprofit, and low-latency financial trading.

Trisha is super passionate about sharing all the stuff she learned the hard way during those years as a developer, so she became a Developer Advocate to give her an excuse to write blog posts, speak at conferences, and create videos to pass on some of her knowledge. She spent five years as a Java Developer Advocate at JetBrains, and another two years leading the JetBrains Java Advocacy team. During this time she learned a lot about the kinds of problems real Java developers face.

Trisha has been talking to Bert (on and off) about updating *Head First Java* for the last eight years! She remembers those weekly phone calls with Bert with great affection; regular contact with a knowledgeable and warm human being like Bert helped keep her sane. Bert and Kathy’s approach to encouraging learning has formed the core of what she’s been trying to do for nearly 10 years.

You can follow Trisha on Twitter: [@trisha\\_gee](https://twitter.com/trisha_gee).

# Table of Contents (summary)

Intro	xxi	
1	Breaking the Surface: <i>dive in: a quick dip</i>	1
2	A Trip to Objectville: <i>classes and objects</i>	27
3	Know Your Variables: <i>primitives and references</i>	49
4	How Objects Behave: <i>methods use instance variables</i>	71
5	Extra-Strength Methods: <i>writing a program</i>	95
6	Using the Java Library: <i>get to know the Java API</i>	125
7	Better Living in Objectville: <i>inheritance and polymorphism</i>	167
8	Serious Polymorphism: <i>interfaces and abstract classes</i>	199
9	Life and Death of an Object: <i>constructors and garbage collection</i>	237
10	Numbers Matter: <i>numbers and statics</i>	275
11	Data Structures: <i>collections and generics</i>	309
12	What, Not How: <i>lambdas and streams</i>	369
13	Risky Behavior: <i>exception handling</i>	421
14	A Very Graphic Story: <i>intro to GUI, event handling, and inner classes</i>	461
15	Work on Your Swing: <i>using swing</i>	509
16	Saving Objects (and Text): <i>serialization and file I/O</i>	539
17	Make a Connection: <i>networking and threads</i>	587
18	Dealing with Concurrency Issues: <i>race conditions and immutable data</i>	639
A	Appendix A: <i>final code kitchen</i>	673
B	Appendix B: <i>the top ten-ish topics that didn't make it into the rest of the book</i>	683
	Index	701

---

# Table of Contents (the real thing)



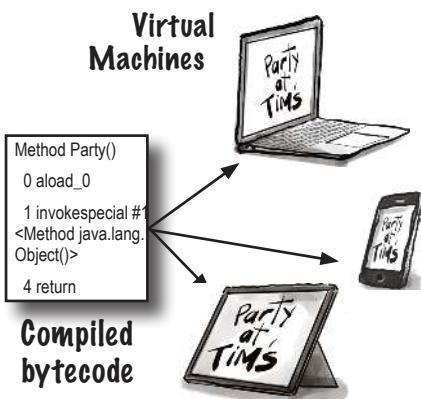
## Intro

**Your brain on Java.** Here you are trying to *learn* something, while here your *brain* is doing you a favor by making sure the learning doesn't stick. Your brain's thinking, "Better leave room for more important things, like which wild animals to avoid and whether naked snowboarding is a bad idea." So how do you trick your brain into thinking that your life depends on knowing Java?

Who is this book for?	xxvi
We know what you're thinking.	xxvii
Metacognition: thinking about thinking.	xxix
Here's what WE did	xxx
Here's what YOU can do to bend your brain into submission	xxxI
What you need for this book	xxxII
Last-minute things you need to know	xxxIII

# 1 Breaking the Surface

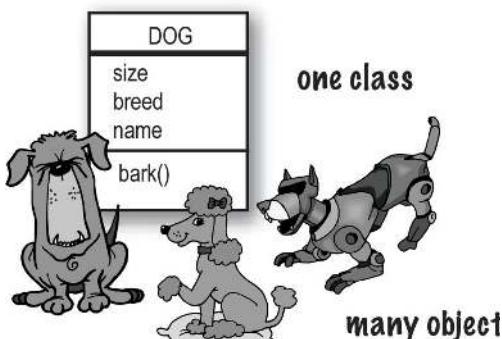
**Java takes you to new places.** From its humble release to the public as the (wimpy) version 1.02, Java seduced programmers with its friendly syntax, object-oriented features, memory management, and best of all—the promise of portability. We'll take a quick dip and write some code, compile it, and run it. We're talking syntax, loops, branching, and what makes Java so cool. Dive in.



The Way Java Works	2
What you'll do in Java	3
A Very Brief History of Java	4
Code structure in Java	7
Writing a class with a main()	9
Simple boolean tests	13
Conditional branching	15
Coding a Serious Business	16
Phrase-O-Matic	19
Exercises	20
Exercise Solutions	25

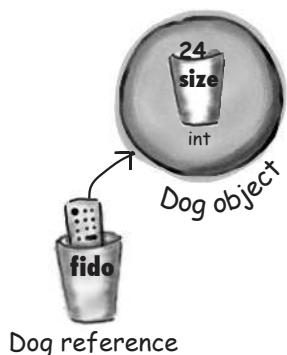
# 2 A Trip to Objectville

**I was told there would be objects.** In Chapter 1, we put all of our code in the main() method. That's not exactly object-oriented. So now we've got to leave that procedural world behind and start making some objects of our own. We'll look at what makes object-oriented (OO) development in Java so much fun. We'll look at the difference between a class and an object. We'll look at how objects can improve your life.



Chair Wars	28
Making your first object	36
Making and testing Movie objects	37
Quick! Get out of main!	38
Running the Guessing Game	40
Exercises	42
Exercise Solutions	47

# 3 Know Your Variables



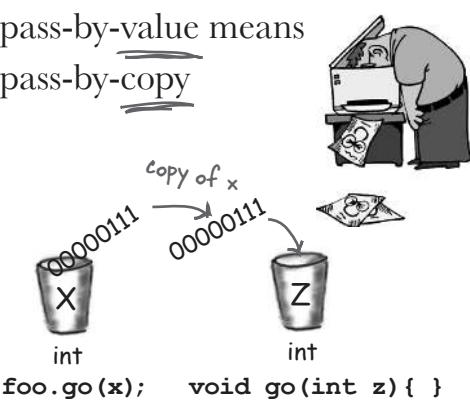
Declaring a variable	50
“I’d like a double mocha, no, make it an int.”	51
Back away from that keyword!	53
Controlling your Dog object	54
An object reference is just another variable value.	55
Life on the garbage-collectible heap	57
An array is like a tray of cups	59
A Dog example	62
Exercises	63
Exercise Solutions	68

---

# 4 How Objects Behave

**State affects behavior, behavior affects state.** We know that objects have **state** and **behavior**, represented by **instance variables** and **methods**. Now we’ll look at how state and behavior are *related*. An object’s behavior uses an object’s unique state. In other words, **methods use instance variable values**. Like, “if dog weight is less than 14 pounds, make yippy sound, else...” *Let’s go change some state!*

pass-by-value means  
pass-by-copy



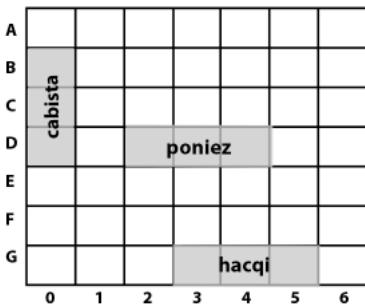
Remember: a class describes what an object knows and what an object does

The size affects the bark	73
You can send things to a method	74
You can get things <i>back</i> from a method.	75
You can send more than one thing to a method	76
Cool things you can do with parameters and return types	79
Encapsulation	80
How do objects in an array behave?	83
Declaring and initializing instance variables	84
Comparing variables (primitives or references)	86
Exercises	88
Exercise Solutions	93

# 5 Extra-Strength Methods

**Let's put some muscle in our methods.** You dabbled with variables, played with a few objects, and wrote a little code. But you need more tools. Like **operators**. And **loops**. Might be useful to **generate random numbers**. And **turn a String into an int**, yeah, that would be cool. And why don't we learn it all by *building something real*, to see what it's like to write (and test) a program from scratch. **Maybe a game**, like Sink a Startup (similar to Battleship).

We're gonna build the  
Sink a Startup game



Let's build a Battleship-style game: "Sink a Startup"	96
Developing a Class	99
Writing the method implementations	101
Writing test code for the SimpleStartup class	102
The checkYourself() method	104
Prep code for the SimpleStartupGame class	108
The game's main() method	110
Let's play	113
More about for loops	114
The enhanced for loop	116
Casting primitives	117
Exercises	118
Exercise Solutions	122

# 6 Using the Java Library

**Java ships with hundreds of prebuilt classes.** You don't have to reinvent the wheel if you know how to find what you need from the Java library, commonly known as the **Java API**. *You've got better things to do.* If you're going to write code, you might as well write *only* the parts that are custom for your application. The core Java library is a giant pile of classes just waiting for you to use like building blocks.

"Good to know there's an `ArrayList` in the `java.util` package. But by myself, how would I have figured that out?"

- Julia, 31, hand model



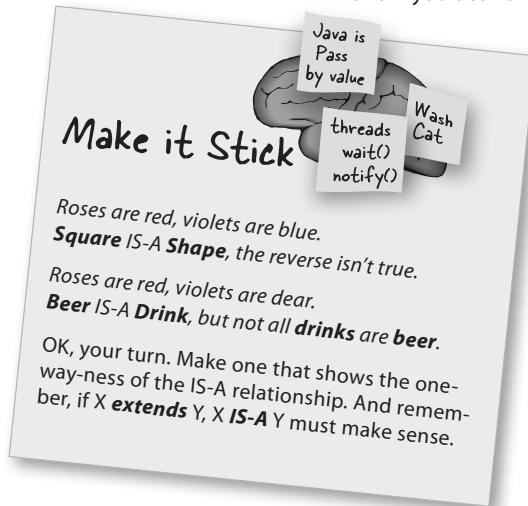
In our last chapter, we left you with the cliff-hanger. A bug.	126
Wake up and smell the library	132
Some things you can do with <code>ArrayList</code>	133
Comparing <code>ArrayList</code> to a regular array	137
Let's build the REAL game: "Sink a Startup"	140
Prep code for the real <code>StartupBust</code> class	144
The final version of the <code>Startup</code> class	150
Super Powerful Boolean Expressions	151
Using the Library (the Java API)	154
Exercises	163
Exercise Solutions	165

## 7

## Better Living in Objectville

**Plan your programs with the future in mind.** What if you could write

code that someone else could extend, **easily**? What if you could write code that was flexible, for those pesky last-minute spec changes? When you get on the Polymorphism Plan, you'll learn the 5 steps to better class design, the 3 tricks to polymorphism, the 8 ways to make flexible code, and if you act now—a bonus lesson on the 4 tips for exploiting inheritance.

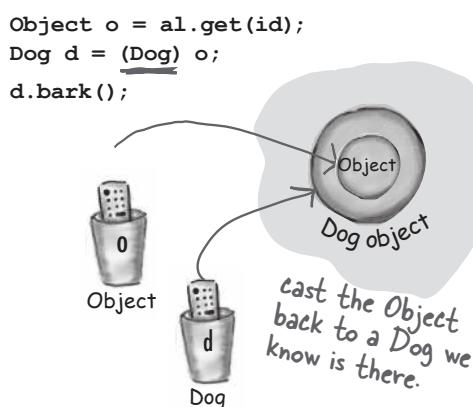


Chair Wars Revisited...	168
Understanding inheritance	170
Let's design the inheritance tree for an Animal simulation program	172
Looking for more inheritance opportunities	175
Using IS-A and HAS-A	179
How do you know if you've got your inheritance right?	181
When designing with inheritance, are you using or abusing?	183
Keeping the contract: rules for overriding	192
Overloading a method	193
Exercises	194
Exercise Solutions	197

## 8

## Serious Polymorphism

**Inheritance is just the beginning.** To exploit polymorphism, we need interfaces. We need to go beyond simple inheritance to flexibility you can get only by designing and coding to interfaces. What's an interface? A 100% abstract class. What's an abstract class? A class that can't be instantiated. What's that good for? Read the chapter...



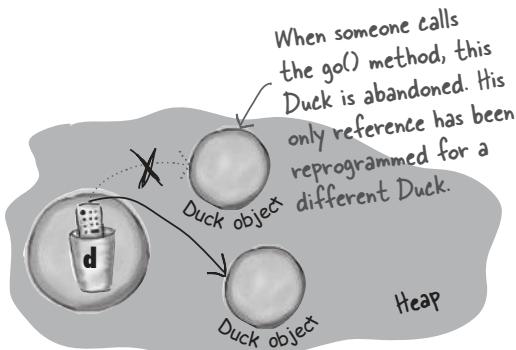
Did we forget about something when we designed this?	200
The compiler won't let you instantiate an abstract class	203
Abstract vs. Concrete	204
You MUST implement all abstract methods	206
Polymorphism in action	208
Why not make a class generic enough to take anything?	210
When a Dog won't act like a Dog	214
Let's explore some design options	221
Making and Implementing the Pet interface	227
Invoking the superclass version of a method	230
Exercises	232
Exercise Solutions	235



## 9

## Life and Death of an Object

**Objects are born and objects die.** You're in charge. You decide when and how to *construct* them. You decide when to *abandon* them. The **Garbage Collector (gc)** reclaims the memory. We'll look at how objects are created, where they live, and how to keep or abandon them efficiently. That means we'll talk about the heap, the stack, scope, constructors, super constructors, null references, and gc eligibility.



'd' is assigned a new Duck object, leaving the original (first) Duck object abandoned. That first Duck is toast.

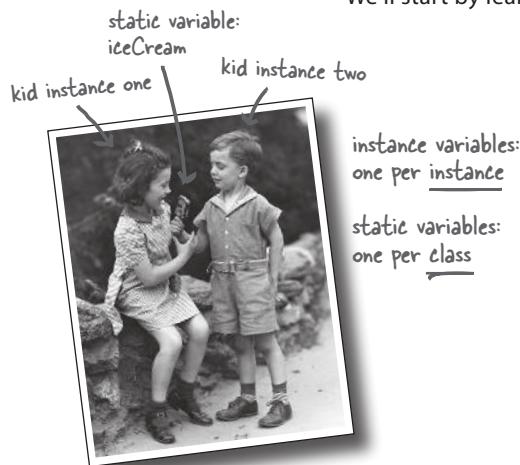
The Stack and the Heap: where things live	238
Methods are stacked	239
What about local variables that are objects?	240
The miracle of object creation	242
Construct a Duck	244
Doesn't the compiler always make a no-arg constructor for you?	248
Nanoreview: four things to remember about constructors	251
The role of superclass constructors in an object's life	253
Can the child exist before the parents?	256
What about reference variables?	262
I don't like where this is headed.	263
Exercises	268
Exercise Solutions	272

## 10

## Numbers Matter

**Do the Math.** The Java API has methods for absolute value, rounding, min/max, etc. But what about formatting? You might want numbers to print exactly two decimal points, or with commas in all the right places. And you might want to print and manipulate dates, too. And what about parsing a String into a number? Or turning a number into a String? We'll start by learning what it means for a variable or method to be *static*.

**Static variables  
are shared by  
all instances of  
a class.**

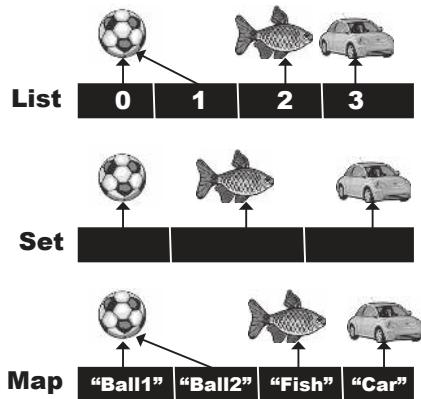


MATH methods: as close as you'll ever get to a <i>global</i> method	276
The difference between regular (non-static) and static methods	277
Initializing a static variable	283
Math methods	288
Wrapping a primitive	290
Autoboxing works almost everywhere	292
Turning a primitive number into a String	295
Number formatting	296
The format specifier	300
Exercise	306
Exercise Solutions	308

# 11

## Data Structures

**Sorting is a snap in Java.** You have all the tools for collecting and manipulating your data without having to write your own sort algorithms. The Java Collections Framework has a data structure that should work for virtually anything you'll ever need to do. Want to keep a list that you can easily keep adding to? Want to find something by name? Want to create a list that automatically takes out all the duplicates? Sort your co-workers by the number of times they've stabbed you in the back?

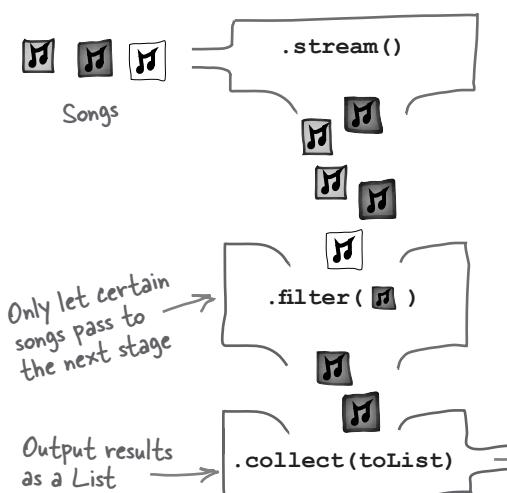


Exploring the java.util API, List and Collections	314
Generics means more type-safety	320
Revisiting the sort() method	327
The new, improved, comparable Song class	330
Sorting using only Comparators	336
Updating the Jukebox code with Lambdas	342
Using a HashSet instead of ArrayList	347
What you MUST know about TreeSet...	353
We've seen Lists and Sets, now we'll use a Map	355
Finally, back to generics	358
Exercise Solutions	364

# 12

## Lambdas and Streams: What, Not How

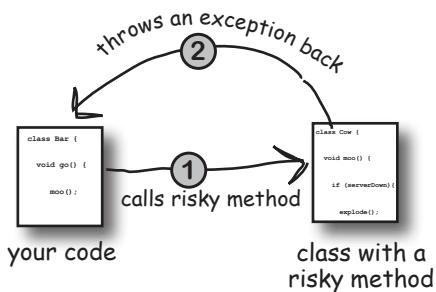
**What if...you didn't need to tell the computer HOW to do something?** In this chapter we'll look at the Streams API. You'll see how helpful lambda expressions can be when you're using streams, and you'll learn how to use the Streams API to query and transform the data in a collection.



Tell the computer WHAT you want	370
When for loops go wrong	372
Introducing the Streams API	375
Getting a result from a Stream	378
Guidelines for working with streams	384
Hello Lambda, my (not so) old friend	388
Spotting Functional Interfaces	396
Lou's Challenge #1: Find all the "rock" songs	400
Lou's Challenge #2: List all the genres	404
Exercises	415
Exercise Solutions	417

# 13 Risky Behavior

**Stuff happens.** The file isn't there. The server is down. No matter how good a programmer you are, you can't control *everything*. When you write a risky method, you need code to handle the bad things that might happen. But how do you *know* when a method is risky? Where do you put the code to *handle* the **exceptional** situation? In *this* chapter, we're going to build a MIDI Music Player that uses the risky JavaSound API, so we better find out.



Let's make a Music Machine	422
First we need a Sequencer	424
An exception is an object...of type Exception	428
Flow control in try/catch blocks	432
Did we mention that a method can throw more than one exception?	435
Multiple catch blocks must be ordered from smallest to biggest	438
Ducking (by declaring) only delays the inevitable	442
Code Kitchen	445
Version 1: Your very first sound player app	448
Version 2: Using command-line args to experiment with sounds	452
Exercises	454
Exercise Solutions	457

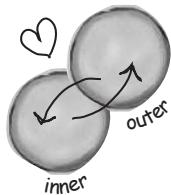
# 14 A Very Graphic Story

**Face it, you need to make GUIs.** Even if you believe that for the rest of your life you'll write only server-side code, sooner or later you'll need to write tools, and you'll want a graphical interface. We'll spend two chapters on GUIs and learn more language features including **Event Handling** and **Inner Classes**. We'll put a button on the screen, we'll paint on the screen, we'll display a JPEG image, and we'll even do some animation.

```
class MyOuter {
    class MyInner {
        void go() {
        }
    }
}
```

The outer and inner objects are now intimately linked.

These two objects on the heap have a special bond. The inner can use the outer's variables (and vice versa).



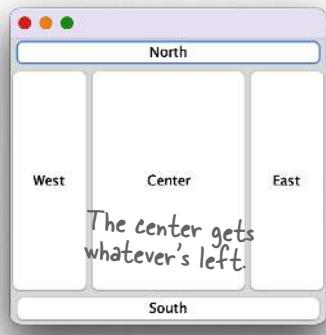
It all starts with a window	462
Getting a user event	465
Listeners, Sources, and Events	469
Make your own drawing widget	472
Fun things to do in paintComponent()	473
GUI layouts: putting more than one widget on a frame	478
Inner class to the rescue!	484
lambdas to the rescue! (again)	490
Using an inner class for animation	492
An easier way to make messages/events	498
Exercises	502
Exercise Solutions	507

## 15

**Work on Your Swing**

Components in the east and west get their preferred width.

Things in the north and south get their preferred height.



**Swing is easy.** Unless you actually *care* where everything goes. Swing code *looks* easy, but then compile it, run it, look at it, and think, “hey, *that’s not supposed to go there*.” The thing that makes it *easy to code* is the thing that makes it *hard to control*—the **Layout Manager**. But with a little work, you can get layout managers to submit to your will. In this chapter, we’ll work on our Swing and learn more about widgets.

Swing components	510
Layout Managers	511
The Big Three layout managers: border, flow, and box.	513
Playing with Swing components	523
Code Kitchen	526
Making the BeatBox	529
Exercises	534
Exercise Solutions	537

## 16

**Saving Objects (and Text)**

**Objects can be flattened and inflated.** Objects have state and behavior.

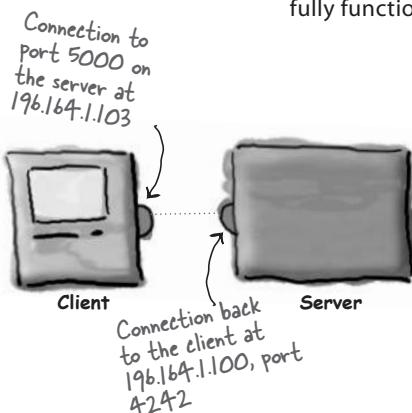
Behavior lives in the class, but *state* lives within each individual *object*. If your program needs to save state, *you can do it the hard way*, interrogating each object, painstakingly writing the value of each instance variable. Or, **you can do it the easy OO way**—you simply freeze-dry the object (serialize it) and reconstitute (deserialize) it to get it back.

Any questions?



Writing a serialized object to a file	542
If you want your class to be serializable, implement Serializable	547
Deserialization: restoring an object	551
Version ID: A Big Serialization Gotcha	556
Writing a String to a Text File	559
Reading from a Text File	566
Quiz Card Player (code outline)	567
Path, Paths, and Files (messing with directories)	573
Finally, a closer look at finally	574
Saving a BeatBox pattern	579
Exercises	580
Exercise Solutions	584

# 17 Make a Connection

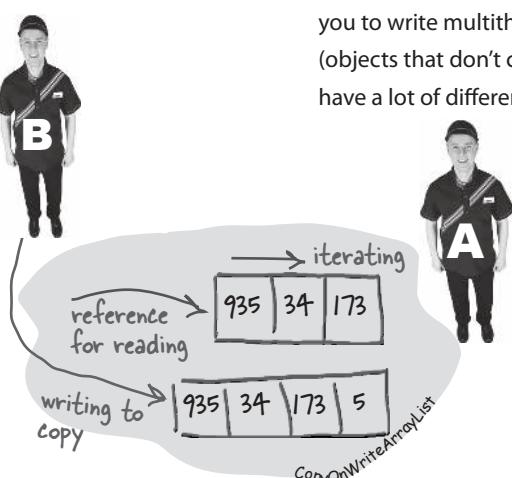


**Connect with the outside world.** It's easy. All the low-level networking details are taken care of by classes in the `java.net` library. One of Java's best features is that sending and receiving data over a network is really just I/O with a slightly different connection stream at the end of the chain. In this chapter we'll make client sockets. We'll make server sockets. We'll make clients and servers. Before the chapter's done, you'll have a fully functional, multithreaded chat client. Did we just say *multithreaded*?

Connecting, Sending, and Receiving	590
The DailyAdviceClient	598
Writing a simple server application	601
Java has multiple threads but only one Thread class	610
The three states of a new thread	616
Putting a thread to sleep	622
Making and starting two threads (or more!)	626
Closing time at the thread pool	629
New and improved SimpleChatClient	632
Exercises	631
Exercise Solutions	636

# 18 Dealing with Concurrency Issues

**Doing two or more things at once is hard.** Writing multithreaded code is easy. Writing multithreaded code that works the way you expect can be much harder. In this final chapter, we're going to show you some of the things that can go wrong when two or more threads are working at the same time. You'll learn about some of the tools in `java.util.concurrent` that can help you to write multithreaded code that works correctly. You'll learn how to create immutable objects (objects that don't change) that are safe for multiple threads to use. By the end of the chapter, you'll have a lot of different tools in your toolkit for working with concurrency.

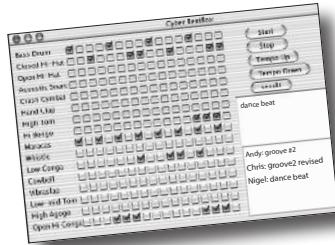


The Ryan and Monica problem, in code	642
Using an object's lock	647
The dreaded "Lost Update" problem	650
Make the increment() method atomic. Synchronize it!	652
Deadlock, a deadly side of synchronization	654
Compare-and-swap with atomic variables	656
Using immutable objects	659
More problems with shared data	662
Use a thread-safe data structure	664
Exercises	668
Exercise Solutions	670

# A

## Appendix A

**Final Code Kitchen.** All the code for the full client-server chat beat box. Your chance to be a rock star.



Final BeatBox client program	674
Final BeatBox server program	681

# B

## Appendix B

**The top ten-ish topics that didn't make it into the rest of the book.** We can't send you out into the world just yet. We have a few more things for you, but this *is* the end of the book. And this time we really mean it.

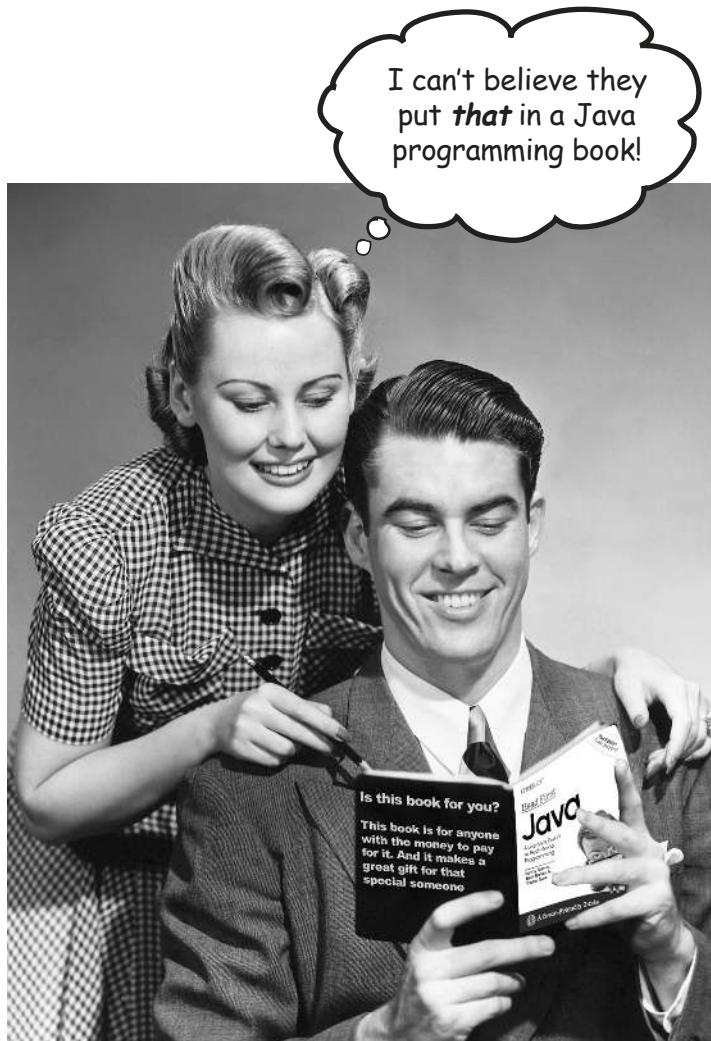
#11 JShell (Java REPL)	684
#10 Packages	685
#9 Immutability in Strings and Wrappers	688
#8 Access levels and access modifiers (who sees what)	689
#7 Varargs	691
#6 Annotations	692
#5 Lambdas and Maps	693
#4 Parallel Streams	695
#3 Enumerations (also called enumerated types or enums)	696
#2 Local Variable Type Inference (var)	698
#1 Records	699

# i

## Index

how to use this book

# Intro



In this section, we answer the burning question:  
"So, why DID they put that in a Java programming book?"

## Who is this book for?

If you can answer “yes” to *all* of these:

- ① **Have you done some programming?**
- ② **Do you want to learn Java?**
- ③ **Do you prefer stimulating dinner party conversation to dry, dull, technical lectures?**

this book is for you.

**This is NOT a reference book. *Head First Java* is a book designed for *learning*, not an encyclopedia of Java facts.**

## Who should probably back away from this book?

If you can answer “yes” to any *one* of these:

- ① **Is your programming background limited to HTML only, with no scripting language experience?**  
(If you've done anything with looping or if/then logic, you'll do fine with this book, but HTML tagging alone might not be enough.)
- ② **Are you a kick-butt C++ programmer looking for a *reference* book?**
- ③ **Are you afraid to try something different? Would you rather have a root canal than mix stripes with plaid? Do you believe that a technical book can't be serious if there's a picture of a duck in the memory management section?**

this book is *not* for you.



[note from marketing: who took out the part about how this book is for anyone with a valid credit card? And what about that “Give the Gift of Java” holiday promotion we discussed... -Fred]

# We know what you're thinking

“How can *this* be a serious Java programming book?”

“What’s with all the graphics?”

“Can I actually *learn* it this way?”

“Do I smell pizza?”



## And we know what your *brain* is thinking

Your brain craves novelty. It’s always searching, scanning, *waiting* for something unusual. It was built that way, and it helps you stay alive.

Today, you’re less likely to be a tiger snack. But your brain’s still looking. You just never know.

So what does your brain do with all the routine, ordinary, normal things you encounter? Everything it *can* to stop them from interfering with the brain’s *real* job—recording things that *matter*. It doesn’t bother saving the boring things; they never make it past the “this is obviously not important” filter.

How does your brain *know* what’s important? Suppose you’re out for a day hike and a tiger jumps in front of you, what happens inside your head?

Neurons fire. Emotions crank up. *Chemicals surge*.

And that’s how your brain knows...

### This must be important! Don’t forget it!

But imagine you’re at home, or in a library. It’s a safe, warm, tiger-free zone. You’re studying. Getting ready for an exam. Or trying to learn some tough technical topic your boss thinks will take a week, ten days at the most.

Just one problem. Your brain’s trying to do you a big favor. It’s trying to make sure that this *obviously* non-important content doesn’t clutter up scarce resources. Resources that are better spent storing the really *big* things. Like tigers. Like the danger of fire. Like how you should never again snowboard in shorts.

And there’s no simple way to tell your brain, “Hey, brain, thank you very much, but no matter how dull this book is, and how little I’m registering on the emotional richter scale right now, I really *do* want you to keep this stuff around.”



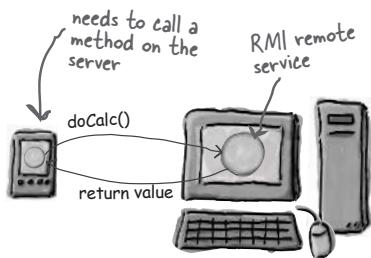
## We think of a Head First Java reader as a learner.

So what does it take to *learn* something? First, you have to *get it*, then make sure you don't *forget it*. It's not about pushing facts into your head. Based on the latest research in cognitive science, neurobiology, and educational psychology, learning takes a lot more than text on a page. We know what turns your brain on.

### Some of the Head First learning principles:



**Make it visual.** Images are far more memorable than words alone, and make learning much more effective (up to 89% improvement in recall and transfer studies). It also makes things more understandable. **Put the words within or near the graphics** they relate to, rather than on the bottom or on another page, and learners will be up to twice as likely to solve problems related to the content.



It really sucks to be an abstract method. You don't have a body.

**Use a conversational and personalized style.** In recent studies, students performed up to 40% better on post-learning tests if the content spoke directly to the reader, using a first-person, conversational style rather than taking a formal tone. Tell stories instead of lecturing. Use casual language. Don't take yourself too seriously. Which would you pay more attention to: a stimulating dinner party companion, or a lecture?



abstract void roam();

No method body!  
End it with a semicolon.

**Get the learner to think more deeply.** In other words, unless you actively flex your neurons, nothing much happens in your head. A reader has to be motivated, engaged, curious, and inspired to solve problems, draw conclusions, and generate new knowledge.

And for that, you need challenges, exercises, thought-provoking questions, and activities that involve both sides of the brain and multiple senses.

Does it make sense to say Tub IS-A Bathroom? Bathroom IS-A Tub? Or is it a HAS-A relationship?



**Get—and keep—the reader's attention.** We've all had the "I really want to learn this but I can't stay awake past page one" experience. Your brain pays attention to things that are out of the ordinary, interesting, strange, eye-catching, unexpected. Learning a new, tough, technical topic doesn't have to be boring. Your brain will learn much more quickly if it's not.

**Touch their emotions.** We now know that your ability to remember something is largely dependent on its emotional content. You remember what you care about. You remember when you feel something. No we're not talking heart-wrenching stories about a boy and his dog. We're talking emotions like surprise, curiosity, fun, "what the...?", and the feeling of "I Rule!" that comes when you solve a puzzle, learn something everybody else thinks is hard, or realize you know something that "I'm more technical than thou" Bob from engineering doesn't.



# Metacognition: thinking about thinking

If you really want to learn, and you want to learn more quickly and more deeply, pay attention to how you pay attention. Think about how you think. Learn how you learn.

Most of us did not take courses on metacognition or learning theory when we were growing up. We were *expected* to learn, but rarely *taught* to learn.

But we assume that if you're holding this book, you want to learn Java. And you probably don't want to spend a lot of time.

To get the most from this book, or *any* book or learning experience, take responsibility for your brain. Your brain on *that* content.

The trick is to get your brain to see the new material you're learning as Really Important. Crucial to your well-being. As important as a tiger.

Otherwise, you're in for a constant battle, with your brain doing its best to keep the new content from sticking.

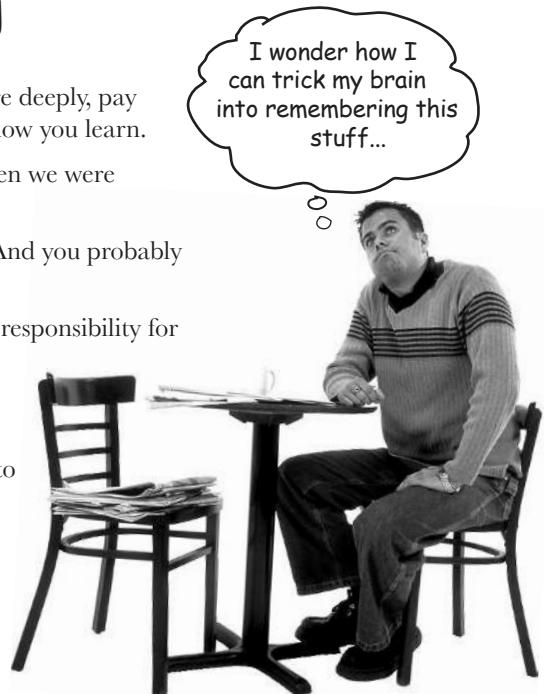
## So just how **DO** you get your brain to treat Java like it was a hungry tiger?

There's the slow, tedious way, or the faster, more effective way. The slow way is about sheer repetition. You obviously know that you *are* able to learn and remember even the dullest of topics, if you keep pounding on the same thing. With enough repetition, your brain says, "This doesn't *feel* important to him, but he keeps looking at the same thing *over* and *over* and *over*, so I suppose it must be."

The faster way is to do **anything that increases brain activity**, especially different *types* of brain activity. The things on the previous page are a big part of the solution, and they're all things that have been proven to help your brain work in your favor. For example, studies show that putting words *within* the pictures they describe (as opposed to somewhere else in the page, like a caption or in the body text) causes your brain to try to make sense of how the words and picture relate, and this causes more neurons to fire. More neurons firing = more chances for your brain to *get* that this is something worth paying attention to, and possibly recording.

A conversational style helps because people tend to pay more attention when they perceive that they're in a conversation, since they're expected to follow along and hold up their end. The amazing thing is, your brain doesn't necessarily *care* that the "conversation" is between you and a book! On the other hand, if the writing style is formal and dry, your brain perceives it the same way you experience being lectured to while sitting in a roomful of passive attendees. No need to stay awake.

But pictures and conversational style are just the beginning.



## Here's what WE did:

We used **pictures**, because your brain is tuned for visuals, not text. As far as your brain's concerned, a picture really *is* worth 1024 words. And when text and pictures work together, we embedded the text *in* the pictures because your brain works more effectively when the text is *within* the thing the text refers to, as opposed to in a caption or buried in the text somewhere.

We used **repetition**, saying the same thing in different ways and with different media types, and *multiple senses*, to increase the chance that the content gets coded into more than one area of your brain.

We used concepts and pictures in **unexpected** ways because your brain is tuned for novelty, and we used pictures and ideas with at least *some emotional content*, because your brain is tuned to pay attention to the biochemistry of emotions. That which causes you to *feel* something is more likely to be remembered, even if that feeling is nothing more than a little **humor, surprise, or interest**.

We used a personalized, **conversational style**, because your brain is tuned to pay more attention when it believes you're in a conversation than if it thinks you're passively listening to a presentation. Your brain does this even when you're *reading*.

We included more than 50 **exercises** because your brain is tuned to learn and remember more when you **do** things than when you *read* about things. And we made the exercises challenging-yet-do-able, because that's what most *people* prefer.

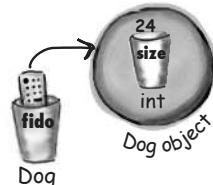
We used **multiple learning styles**, because *you* might prefer step-by-step procedures, while someone else wants to understand the big picture first, while someone else just wants to see a code example. But regardless of your own learning preference, *everyone* benefits from seeing the same content represented in multiple ways.

We include content for **both sides of your brain**, because the more of your brain you engage, the more likely you are to learn and remember, and the longer you can stay focused. Since working one side of the brain often means giving the other side a chance to rest, you can be more productive at learning for a longer period of time.

And we included **stories** and exercises that present **more than one point of view**, because your brain is tuned to learn more deeply when it's forced to make evaluations and judgments.

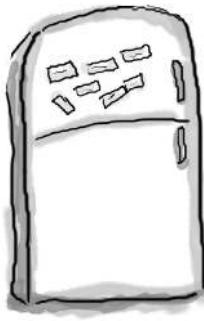
We included **challenges**, with exercises, and by asking **questions** that don't always have a straight answer, because your brain is tuned to learn and remember when it has to *work* at something (just as you can't get your *body* in shape by watching people at the gym). But we did our best to make sure that when you're working hard, it's on the *right* things. That **you're not spending one extra dendrite** processing a hard-to-understand example, or parsing difficult, jargon-laden, or extremely terse text.

We used an **80/20** approach. We assume that if you're going for a PhD in Java, this won't be your only book. So we don't talk about *everything*. Just the stuff you'll actually *use*.



## BULLET POINTS





# Here's what YOU can do to bend your brain into submission

So, we did our part. The rest is up to you. These tips are a starting point; Listen to your brain and figure out what works for you and what doesn't. Try new things.

*Cut this out and stick it on your refrigerator.*



## ① Slow down. The more you understand, the less you have to memorize.

Don't just *read*. Stop and think. When the book asks you a question, don't just skip to the answer. Imagine that someone really *is* asking the question. The more deeply you force your brain to think, the better chance you have of learning and remembering.

## ② Do the exercises. Write your own notes.

We put them in, but if we did them for you, that would be like having someone else do your workouts for you. And don't just *look* at the exercises. **Use a pencil.** There's plenty of evidence that physical activity *while* learning can increase the learning.

## ③ Read the “There are No Dumb Questions”

That means all of them. They're not optional sidebars—they're part of the core content! Sometimes the questions are more useful than the answers.

## ④ Don't do all your reading in one place.

Stand up, stretch, move around, change chairs, change rooms. It'll help your brain *feel* something, and it keeps your learning from being too connected to a particular place.

## ⑤ Make this the last thing you read before bed. Or at least the last challenging thing.

Part of the learning (especially the transfer to long-term memory) happens *after* you put the book down. Your brain needs time on its own, to do more processing. If you put in something new during that processing time, some of what you just learned will be lost.

## ⑥ Drink water. Lots of it.

Your brain works best in a nice bath of fluid. Dehydration (which can happen before you ever feel thirsty) decreases cognitive function.

## ⑦ Talk about it. Out loud.

Speaking activates a different part of the brain. If you're trying to understand something or increase your chance of remembering it later, say it out loud. Better still, try to explain it out loud to someone else. You'll learn more quickly, and you might uncover ideas you hadn't known were there when you were reading about it.

## ⑧ Listen to your brain.

Pay attention to whether your brain is getting overloaded. If you find yourself starting to skim the surface or forget what you just read, it's time for a break. Once you go past a certain point, you won't learn faster by trying to shove more in, and you might even hurt the process.

## ⑨ Feel something!

Your brain needs to know that this *matters*. Get involved with the stories. Make up your own captions for the photos. Groaning over a bad joke is *still* better than feeling nothing at all.

## ⑩ Type and run the code.

Type and run the code examples. Then you can experiment with changing and improving the code (or breaking it, which is sometimes the best way to figure out what's really happening). Most of the code, especially long examples and Ready-Bake Code, are at [https://oreil.ly/hfJava\\_3e\\_examples](https://oreil.ly/hfJava_3e_examples).

## What you need for this book:

You do *not* need any other development tool, such as an Integrated Development Environment (IDE). We strongly recommend that you *not* use anything but a basic text editor until you complete this book. An IDE can protect you from some of the details that really matter, so you're much better off learning from the command line and then, once you really understand what's happening, move to a tool that automates some of the process.

**This book assumes you're using Java 11 (with the exception of Appendix B). However, if you're using Java 8, you will find most of the code still works.**

**If there's discussion of a feature from a version of Java higher than Java 8, the required version will be mentioned.**

### SETTING UP JAVA

- Because versions are moving quickly and advice on the right JDK to use may change, we've put detailed instructions on how to install Java into the code samples project online:  
[https://oreil.ly/hfJava\\_install](https://oreil.ly/hfJava_install)  
This is a simplified version.
- If you don't know which version of Java to download, we recommend using Java 17.
- There are many free builds of OpenJDK available (the open source version of Java). We suggest the community-supported Eclipse Adoptium JDK at <https://adoptium.net>.
- The JDK includes everything you need to compile and run Java. The JDK does not include the API documentation, and you need that! Download the Java SE API documentation. You can also access the API docs online without downloading them, but trust us, it's worth the download.
- You need a text editor. Virtually any text editor will do (vi, emacs), including the GUI ones that come with most operating systems. Notepad, Wordpad,TextEdit, etc., all work, as long as you're using plain text (not rich text) and make sure they don't append a ".txt" on to the end of your source code ("java") file.
- Once you've downloaded and unpacked/installed/whatever (depends on which version and for which OS), you need to add an entry to your PATH environment variable that points to the *bin* directory inside the main Java directory. The *bin* directory is the one you need a PATH to, so that when you type:

```
% javac
```

at the command line, your terminal will know how to find the javac compiler.

- Note: if you have trouble with your installation, we recommend you go to [javaranch.com](http://javaranch.com) and join the Java-Beginning forum! Actually, you should do that whether you have trouble or not.



The code from this book is available at [https://oreil.ly/hfJava\\_3e\\_examples](https://oreil.ly/hfJava_3e_examples).

# Last-minute things you need to know:

This is a learning experience, not a reference book. We deliberately stripped out everything that might get in the way of *learning* whatever it is we're working on at that point in the book. And the first time through, you need to begin at the beginning, because the book makes assumptions about what you've already seen and learned.

## We use simple UML-like diagrams.

If we'd used *pure* UML, you'd be seeing something that *looks* like Java, but with syntax that's just plain *wrong*. So we use a simplified version of UML that doesn't conflict with Java syntax. If you don't already know UML, you won't have to worry about learning Java *and* UML at the same time.

## We don't worry about organizing and packaging your own code.

In this book, you can get on with the business of learning Java, without stressing over some of the organizational or administrative details of developing Java programs. You *will*, in the real world, need to know—and use—these details, but since building and deploying Java applications generally relies on third-party build tools like Maven and Gradle, we have assumed you'll learn those tools separately.

## The end-of-chapter exercises are mandatory; puzzles are optional. Answers for both are at the end of each chapter.

One thing you need to know about the puzzles—they're *puzzles*. As in logic puzzles, brain teasers, crossword puzzles, etc. The *exercises* are here to help you practice what you've learned, and you should do them all. The puzzles are a different story, and some of them are quite challenging in a *puzzle* way. These puzzles are meant for *puzzlers*, and you probably already know if you are one. If you're not sure, we suggest you give some of them a try, but whatever happens, don't be discouraged if you *can't* solve a puzzle or if you simply can't be bothered to take the time to work them out.

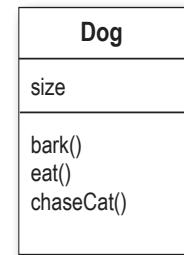
## The “Sharpen Your Pencil” exercises don't all have answers.

Not printed in the book, anyway. For some of them, there *is* no right answer, and for the others, part of the learning experience for the Sharpen activities is for *you* to decide if and when your answers are right.

## The code examples are as lean as possible.

It's frustrating to wade through 200 lines of code looking for the two lines you need to understand. Most examples in this book are shown within the smallest possible context, so that the part you're trying to learn is clear and simple. So don't expect the code to be robust, or even complete. That's *your* assignment for after you finish the book. The book examples are written specifically for *learning* and aren't always fully functional.

We use a simpler,  
modified faux-UML



You should do ALL  
of the “Sharpen your  
pencil” activities



Activities marked with the  
Exercise (running shoe) logo  
are mandatory! Don't skip  
them if you're serious about  
learning Java.



If you see the Puzzle logo, the  
activity is optional, and if you  
don't like twisty logic or cross-  
word puzzles, you won't like these  
either.



## Technical Reviewers for the 3rd Edition

Marc Loy



Abraham Marin-Perez



**Marc** started with Java training at Sun Microsystems in the early days (shout-out to HotJava!) and never looked back. He authored a number of early Java books and training courses, working with a wide variety of companies across the US, Europe, and Asia along the way. Most recently for O'Reilly, Marc authored *Smaller C* and co-authored the fifth edition of *Learning Java*. Currently in Ohio, Marc is a software developer and maker specializing in microcontrollers.

**Abraham** is a Java programmer, consultant, author, and public speaker with more than ten years of experience in a variety of industries. Originally from Valencia, Spain, Abraham has built most of his career in London, UK, working with entities like JP Morgan or the United Kingdom's Home Office, frequently in collaboration with Equal Experts. Thinking his experiences could be useful to others, Abraham became a Java news editor at InfoQ, authored *Real-World Maintainable Software*, and co-authored *Continuous Delivery in Java*. He also helps run the London Java Community. Always the learner, Abraham is pursuing a degree in physics.

# Other people to acknowledge for the 3rd Edition

## *At O'Reilly:*

Huge thanks to **Zan McQuade** and **Nicole Taché** for enabling us to finally get this edition out! Zan, thanks for connecting Trisha back up to the Head First world, and Nicole, fantastic work driving us to get this done. Thanks to **Meghan Blanchette**, who left O'Reilly a hundred years ago, but it was she who introduced Bert and Trisha back in 2014.

## *Trisha would like to thank:*

**Helen Scott**, for providing frequent feedback on the new topics covered. She consistently stopped me from going too deep or assuming too much knowledge, and is a true champion of the learner. I can't wait to start working even more closely with her on our next project.



My team at JetBrains for their patience and encouragement: **Dalia Abo Sheasha**, for test-driving the lambdas and streams chapter, and **Mala Gupta**, for giving me exactly the information I needed about modern Java certifications. Extra special thanks to **Hadi Hariri** for all his support, always.

The Friday Pub Lunch *informaticos*, for tolerating lunchtime conversations on whatever aspect of Java I was trying to explain that day or week, and **Alys**, **Jen**, and **Clare** for helping me to figure out when to prioritize this book over family. Thanks to **Holly Cummins** for finding a last minute bug.

**Evie** and **Amy** for the suggestions on how to improve the ice cream examples for Java's Optional type. Thank you both for being genuinely interested in my progress, and for the spontaneous high-fives when you heard I'd finished.

None of this would have been possible without **Israel Boza Rodriguez**. You put up with me derailing important conversations like "what should we have for dinner?" with questions like "do you think CountDownLatch is too niche to teach beginner developers?" Crucially, you helped me to create space and time to work on the book, and regularly reminded me why I wanted to take on the project in the first place.

Thank you to **Bert** and **Kathy** for bringing me on this journey. It was an honor to learn how to be a Head First author from the horse's mouth, so to speak.

## *Bert and Kathy would like to thank:*

**Beth Robson** and **Eric Freeman**, for their overall, ongoing, badass support of the Head First series. A special thanks to Beth for the many conversations we had discussing what new Java topics to teach and how to teach them.

**Paul Wheaton** and the amazing moderators at CodeRanch.com (a.k.a. JavaRanch), for keeping CodeRanch a friendly place for Java beginners. A special thanks to **Campbell Ritchie**, **Jeanne Boyarsky**, **Stephan van Hulst**, **Rob Spoor**, **Tim Cooke**, **Fred Rosenberger**, and **Frits Walraven** for their invaluable input concerning what have been the truly important additions to Java since the 2nd edition.

**Dave Gustafson**, for teaching me so much about software development and rock climbing, AND for great discussions concerning the state of programming. **Eric Normand**, for teaching us a little FP, and helping us figure out how to slip a few of the best ideas from FP into an OO book. **Simon Roberts**, for his ongoing and passionate teaching of Java to students all over the world. Thanks to **Heinz Kabutz** and **Venkat Subramaniam** for helping us explore the nooks and crannies of Java Streams.

**Laura Baldwin** and **Mike Loukides**, for their tireless support of Head First for all these years. **Ron Bilodeau** and **Kristen Brown**, for their outstanding, always patient and friendly support.

## Technical Editors for the 2nd Edition

Endless thanks to Jessica and Val for their hard work editing the 2nd edition.



**Jess** works at Hewlett-Packard on the Self-Healing Services Team. She has a bachelor's in computer engineering from Villanova University, has her SCJP 1.4 and SCWCD certifications, and is literally months away from receiving her master's in software engineering at Drexel University (whew!).

When she's not working, studying, or motoring in her MINI Cooper S, Jess can be found fighting her cat for yarn as she completes her latest knitting or crochet project (anybody want a hat?). She is originally from Salt Lake City, Utah (no, she's not Mormon...yes, you were too going to ask) and is currently living near Philadelphia with her husband, Mendra, and two cats: Chai and Sake.

You can catch her moderating technical forums at [javaranch.com](http://javaranch.com).



**Valentin** has a master's degree in information and computer science from the Swiss Federal Institute of Technology in Lausanne (EPFL). He has worked as a software engineer with SRI International (Menlo Park, CA) and as a principal engineer in the Software Engineering Laboratory of EPFL.

Valentin is the cofounder and CTO of Condris Technologies, a company specializing in the development of software architecture solutions.

His research and development interests include aspect-oriented technologies, design and architectural patterns, web services, and software architecture. Besides taking care of his wife, gardening, reading, and doing some sport, Valentin moderates the SCBCD and SCDJWS forums at Javaranch.com. He holds the SCJP, SCJD, SCBCD, SCWCD, and SCDJWS certifications. He has also had the opportunity to serve as a co-author for Whizlabs SCBCD Exam Simulator.

(We're still in shock from seeing him in a *tie*.)

credit, for the 2nd  
Edition

the intro

## Other people to blame:

### At O'Reilly:

Our biggest thanks to **Mike Loukides** at O'Reilly, for taking a chance on this, and helping to shape the Head First concept into a book (and series). As this second edition goes to print there are now five Head First books, and he's been with us all the way. To **Tim O'Reilly**, for his willingness to launch into something *completely* new and different. Thanks to the clever **Kyle Hart** for figuring out how Head First fits into the world and for launching the series. Finally, to **Edie Freedman** for designing the Head First "emphasize the *head*" cover.

### Our intrepid beta testers and reviewer team:

Our top honors and thanks go to the director of our javaranch tech review team, **Johannes de Jong**. This is your fifth time around with us on a Head First book, and we're thrilled you're still speaking to us. **Jeff Cumps** is on his third book with us now and relentless about finding areas where we needed to be more clear or correct.

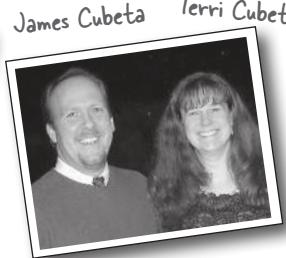
**Corey McGlone**, you rock. And we think you give the clearest explanations on JavaRanch. You'll probably notice we stole one or two of them. **Jason Menard** saved our technical butts on more than a few details, and **Thomas Paul**, as always, gave us expert feedback and found the subtle Java issues the rest of us missed. **Jane Griscti** has her Java chops (and knows a thing or two about *writing*), and it was great to have her helping on the new edition along with long-time javarancher **Barry Gaunt**.

**Marilyn de Queiroz** gave us excellent help on *both* editions of the book. **Chris Jones, John Nyquist, James Cubeta, Terri Cubeta**, and **Ira Becker** gave us a ton of help on the first edition.

Special thanks to a few of the Head Firsters who've been helping us from the beginning: **Angelo Celeste, Mikalai Zaikin**, and **Thomas Duff** ([twduff.com](http://twduff.com)). And thanks to our terrific agent, **David Rogelberg** of StudioB (but seriously, what about the *movie* rights?)



Rodney J.  
Woodruff



James Cubeta   Terri Cubeta



Ira Becker



John Nyquist



Chris Jones

Some of our Java  
expert reviewers...

Jef Cumps



Corey McGlone



Johannes de Jong



Jason Menard



Thomas Paul



Marilyn de  
Queiroz



## still more acknowledgments

# Just when you thought there wouldn't be any more acknowledgments\*

### ***More Java technical experts who helped out on the first edition (in pseudo-random order):***

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

### ***The first edition puzzle team:***

Dirk Schreckmann, Mary “JavaCross Champion” Leners, Rodney J. Woodruff, Gavin Bong, and Jason Menard. Javaranch is lucky to have you all helping out.

### ***Other co-conspirators to thank:***

**Paul Wheaton**, the javaranch Trail Boss for supporting thousands of Java learners.

**Solveig Haugland**, mistress of J2EE and author of *Dating Design Patterns*.

Authors **Dori Smith** and **Tom Negrino** ([backupbrain.com](http://backupbrain.com)), for helping us navigate the tech book world.

Our Head First partners in crime, **Eric Freeman and Beth Freeman** (authors of Head First Design Patterns), for giving us the Bawls™ to finish this on time.

**Sherry Dorris**, for the things that really matter.

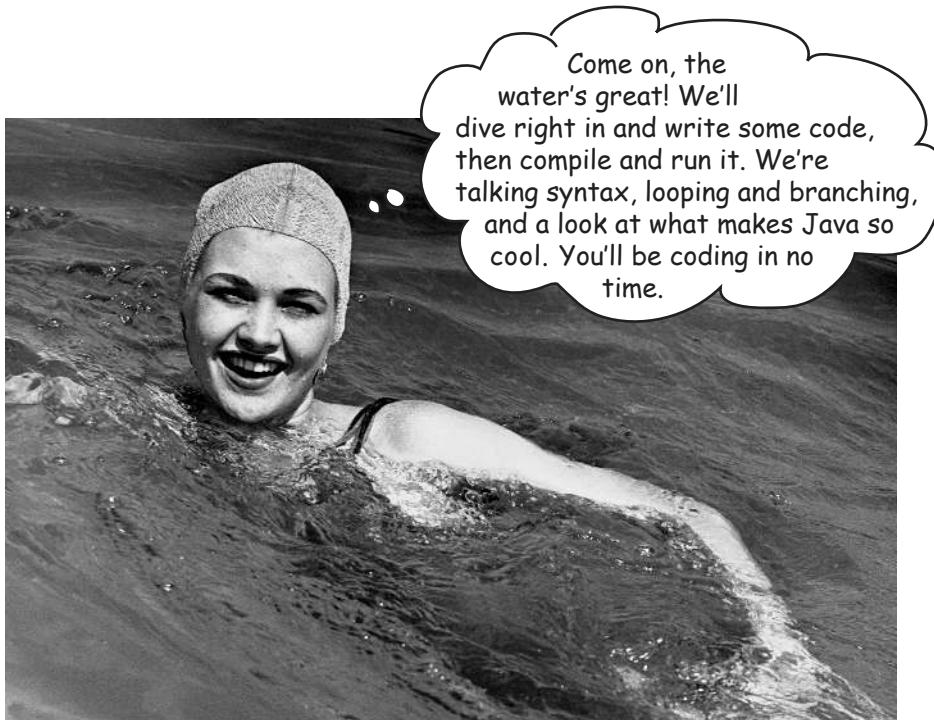
### ***Brave early adopters of the Head First series:***

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, and Mike Bibby (the first).

\*The large number of acknowledgments is because we're testing the theory that everyone mentioned in a book acknowledgment will buy at least one copy, probably more, what with relatives and everything. If you'd like to be in the acknowledgment of our *next* book, and you have a large family, write to us.

## 1 dive in: a quick dip

# Breaking the Surface

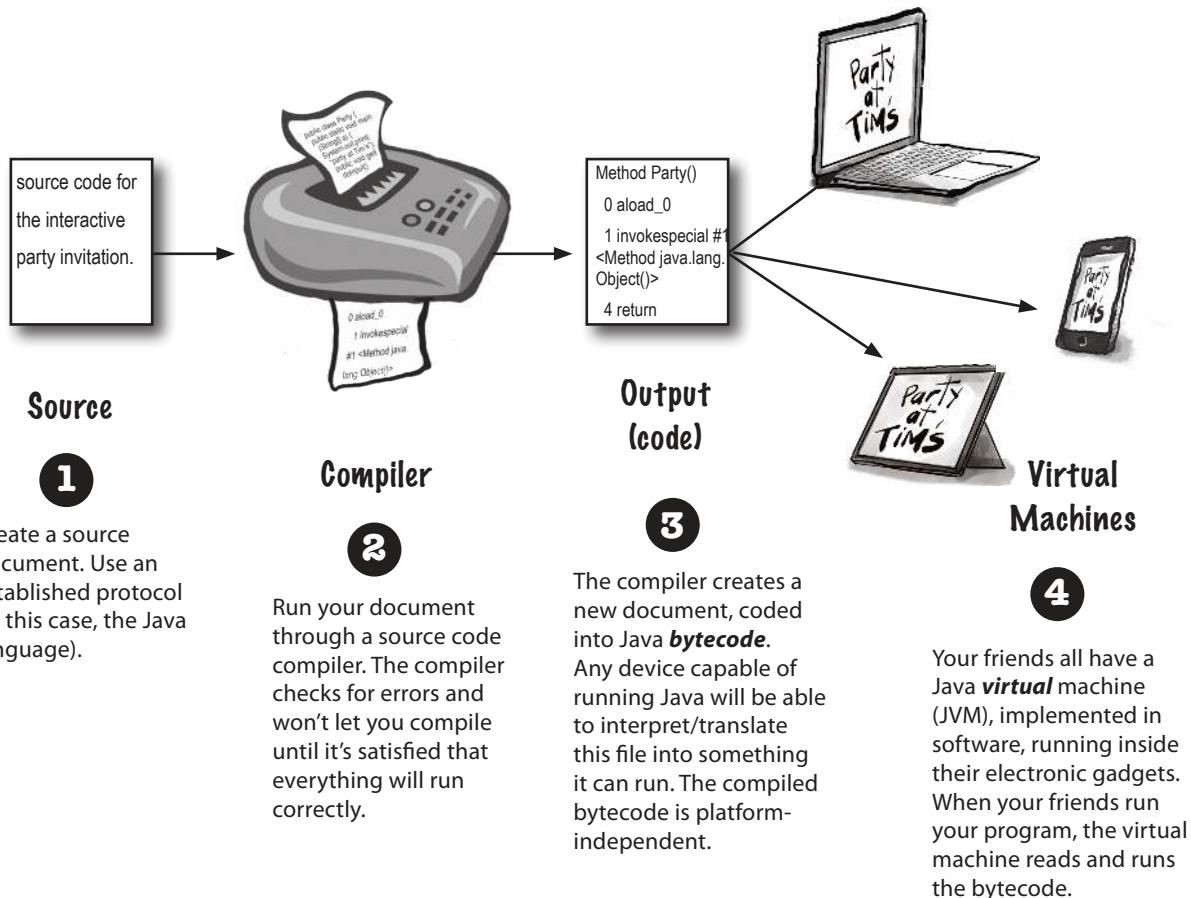


Come on, the water's great! We'll dive right in and write some code, then compile and run it. We're talking syntax, looping and branching, and a look at what makes Java so cool. You'll be coding in no time.

**Java takes you to new places.** From its humble release to the public as the (wimpy) version 1.02, Java seduced programmers with its friendly syntax, object-oriented features, memory management, and best of all—the promise of portability. The lure of **write-once/run-anywhere** is just too strong. A devoted following exploded, as programmers fought against bugs, limitations, and, oh yeah, the fact that it was dog slow. But that was ages ago. If you're just starting in Java, **you're lucky**. Some of us had to walk five miles in the snow, uphill both ways (barefoot), to get even the most trivial application to work. But *you, why, you* get to ride the **sleeker, faster, easier-to-read-and-write** Java of today.

# The way Java works

**The goal is to write one application (in this example, an interactive party invitation) and have it work on whatever device your friends have.**



# What you'll do in Java

You'll type a source code file, compile it using the **javac compiler**, and then run the compiled bytecode on a Java virtual machine.

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

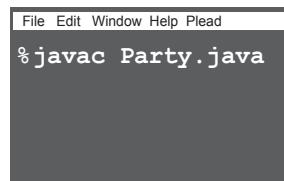
class Party {
    public void buildInvite() {
        Frame f = new Frame();
        Label l = new Label("Party at Tim's");
        Button b = new Button("You bet!");
        Button c = new Button("Shoot me!");
        Panel p = new Panel();
        p.add(l);
        p.add(b);
        p.add(c);
    } // more code here...
}
```

## Source

1

Type your source code.

Save as: ***Party.java***



## Compiler

2

Compile the ***Party.java*** file by running `javac` (the compiler application). If you don't have errors, you'll get a second document named ***Party.class***.

The compiler-generated *Party.class* file is made up of *bytecodes*.

```
Method Party()
0 aload_0
1 invokespecial #1 <Method
java.lang.Object()>
4 return

Method void buildInvite()
0 new #2 <Class java.awt.Frame>
3 dup
4 invokespecial #3 <Method
java.awt.Frame()>
```

## Output (code)

3

Compiled code: ***Party.class***



## Virtual Machines

4

Run the program by starting the Java Virtual Machine (JVM) with the ***Party.class*** file. The JVM translates the *bytecode* into something the underlying platform understands, and runs your program.

(Note: this is **NOT** meant to be a tutorial... you'll be writing real code in a moment, but for now, we just want you to get a feel for how it all fits together.)

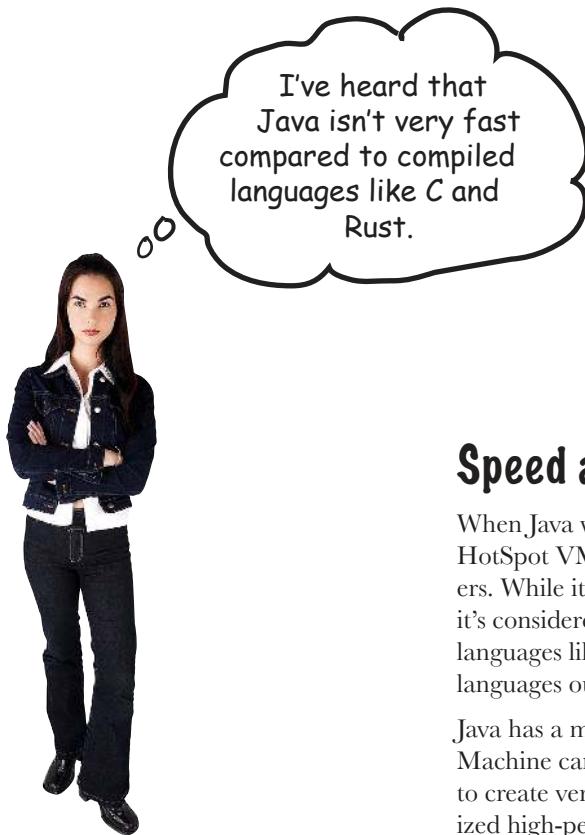
In other words, the code on this page isn't quite real; don't try to compile it.)

## A very brief history of Java

Java was initially released (some would say “escaped”), on January 23, 1996. It’s over 25 years old! In the first 25 years, Java as a language evolved, and the Java API grew enormously. The best estimate we have is that over 17 gazillion lines of Java code have been written in the last 25 years. As you spend time programming in Java, you will most certainly come across Java code that’s quite old, and some that’s much newer. Java is famous for its backward compatibility, so old code can run quite happily on new JVMs.

In this book we’ll generally start off by using older coding styles (remember, you’re likely to encounter such code in the “real world”), and then we’ll introduce newer-style code.

In a similar fashion, we will sometimes show you older classes in the Java API, and then show you newer alternatives.



## Speed and memory usage

When Java was first released, it was slow. But soon after, the HotSpot VM was created, as were other performance enhancers. While it’s true that Java isn’t the fastest language out there, it’s considered to be a very fast language—almost as fast as languages like C and Rust, and **much** faster than most other languages out there.

Java has a magic super-power—the JVM. The Java Virtual Machine can optimize your code *while it's running*, so it’s possible to create very fast applications without having to write specialized high-performance code.

But—full disclosure—compared to C and Rust, Java uses a lot of memory.

## Look how easy it is to write Java

```

int size = 27;
String name = "Fido";
Dog myDog = new Dog(name, size);
x = size - 5;
if (x < 15) myDog.bark(8);

while (x > 3) {
    myDog.play();
}

int[] numList = {2, 4, 6, 8};
System.out.print("Hello");
System.out.print("Dog: " + name);
String num = "8";
int z = Integer.parseInt(num);

try {
    readTheFile("myFile.txt");
}
catch (FileNotFoundException ex) {
    System.out.print("File not found.");
}

```



→ Answers on page 6.

Try to guess what each line of code is doing...  
(answers are on the next page).

declare an integer variable named 'size' and give it the value 27

if x (value of 22) is less than 15, tell the dog to bark 8 times

print out "Hello" ... probably at the command line

**Q:** The naming conventions for Java's versions are confusing. There was JDK 1.0, and 1.2, 1.3, 1.4, then a jump to J2SE 5.0, then it changed to Java 6, Java 7, and last time I checked, Java was up to Java 18. What's going on?

**A:** The version numbers have varied a lot over the last 25+ years! We can ignore the letters (J2SE/SE) since these are not really used now. The numbers are a little more involved.

Technically Java SE 5.0 was actually Java 1.5. Same for 6 (1.6), 7 (1.7), and 8 (1.8). In theory, Java is still on version

1.x because new versions are backward compatible, all the way back to 1.0.

However, it was a bit confusing having a version number that was different to the name everyone used, so the official version number from Java 9 onward is just the number, without the "1" prefix; i.e., Java 9 really is version 9, not version 1.9.

In this book we'll use the common convention of 1.0–1.4, then from 5 onward we'll drop the "1" prefix.

Also, since Java 9 was released in September 2017, there's been a release of Java every six months, each with a new "major" version number, so we moved very quickly from 9 to 18!



## Sharpen your pencil answers

### Look how easy it is to write Java

```

int size = 27;
String name = "Fido";
Dog myDog = new Dog(name, size);
x = size - 5;
if (x < 15) myDog.bark(8);

while (x > 3) {
    myDog.play();
}

int[] numList = {2, 4, 6, 8};
System.out.print("Hello");
System.out.print("Dog: " + name);
String num = "8";
int z = Integer.parseInt(num);

try {
    readTheFile("myFile.txt");
}

catch (FileNotFoundException ex) {
    System.out.print("File not found.");
}

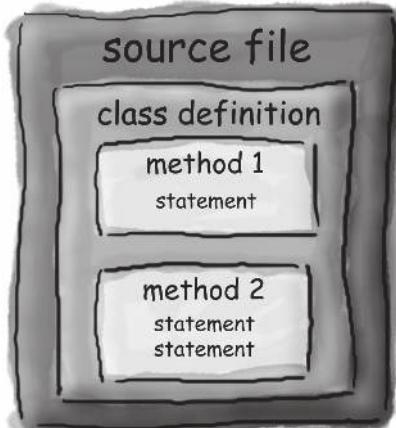
```

**Don't worry about whether you understand any of this yet!**

Everything here is explained in great detail in the book (most within the first 40 pages). If Java resembles a language you've used in the past, some of this will be simple. If not, don't worry about it. We'll get there...

declare an integer variable named 'size' and give it the value 27
declare a string of characters variable named 'name' and give it the value "Fido"
declare a new Dog variable 'myDog' and make the new Dog using 'name' and 'size'
subtract 5 from 27 (value of 'size') and assign it to a variable named 'x'
if x (value of 22) is less than 15, tell the dog to bark 8 times
keep looping as long as x is greater than 3...
tell the dog to play (whatever THAT means to a dog...)
this looks like the end of the loop -- everything in {} is done in the loop
declare a list of integers variable 'numList', and put 2,4,6,8 into the list
print out "Hello"... probably at the command line
print out "Dog: Fido" (the value of 'name' is "Fido") at the command line
declare a character string variable 'num' and give it the value of "8"
convert the string of characters "8" into an actual numeric value 8
try to do something... maybe the thing we're trying isn't guaranteed to work...
read a text file named "myFile.txt" (or at least TRY to read the file...)
must be the end of the "things to try", so I guess you could try many things...
this must be where you find out if the thing you tried didn't work...
if the thing we tried failed, print "File not found" out at the command line
looks like everything in the {} is what to do if the 'try' didn't work...

# Code structure in Java



**In a source file, put a class.**

**In a class, put methods.**

**In a method, put statements.**

## What goes in a source file?

A source code file (with the *.java* extension) typically holds one **class** definition. The class represents a *piece* of your program, although a very tiny application might need just a single class. The class must go within a pair of curly braces.

```
public class Dog {  
    class
```

## What goes in a class?

A class has one or more **methods**. In the Dog class, the **bark** method will hold instructions for how the Dog should bark. Your methods must be declared *inside* a class (in other words, within the curly braces of the class).

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

## What goes in a method?

Within the curly braces of a method, write your instructions for how that method should be performed. Method *code* is basically a set of statements, and for now you can think of a method kind of like a function or procedure.

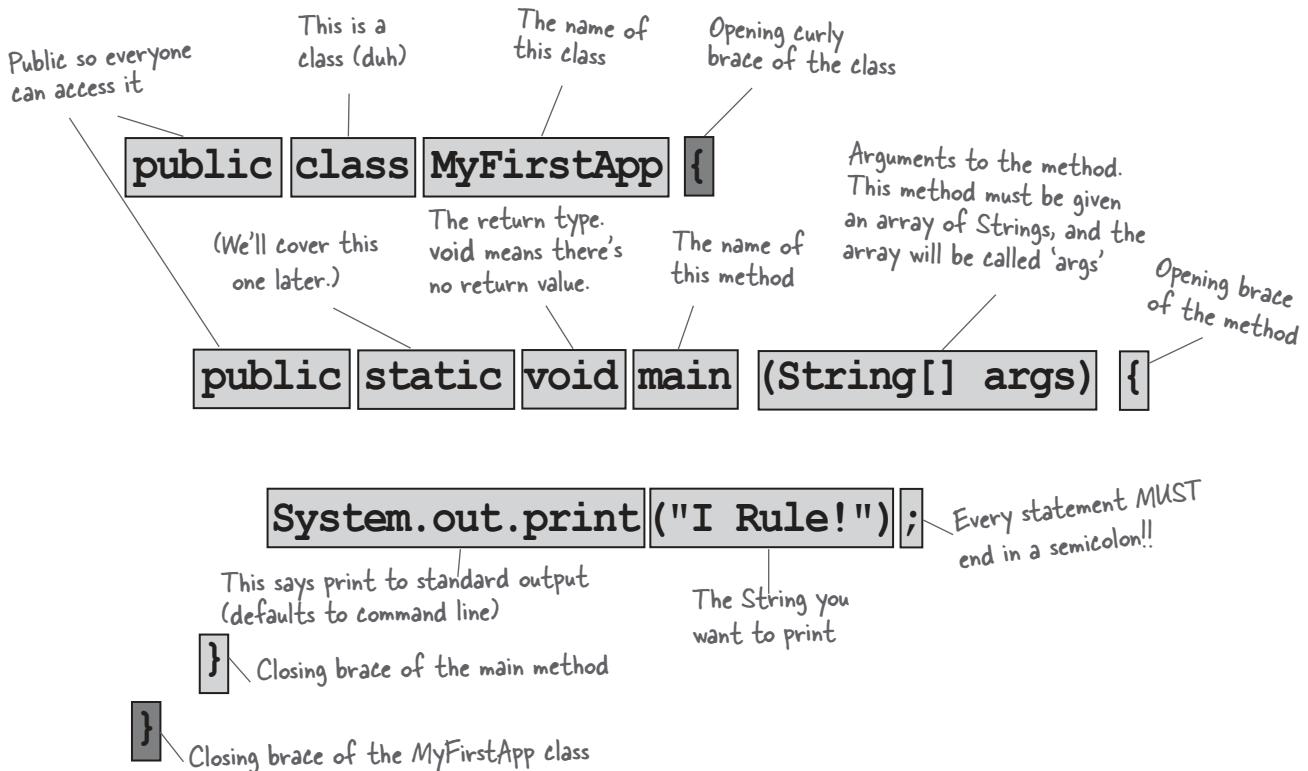
```
public class Dog {  
    void bark() {  
        statement1;  
        statement2;  
    }  
    statements
```

## Anatomy of a class

When the JVM starts running, it looks for the class you give it at the command line. Then it starts looking for a specially written method that looks exactly like:

```
public static void main (String[] args) {  
    // your code goes here  
}
```

Next, the JVM runs everything between the curly braces {} of your main method. Every Java application has to have at least one **class**, and at least one **main** method (not one main per *class*; just one main per *application*).



Don't worry about memorizing anything right now...  
this chapter is just to get you started.

# Writing a class with a `main()`

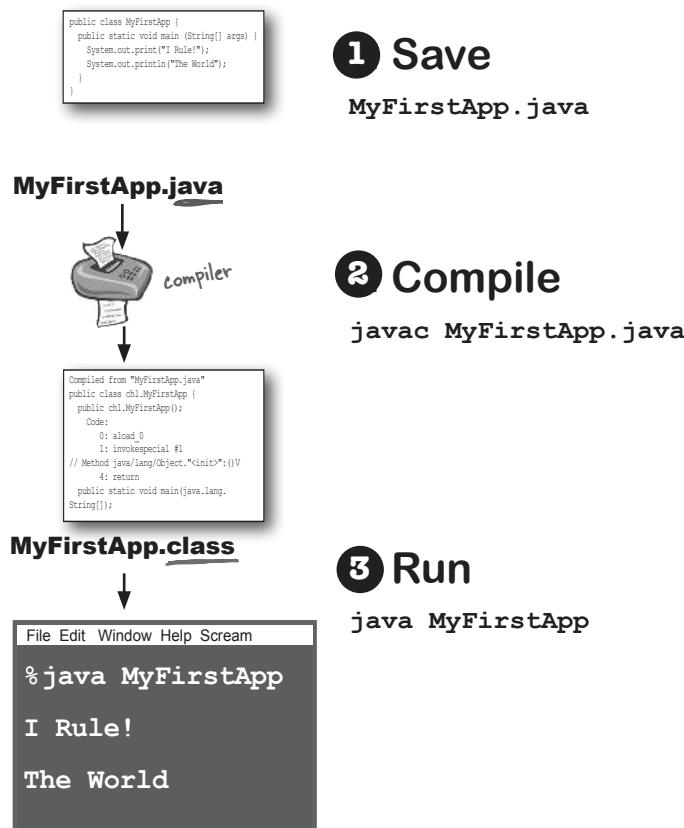
In Java, everything goes in a **class**. You'll type your source code file (with a *.java* extension), then compile it into a new class file (with a *.class* extension). When you run your program, you're really running a class.

Running a program means telling the Java Virtual Machine (JVM) to “Load the **MyFirstApp** class, then start executing its **main()** method. Keep running ‘til all the code in main is finished.”

In Chapter 2, *A Trip to Objectville*, we go deeper into the whole *class* thing, but for now, the only question you need to ask is, **how do I write Java code so that it will run?** And it all begins with **main()**.

The **main()** method is where your program starts running.

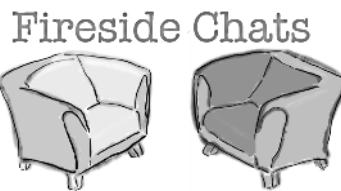
No matter how big your program is (in other words, no matter how many *classes* your program uses), there's got to be a **main()** method to get the ball rolling.



**1 Save**  
`MyFirstApp.java`

```
public class MyFirstApp {

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



### The Java Virtual Machine

What, are you kidding? *HELLO*. I am Java. I'm the one who actually makes a program run. The compiler just gives you a file. That's it. Just a file. You can print it out and use it for wallpaper, kindling, lining the bird cage, whatever, but the file doesn't do anything unless I'm there to run it.

And that's another thing, the compiler has no sense of humor. Then again, if you had to spend all day checking nitpicky little syntax violations...

I'm not saying you're, like, *completely* useless. But really, what is it that you do? Seriously. I have no idea. A programmer could just write bytecode by hand, and I'd take it. You might be out of a job soon, buddy.

(I rest my case on the humor thing.) But you still didn't answer my question, what *do* you actually do?

Tonight's Talk: **The compiler and the JVM battle over the question, “Who's more important?”**

### The Compiler

I don't appreciate that tone.

Excuse me, but without *me*, what exactly would you run? There's a *reason* Java was designed to use a bytecode compiler, for your information. If Java were a purely interpreted language, where—at runtime—the virtual machine had to translate straight-from-a-text-editor source code, a Java program would run at a ludicrously glacial pace.

Excuse me, but that's quite an ignorant (not to mention *arrogant*) perspective. While it *is* true that—*theoretically*—you can run any properly formatted bytecode even if it didn't come out of a Java compiler, in practice that's absurd. A programmer writing bytecode by hand is like painting pictures of your vacation instead of taking photos—sure, it's an art, but most people prefer to use their time differently. And I would appreciate it if you would *not* refer to me as “buddy.”

Remember that Java is a strongly typed language, and that means I can't allow variables to hold data of the wrong type. This is a crucial safety feature, and I'm able to stop the vast majority of violations before they ever get to you. And I also—

**The Java Virtual Machine**

But some still get through! I can throw ClassCastException s and sometimes I get people trying to put the wrong type of thing in an array that was declared to hold something else, and—

OK. Sure. But what about *security*? Look at all the security stuff I do, and you're like, what, checking for *semicolons*? Oooohhh big security risk! Thank goodness for you!

Whatever. I have to do that same stuff *too*, though, just to make sure nobody snuck in after you and changed the bytecode before running it.

Oh, you can count on it. *Buddy*.

**The Compiler**

Excuse me, but I wasn't done. And yes, there *are* some datatype exceptions that can emerge at runtime, but some of those have to be allowed to support one of Java's other important features—dynamic binding. At runtime, a Java program can include new objects that weren't even *known* to the original programmer, so I have to allow a certain amount of flexibility. But my job is to stop anything that would never—*could* never—succeed at runtime. Usually I can tell when something won't work, for example, if a programmer accidentally tried to use a Button object as a Socket connection, I would detect that and thus protect them from causing harm at runtime.

Excuse me, but I am the first line of defense, as they say. The datatype violations I previously described could wreak havoc in a program if they were allowed to manifest. I am also the one who prevents access violations, such as code trying to invoke a private method, or change a method that—for security reasons—must never be changed. I stop people from touching code they're not meant to see, including code trying to access another class' critical data. It would take hours, perhaps days even, to describe the significance of my work.

Of course, but as I indicated previously, if I didn't prevent what amounts to perhaps 99% of the potential problems, you would grind to a halt. And it looks like we're out of time, so we'll have to revisit this in a later chat.

## What can you say in the main method?

Once you're inside main (or *any* method), the fun begins. You can say all the normal things that you say in most programming languages to **make the computer do something**.

Your code can tell the JVM to:

### ➊ do something

**Statements:** declarations, assignments, method calls, etc.

```
int x = 3;
String name = "Dirk";
x = x * 17;
System.out.print("x is " + x);
double d = Math.random();
// this is a comment
```

### ➋ do something again and again

**Loops:** *for* and *while*

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

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

### ➌ do something under this condition

**Branching:** *if/else* tests

```
if (x == 10) {
    System.out.print("x must be 10");
} else {
    System.out.print("x isn't 10");
}

if ((x < 3) && (name.equals("Dirk"))) {
    System.out.println("Gently");
}

System.out.print("this line runs no matter what");
```



## Syntax Fun

★ Each statement must end in a semicolon.

x = x + 1;

★ A single-line comment begins with two forward slashes.

x = 22;

// this line disturbs me

★ Most white space doesn't matter.

x = 3 ;

★ Variables are declared with a **name** and a **type** (you'll learn about all the Java **types** in Chapter 3).

```
int weight;
// type: int, name: weight
```

★ Classes and methods must be defined within a pair of curly braces.

```
public void go() {
    // amazing code here
}
```



## Looping and looping and...

Java has a lot of looping constructs: `while`, `do-while`, and `for`, being the oldest. You'll get the full loop scoop later in the book, but not right now. Let's start with `while`.

The syntax (not to mention logic) is so simple you're probably asleep already. As long as some condition is true, you do everything inside the loop *block*. The loop block is bounded by a pair of curly braces, so whatever you want to repeat needs to be inside that block.

The key to a loop is the *conditional test*. In Java, a conditional test is an expression that results in a *boolean* value—in other words, something that is either **true** or **false**.

If you say something like, “While *iceCreamInTheTub* is **true**, keep scooping,” you have a clear boolean test. There either *is* ice cream in the tub or there *isn't*. But if you were to say, “While *Bob* keep scooping,” you don't have a real test. To make that work, you'd have to change it to something like, “While *Bob* is *snoring...*” or “While *Bob* is *not* wearing plaid...”

```
while (moreBalls == true) {
    keepJuggling();
}
```

### Simple boolean tests

You can do a simple boolean test by checking the value of a variable, using a comparison operator like:

**<** (less than)

**>** (greater than)

**==** (equality) (yes, that's *two* equals signs)

Notice the difference between the *assignment* operator (a *single* equals sign) and the *equals* operator (*two* equals signs). Lots of programmers accidentally type **=** when they *want* **==**. (But not you.)

```
int x = 4; // assign 4 to x
while (x > 3) {
    // loop code will run because
    // x is greater than 3
    x = x - 1; // or we'd loop forever
}
int z = 27; //
while (z == 17) {
    // loop code will not run because
    // z is not equal to 17
}
```

## there are no Dumb Questions

**Q:** Why does everything have to be in a class?

**A:** Java is an object-oriented (OO) language. It's not like the old days when you had steam-driven compilers and wrote one monolithic source file with a pile of procedures. In Chapter 2, *A Trip to Objectville*, you'll learn that a class is a blueprint for an object, and that nearly everything in Java is an object.

**Q:** Do I have to put a main in every class I write?

**A:** Nope. A Java program might use dozens of classes (even hundreds), but you might only have *one* with a main method—the one that starts the program running.

**Q:** In my other language I can do a boolean test on an integer. In Java, can I say something like:

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

**A:** No. A *boolean* and an *integer* are not compatible types in Java. Since the result of a conditional test *must* be a boolean, the only variable you can directly test (without using a comparison operator) is a **boolean**. For example, you can say:

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

## Example of a while loop

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

*This is the output*

Before the Loop  
In the loop  
Value of x is 1  
In the loop  
Value of x is 2  
In the loop  
Value of x is 3  
This is after the loop

### BULLET POINTS

- Statements end in a semicolon ;
- Code blocks are defined by a pair of curly braces {}
- Declare an *int* variable with a name and a type: int x;
- The **assignment** operator is *one* equals sign =
- The **equals** operator uses *two* equals signs ==
- A *while* loop runs everything within its block (defined by curly braces) as long as the *conditional test* is **true**.
- If the conditional test is **false**, the *while* loop code block won't run, and execution will move down to the code immediately *after* the loop block.
- Put a boolean test inside parentheses:  
`while (x == 4) { }`

# Conditional branching

In Java, an *if* test is basically the same as the boolean test in a *while* loop—except instead of saying, “**while** there’s still chocolate,” you’ll say, “**if** there’s still chocolate...”

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

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

*Code output*

The preceding code executes the line that prints “x must be 3” only if the condition (*x* is equal to 3) is true. Regardless of whether it’s true, though, the line that prints “This runs no matter what” will run. So depending on the value of *x*, either one statement or two will print out.

But we can add an *else* to the condition so that we can say something like, “*If* there’s still chocolate, keep coding, *else* (otherwise) get more chocolate, and then continue on...”

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

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

*New output*

## System.out.print vs. System.out.println

If you’ve been paying attention (of course you have), then you’ve noticed us switching between **print** and **println**.

### Did you spot the difference?

**System.out.println** inserts a newline (think of **println** as **printnewline**), while **System.out.print** keeps printing to the *same* line. If you want each thing you print out to be on its own line, use **println**. If you want everything to stick together on one line, use **print**.



## Sharpen your pencil

### Given the output:

```
% java DooBee
DooBeeDooBeeDo
```

### Fill in the missing code:

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

→ Answers on page 25.

# Coding a serious business application

Let's put all your new Java skills to good use with something practical. We need a class with a `main()`, an `int` and a `String` variable, a `while` loop, and an `if` test. A little more polish, and you'll be building that business back-end in no time. But *before* you look at the code on this page, think for a moment about how *you* would code that classic children's favorite, "10 green bottles."



```
public class BottleSong {  
    public static void main(String[] args) {  
        int bottlesNum = 10;  
        String word = "bottles";  
  
        while (bottlesNum > 0) {  
  
            if (bottlesNum == 1) {  
                word = "bottle"; // singular, as in ONE bottle.  
            }  
  
            System.out.println(bottlesNum + " green " + word + ", hanging on the wall");  
            System.out.println(bottlesNum + " green " + word + ", hanging on the wall");  
            System.out.println("And if one green bottle should accidentally fall,");  
            bottlesNum = bottlesNum - 1;  
  
            if (bottlesNum > 0) {  
                System.out.println("There'll be " + bottlesNum +  
                    " green " + word + ", hanging on the wall");  
            } else {  
                System.out.println("There'll be no green bottles, hanging on the wall");  
            } // end else  
        } // end while loop  
    } // end main method  
} // end class
```

There's still one little flaw in our code. It compiles and runs, but the output isn't 100% perfect. See if you can spot the flaw and fix it.

there are no  
**Dumb Questions**

**Q:** Didn't this use to be "99 Bottles of Beer"?

**A:** Yes, but Trisha wanted us to use the UK version of the song. If you'd prefer the 99 bottles version, take that as a fun exercise.

## Monday morning at Bob's Java-enabled house

Bob's alarm clock rings at 8:30 Monday morning, just like every other weekday. But Bob had a wild weekend and reaches for the SNOOZE button. And that's when the action starts, and the Java-enabled appliances come to life...



First, the alarm clock sends a message to the coffee maker  
“Hey, the geek’s sleeping in again, delay the coffee 12 minutes.”



The coffee maker sends a message to the Motorola™ toaster,  
“Hold the toast, Bob’s snoozing.”

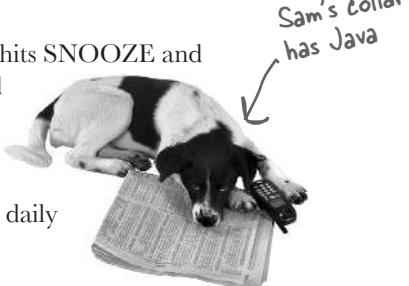
The alarm clock then sends a message to Bob’s  
Android, “Call Bob’s 9 o’clock and tell him we’re  
running a little late.”



Finally, the alarm clock sends a message to Sam’s  
(Sam is the dog) wireless collar, with the too-familiar signal that means, “Get the paper, but  
don’t expect a walk.”



A few minutes later, the alarm goes off again. And *again* Bob hits SNOOZE and the appliances start chattering. Finally, the alarm rings a third time. But just as Bob reaches for the snooze button, the clock sends the “jump and bark” signal to Sam’s collar. Shocked to full consciousness, Bob rises, grateful that his Java skills, and spontaneous internet shopping purchases, have enhanced the daily routines of his life.



*His toast is toasted.*



*His coffee steams.*

*His paper awaits.*

Just another wonderful morning in ***The Java-Enabled House.***

Could this story be true? Mostly, yes! There *are* versions of Java running in devices including cell phones (*especially* cell phones), ATMs, credit cards, home security systems, parking meters, game consoles and more—but you might not find a Java dog collar...yet.

Java has multiple ways to use just a tiny part of the Java platform to run on smaller devices (depending upon the version of Java you’re using). It’s very popular for IoT (Internet of Things) development. And, of course, lots of Android development is done with Java and JVM languages.



Try my new phrase-o-matic and you'll be a slick talker just like the boss or those hotshots in marketing.

OK, so the bottle song wasn't *really* a serious business application. Still need something practical to show the boss? Check out the Phrase-O-Matic code.

Note: when you type this into an editor, let the code do its own word/line-wrapping! Never hit the return key when you're typing a String (a thing between "quotes") or it won't compile. So the hyphens you see on this page are real, and you can type them, but don't hit the return key until AFTER you've closed a String.

```
public class PhraseOMatic {  
    public static void main (String[] args) {  
  
        1 // make three sets of words to choose from. Add your own!  
        String[] wordListOne = {"agnostic", "opinionated",  
        "voice activated", "haptically driven", "extensible",  
        "reactive", "agent based", "functional", "AI enabled",  
        "strongly typed"};  
  
        String[] wordListTwo = {"loosely coupled", "six sigma",  
        "asynchronous", "event driven", "pub-sub", "IoT", "cloud  
        native", "service oriented", "containerized", "serverless",  
        "microservices", "distributed ledger"};  
  
        String[] wordListThree = {"framework", "library",  
        "DSL", "REST API", "repository", "pipeline", "service  
        mesh", "architecture", "perspective", "design",  
        "orientation"};  
  
        2 // find out how many words are in each list  
        int oneLength = wordListOne.length;  
        int twoLength = wordListTwo.length;  
        int threeLength = wordListThree.length;  
  
        3 // generate three random numbers  
        java.util.Random randomGenerator = new java.util.Random();  
        int rand1 = randomGenerator.nextInt(oneLength);  
        int rand2 = randomGenerator.nextInt(twoLength);  
        int rand3 = randomGenerator.nextInt(threeLength);  
  
        4 // now build a phrase  
        String phrase = wordListOne[rand1] + " " +  
        wordListTwo[rand2] + " " + wordListThree[rand3];  
  
        5 // print out the phrase  
        System.out.println("What we need is a " + phrase);  
    }  
}
```

# Phrase-O-Matic

## How it works

In a nutshell, the program makes three lists of words, then randomly picks one word from each of the three lists, and prints out the result. Don't worry if you don't understand *exactly* what's happening in each line. For goodness sake, you've got the whole book ahead of you, so relax. This is just a quick look from a 30,000-foot outside-the-box targeted leveraged paradigm.

1. The first step is to create three String arrays—the containers that will hold all the words. Declaring and creating an array is easy; here's a small one:

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

Each word is in quotes (as all good Strings must be) and separated by commas.

2. For each of the three lists (arrays), the goal is to pick a random word, so we have to know how many words are in each list. If there are 14 words in a list, then we need a random number between 0 and 13 (Java arrays are zero-based, so the first word is at position 0, the second word position 1, and the last word is position 13 in a 14-element array). Quite handily, a Java array is more than happy to tell you its length. You just have to ask. In the `pets` array, we'd say:

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

and `x` would now hold the value 3.

3. We need three random numbers. Java ships out of the box with several ways to generate random numbers, including `java.util.Random` (we will see later why this class name is prefixed with `java.util`). The `nextInt()` method returns a random number between 0 and some-number-we-give-it, *not including* the number that we give it. So we'll give it the number of elements (the array length) in the list we're using. Then we assign each result to a new variable. We could just as easily have asked for a random number between 0 and 5, not including 5:

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

4. Now we get to build the phrase, by picking a word from each of the three lists and smooshing them together (also inserting spaces between words). We use the “`+`” operator, which *concatenates* (we prefer the more technical *smooshes*) the String objects together. To get an element from an array, you give the array the index number (position) of the thing you want by using:

```
String s = pets[0]; // s is now the String "Fido"
s = s + " " + "is a dog"; // s is now "Fido is a dog"
```

5. Finally, we print the phrase to the command line and...voilà! *We're in marketing.*

**what we need  
here is a...**

**extensible microser-  
vices pipeline**

**opinionated loosely  
coupled REST API**

**agent-based  
microservices library**

**AI-enabled service  
oriented orientation**

**agnostic pub-sub  
DSL**

**functional IoT  
perspective**

## exercise: Code Magnets



# Code Magnets

A working Java program is all scrambled up on the fridge. Can you rearrange the code snippets to make a working Java program that produces the output listed below? Some of the curly braces fell on the floor and they were too small to pick up, so feel free to add as many of those as you need!

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

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

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

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

```
    int x = 3;
```

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

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

## Output:

```
File Edit Window Help Sleep  
% java Shuffle1  
a-b c-d
```

→ Answers on page 25.



## BE the Compiler



Each of the Java files on this page represents a complete source file. Your job is to play compiler and determine whether each of these files will compile. If they won't compile, how would you fix them?

→ Answers on page 25.

B

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

A

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

C

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



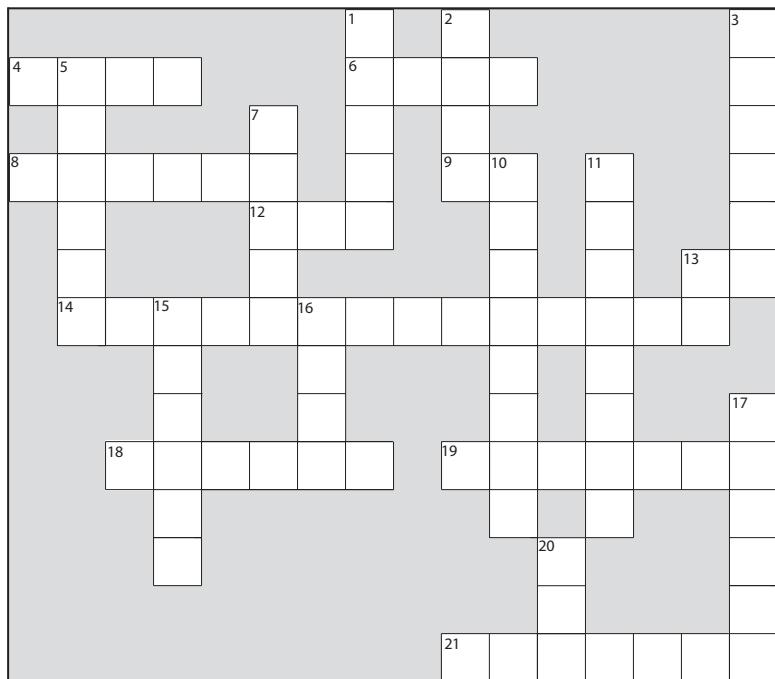
# JavaCrōSS

Let's give your right brain something to do.

It's your standard crossword, but almost all of the solution words are from Chapter 1. Just to keep you awake, we also threw in a few (non-Java) words from the high-tech world.

## Across

4. Command line invoker
6. Back again?
8. Can't go both ways
9. Acronym for your laptop's power
12. Number variable type
13. Acronym for a chip
14. Say something
18. Quite a crew of characters
19. Announce a new class or method
21. What's a prompt good for?



## Down

1. Not an integer (or \_\_\_\_\_ your boat)
2. Come back empty-handed
3. Open house
5. 'Things' holders
7. Until attitudes improve
10. Source code consumer
11. Can't pin it down
13. Department for programmers and operations
15. Shocking modifier
16. Just gotta have one
17. How to get things done
20. Bytecode consumer

→ Answers on page 26.



A short Java program is listed below. One block of the program is missing. Your challenge is to **match the candidate block of code** (on the left) **with the output** that you'd see if the block were inserted. Not all the lines of output will be used, and some of the lines of output might be used more than once. Draw lines connecting the candidate blocks of code with their matching command-line output.

```
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;
        }
    }
}
```

*Candidate code goes here*

**Candidates:**

**Possible output:**

*Match each candidate with one of the possible outputs*

y = x - y;

22 46

y = y + x;

11 34 59

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

02 14 26 38

02 14 36 48

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

00 11 21 32 42

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

11 21 32 42 53

00 11 23 36 410

02 14 25 36 47

→ Answers on page 26.

## puzzle: Pool Puzzle



### Pool Puzzle



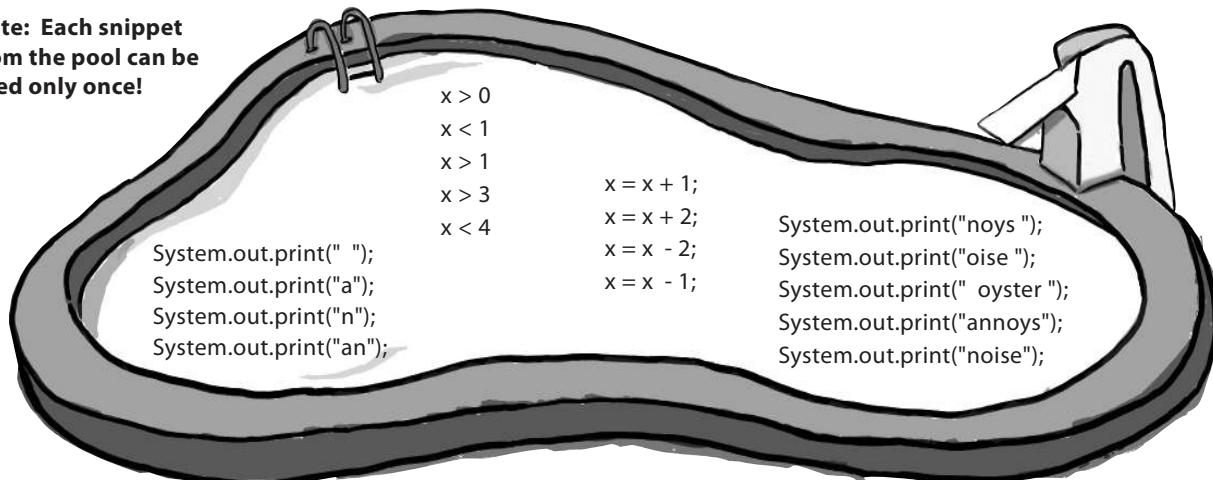
Your **job** is to take code snippets from the pool and place them into the blank lines in the code. You may **not** use the same snippet more than once, and you won't need to use all the snippets. Your **goal** is to make a class that will compile and run and produce the output listed. Don't be fooled—this one's harder than it looks.

→ Answers on page 26.

#### Output

```
File Edit Window Help Cheat
%java PoolPuzzleOne
a noise
annoys
an oyster
```

Note: Each snippet from the pool can be used only once!



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

        while ( _____ ) {

            if ( x < 1 ) {
                _____
            }

            if ( _____ ) {
                _____
            }

            if ( x == 1 ) {
                _____
            }

            if ( _____ ) {
                _____
            }

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



## Exercise Solutions

### Sharpen your pencil (from page 14)

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

### Code Magnets (from page 20)

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

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

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

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

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

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

```
File Edit Window Help Poet
% java Shuffle1
a-b c-d
```

**BE the Compiler**  
(from page 21)

**A** **dive in: a quick dip**

```
class Exercise1a {
    public static void main(String [] args) {
        int x = 1;
        while ( x < 10 ) {
            x = x + 1; ← Add this line to prevent
            if ( x > 3 ) { it running forever...
                System.out.println("big x");
            }
        }
    }
}
```

This will compile and run (no output), but without a line added to the program, it would run forever in an infinite while loop!

---

**B** **class Exercise1b { ← Needs a class declaration**

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

This file won't compile without a class declaration, and don't forget the matching curly brace!

---

**C** **class Exercise1c { ↗ Needs a "main"**

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

The while loop code must be inside a method. It can't just be hanging out inside the class.

## puzzle answers



## Pool Puzzle (from page 24)

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

        while ( x<4 ) {

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

            System.out.print("\n");

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

        x=x+1;
    }
}
```

```
File Edit Window Help Cheat
%java PoolPuzzleOne
a noise
annoys
an oyster
```

## Mixed Messages

```
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;
        }
    }
}
```

### Candidates:

`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;
}
```

### Possible output:

`22 46`

`11 34 59`

`02 14 26 38`

`02 14 36 48`

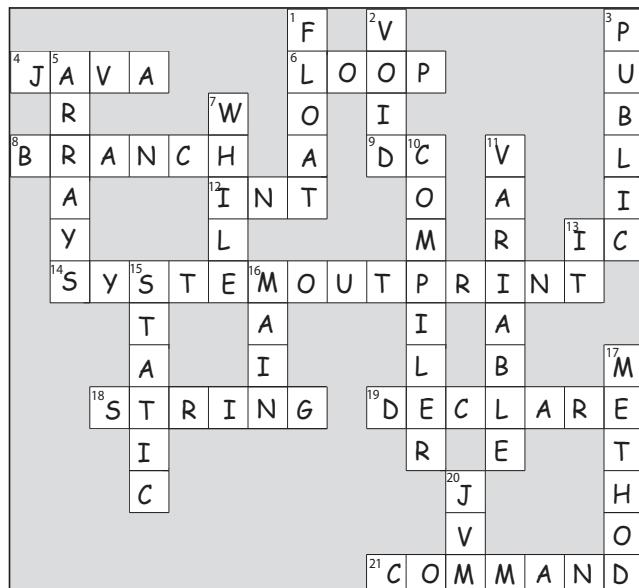
`00 11 21 32 42`

`11 21 32 42 53`

`00 11 23 36 410`

`02 14 25 36 47`

## JavaCrōSS (from page 22)



## 2 classes and objects

# A Trip to Objectville



**I was told there would be objects.** In Chapter 1, we put all of our code in the `main()` method. That's not exactly object-oriented. In fact, that's not object-oriented at all. Well, we did use a few objects, like the `String` arrays for the `Phrase-O-Matic`, but we didn't actually develop any of our own object types. So now we've got to leave that procedural world behind, get the heck out of `main()`, and start making some objects of our own. We'll look at what makes object-oriented (OO) development in Java so much fun. We'll look at the difference between a class and an object. We'll look at how objects can give you a better life (at least the programming part of your life. Not much we can do about your fashion sense). Warning: once you get to Objectville, you might never go back. Send us a postcard.

# Chair Wars

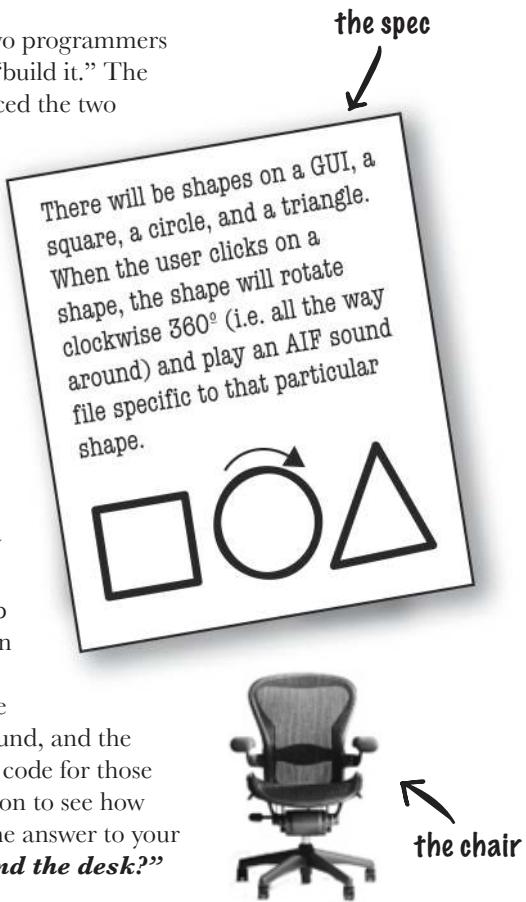
## (or How Objects Can Change Your Life)

Once upon a time in a software shop, two programmers were given the same spec and told to “build it.” The Really Annoying Project Manager forced the two coders to compete, by promising that

whoever delivers first gets a cool Aeron™ chair and adjustable height standing desk like all the Silicon Valley techies have. Laura, the procedural programmer, and Brad, the OO developer, both knew this would be a piece of cake.

Laura, sitting at her (non-adjustable) desk, thought to herself, “What are the things this program has to *do*? What **procedures** do we need?” And she answered herself, “**rotate** and **playSound**.” So off she went to build the procedures. After all, what is a program if not a pile of procedures?

Brad, meanwhile, kicked back at the coffee shop and thought to himself, “What are the **things** in this program...who are the key *players*?” He first thought of **The Shapes**. Of course, there were other things he thought of like the User, the Sound, and the Clicking Event. But he already had a library of code for those pieces, so he focused on building Shapes. Read on to see how Brad and Laura built their programs, and for the answer to your burning question, “**So, who got the Aeron and the desk?**”



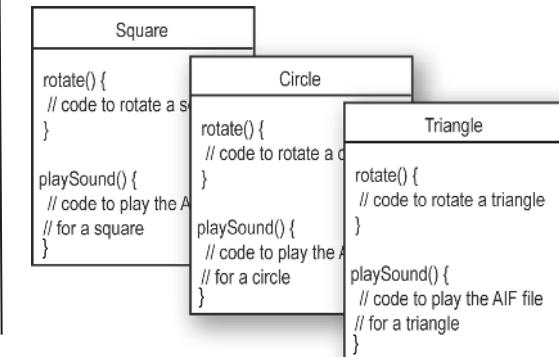
### At Laura's desk

As she had done a gazillion times before, Laura set about writing her **Important Procedures**. She wrote **rotate** and **playSound** in no time.

```
rotate(shapeNum) {
    // make the shape rotate 360°
}
playSound(shapeNum) {
    // use shapeNum to lookup which
    // AIF sound to play, and play it
}
```

### At Brad's laptop at the cafe

Brad wrote a **class** for each of the three shapes.

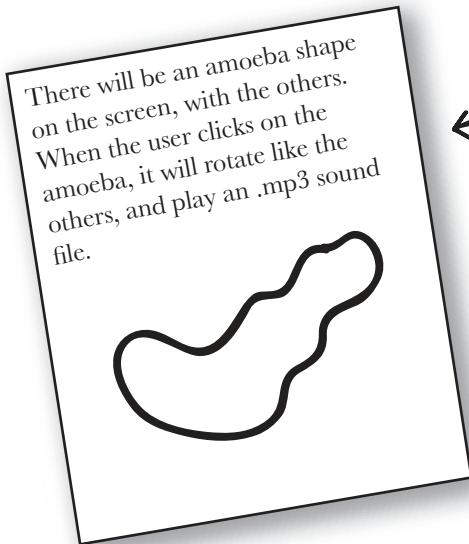


**Laura thought she'd nailed it. She could almost feel the rolled steel of the Aeron beneath her...**

### But wait! There's been a spec change.

"OK, *technically* you were first, Laura," said the Manager, "but we have to add just one tiny thing to the program. It'll be no problem for crack programmers like you two."

*"If I had a dime for every time I've heard that one,"* thought Laura, knowing that spec-change-no-problem was a fantasy. *"And yet Brad looks strangely serene. What's up with that?"* Still, Laura held tight to her core belief that the OO way, while cute, was just slow. And that if you wanted to change her mind, you'd have to pry it from her cold, dead, carpal-tunnelled hands.



← what got added to the spec

### Back at Laura's desk

The rotate procedure would still work; the code used a lookup table to match a shapeNum to an actual shape graphic. But **playSound would have to change.**

```
playSound(shapeNum) {
    // if the shape is not an amoeba,
    // use shapeNum to lookup which
    // AIF sound to play, and play it
    // else
    // play amoeba .mp3 sound
}
```

It turned out not to be such a big deal, but **it still made her queasy to touch previously tested code.** Of all people, *she* should know that no matter what the project manager says, **the spec always changes.**

### At Brad's laptop at the beach

Brad smiled, sipped his fruit frappe, and *wrote one new class*. Sometimes the thing he loved most about OO was that he didn't have to touch code he'd already tested and delivered. "Flexibility, extensibility, ..." he mused, reflecting on the benefits of OO.

Amoeba
<pre>rotate() {     // code to rotate an amoeba }  playSound() {     // code to play the new     // .mp3 file for an amoeba }</pre>

## Laura delivered just moments ahead of Brad

(Hah! So much for that foofy OO nonsense.) But the smirk on Laura's face melted when the Really Annoying Project Manager said (with that tone of disappointment), "Oh, no, *that's* not how the amoeba is supposed to rotate..."

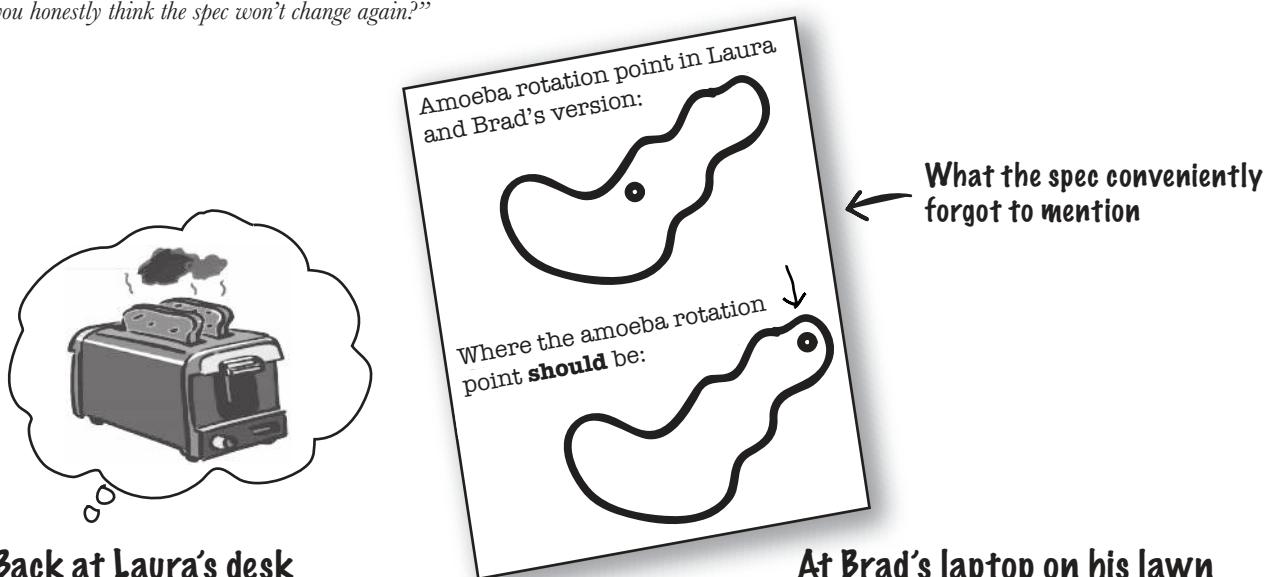
Turns out, both programmers had written their rotate code like this:

**1. determine the rectangle that surrounds the shape.**

**2. calculate the center of that rectangle, and rotate the shape around that point.**

But the amoeba shape was supposed to rotate around a point on one *end*, like a clock hand.

"I'm toast," thought Laura, visualizing charred Wonderbread™. "Although, hmhhh. I could just add another if/else to the rotate procedure and then just hard-code the rotation point code for the amoeba. That probably won't break anything." But the little voice at the back of her head said, "*Big Mistake. Do you honestly think the spec won't change again?*"



## Back at Laura's desk

She figured she better add rotation point arguments to the rotate procedure. ***A lot of code was affected.*** Testing, recompiling, the whole nine yards all over again. Things that used to work, didn't.

```
rotate(shapeNum, xPt, yPt) {
    // if the shape is not an amoeba,
    // calculate the center point
    // based on a rectangle,
    // then rotate
    // else
    // use the xPt and yPt as
    // the rotation point offset
    // and then rotate
}
```

## At Brad's laptop on his lawn chair at the Telluride Bluegrass Festival

Without missing a beat, Brad modified the rotate **method**, but only in the Amoeba class. ***He never touched the tested, working, compiled code*** for the other parts of the program. To give the Amoeba a rotation point, he added an **attribute** that all Amoebas would have. He modified, tested, and delivered (via free festival WiFi) the revised program during a single Bela Fleck set.

Amoeba
int xPoint
int yPoint
rotate() {
// code to rotate an amoeba
// using amoeba's x and y
}
playSound() {
// code to play the new
// .mp3 file for an amoeba
}

## So, Brad the OO guy got the chair and desk, right?

**Not so fast.** Laura found a flaw in Brad's approach. And, since she was sure that if she got the chair and desk, she'd also be next in line for a promotion, she had to turn this thing around.

**LAURA:** You've got duplicated code! The rotate procedure is in all four Shape things.

**BRAD:** It's a **method**, not a *procedure*. And they're **classes**, not *things*.

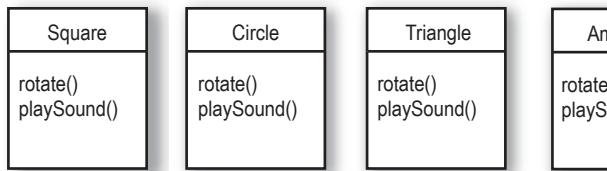
**LAURA:** Whatever. It's a stupid design. You have to maintain *four* different rotate "methods." How can that ever be good?

**BRAD:** Oh, I guess you didn't see the final design. Let me show you how OO **inheritance** works, Laura.



What Laura really wanted ↗

(figured the chair was a step closer to that promotion and the big bucks)



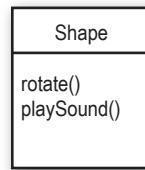
1

I looked at what all four classes have in common.



2

They're Shapes, and they all rotate and playSound. So I abstracted out the common features and put them into a new class called Shape.

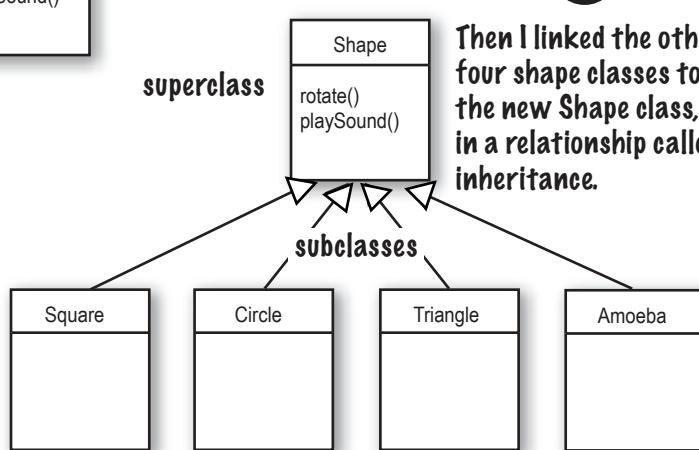


3

Then I linked the other four shape classes to the new Shape class, in a relationship called inheritance.

You can read this as, "Square inherits from Shape," "Circle inherits from Shape," and so on. I removed rotate() and playSound() from the other shapes, so now there's only one copy to maintain.

The Shape class is called the **superclass** of the other four classes. The other four are the **subclasses** of Shape. The subclasses inherit the methods of the superclass. In other words, *if the Shape class has the functionality, then the subclasses automatically get that same functionality*.



## What about the Amoeba rotate()?

**LAURA:** Wasn't that the whole problem here—that the amoeba shape had a completely different rotate and playSound procedure?

**BRAD: Method.**

**LAURA:** Whatever. How can Amoeba do something different if it “inherits” its functionality from the Shape class?

**BRAD:** That's the last step. The Amoeba class **overrides** the methods of the Shape class. Then at runtime, the JVM knows exactly which rotate() method to run when someone tells the Amoeba to rotate.

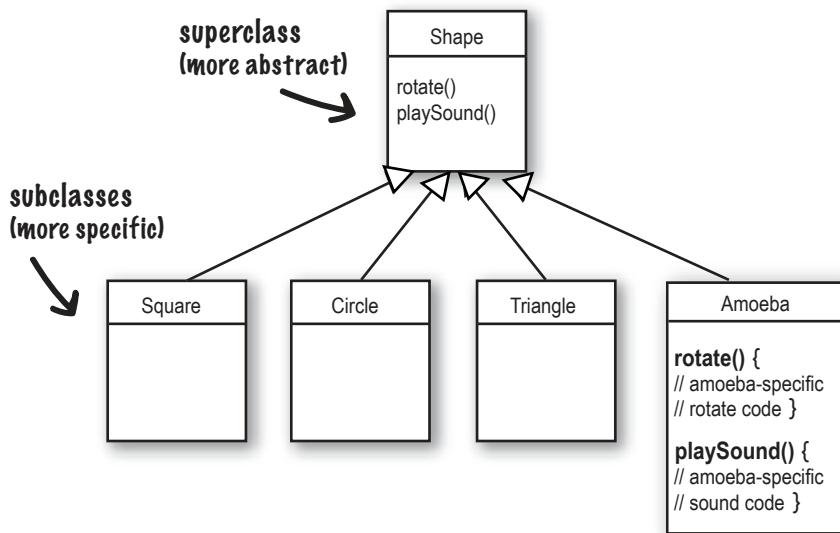


4

I made the Amoeba class override the rotate() and playSound() methods of the superclass Shape.

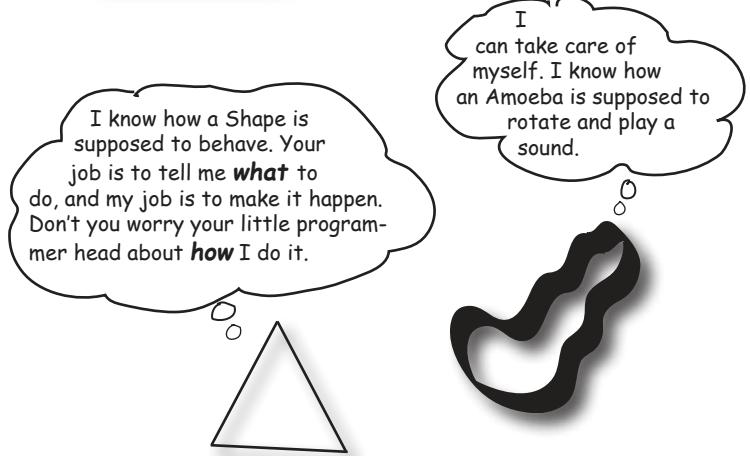
Overriding just means that a subclass redefines one of its inherited methods when it needs to change or extend the behavior of that method.

Overriding methods



**LAURA:** How do you “tell” an Amoeba to do something? Don’t you have to call the procedure, sorry—*method*, and then tell it *which* thing to rotate?

**BRAD:** That’s the really cool thing about OO. When it’s time for, say, the triangle to rotate, the program code invokes (calls) the rotate() method on the triangle object. The rest of the program really doesn’t know or care *how* the triangle does it. And when you need to add something new to the program, you just write a new class for the new object type, so the **new objects will have their own behavior**.



## The suspense is killing me. Who got the chair and desk?



### Amy from the second floor.

(Unbeknownst to all, the Project Manager had given the spec to *three* programmers. Amy completed the project faster since she got on with OO programming without arguing with her co-workers).

## What do you like about OO?

"It helps me design in a more natural way. Things have a way of evolving."

-Joy, 27, software architect

"Not messing around with code I've already tested, just to add a new feature."

-Brad, 32, programmer

"I like that the data and the methods that operate on that data are together in one class."

-Jess, 22, foosball champion

"Reusing code in other applications. When I write a new class, I can make it flexible enough to be used in something new, later."

-Chris, 39, project manager

"I can't believe Chris, who hasn't written a line of code in 5 years, just said that."

-Daryl, 44, works for Chris

"Besides the chair?"

-Amy, 34, programmer



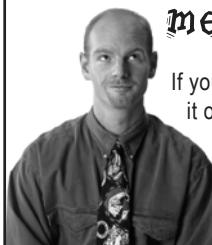
### Time to pump some neurons.

You just read a story about a procedural programmer going head-to-head with an OO programmer. You got a quick overview of some key OO concepts including classes, methods, and attributes. We'll spend the rest of the chapter looking at classes and objects (we'll return to inheritance and overriding in later chapters).

Based on what you've seen so far (and what you may know from a previous OO language you've worked with), take a moment to think about these questions:

What are the fundamental things you need to think about when you design a Java class? What are the questions you need to ask yourself? If you could design a checklist to use when you're designing a class, what would be on the checklist?

### metacognitive tip



If you're stuck on an exercise, try talking about it out loud. Speaking (and hearing) activates a different part of your brain. Although it works best if you have another person to discuss it with, pets work too. That's how our dog learned polymorphism.

**When you design a class, think about the objects that will be created from that class type. Think about:**

- things the object **knows**
- things the object **does**

ShoppingCart
cartContents
addToCart() removeFromCart() checkOut()

**knows**

Button
label color
setColor() setLabel() push() release()

**knows**

Alarm
alarmTime alarmMode
setAlarmTime() getAlarmTime() isAlarmSet() snooze()

**knows**

Things an object **knows** about itself are called

- instance variables

Things an object can do are called

- methods

**instance variables**  
(state)  
**methods**  
(behavior)

Song
title artist
setTitle() setArtist() play()

**knows**

**does**

Things an object **knows** about itself are called **instance variables**. They represent an object's state (the data) and can have unique values for each object of that type.

Think of **instance** as another way of saying **object**.

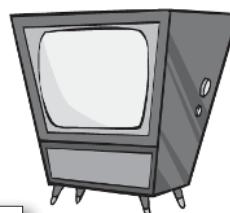
Things an object can **do** are called **methods**. When you design a class, you think about the data an object will need to know about itself, and you also design the methods that operate on that data. It's common for an object to have methods that read or write the values of the instance variables. For example, Alarm objects have an instance variable to hold the alarmTime, and two methods for getting and setting the alarmTime.

So objects have instance variables and methods, but those instance variables and methods are designed as part of the class.



**Sharpen your pencil**

Fill in what a television object might need to know and do.

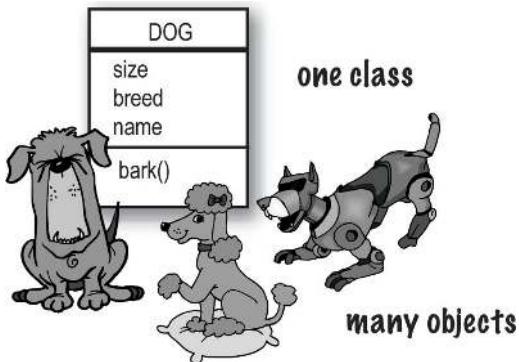


Television

instance variables

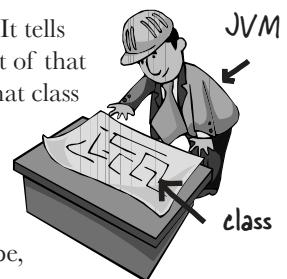
methods

# What's the difference between a class and an object?



**A class is not an object**  
**(but it's used to construct them)**

**A class is a blueprint for an object.** It tells the virtual machine *how* to make an object of that particular type. Each object made from that class can have its own values for the instance variables of that class. For example, you might use the Button class to make dozens of different buttons, and each button might have its own color, size, shape, label, and so on. Each one of these different buttons would be a button *object*.



**Look at it this way...**



**An object is like one entry in your contacts list.**

One analogy for classes and objects is your phone's contact list. Each contact has the same blank fields (the instance variables). When you create a new contact, you are creating an instance (object), and the entries you make for that contact represent its state.

The methods of the class are the things you do to a particular contact; `getName()`, `changeName()`, `setName()` could all be methods for class `Contact`.

So, each contact can *do* the same things (`getName()`, `changeName()`, etc.), but each individual contact *knows* things unique to that particular contact.

# Making your first object

So what does it take to create and use an object? You need *two* classes. One class for the type of object you want to use (Dog, AlarmClock, Television, etc.) and another class to *test* your new class. The *tester* class is where you put the main method, and in that main() method you create and access objects of your new class type. The tester class has only one job: to *try out* the methods and variables of your new object.

From this point forward in the book, you'll see two classes in many of our examples. One will be the *real* class—the class whose objects we really want to use, and the other class will be the *tester* class, which we call <WhateverYourClassNameIs>**TestDrive**. For example, if we make a **Bungee** class, we'll need a **BungeeTestDrive** class as well. Only the <SomeClassName>**TestDrive** class will have a main() method, and its sole purpose is to create objects of your new class (the not-the-tester class), and then use the dot operator (.) to access the methods and variables of the new objects. This will all be made stunningly clear by the following examples. No, *really*.

## 1 Write your class

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

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

*Instance variables*  
A method



## 2 Write a tester (TestDrive) class

```
class DogTestDrive {
    public static void main(String[] args) {
        // Dog test code goes here
    }
}
```

*Just a main method  
(we're gonna put code  
in it in the next step)*

## 3 In your tester, make an object and access the object's variables and methods

```
class DogTestDrive {
    public static void main(String[] args) {
        Dog d = new Dog(); ← Make a Dog object
        d.size = 40;
        d.bark(); ← Use the dot operator (.)
    }
}
```

*Dot operator*

*to set the size of the Dog  
and to call its bark() method*

## The Dot Operator (.)

**The dot operator (.) gives you access to an object's state and behavior (instance variables and methods).**

// make a new object

Dog d = new Dog();

// tell it to bark by using the  
// dot operator on the  
// variable d to call bark()

d.bark();

// set its size using the  
// dot operator

d.size = 40;

If you already have some OO savvy, you'll know we're not using encapsulation. We'll get there in Chapter 4, How Objects Behave.

# Making and testing Movie objects



```

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

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

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

```



MOVIE
title
genre
rating
playIt()

The MovieTestDrive class creates objects (instances) of the Movie class and uses the dot operator (.) to set the instance variables to a specific value. The MovieTestDrive class also invokes (calls) a method on one of the objects. Fill in the chart to the right with the values the three objects have at the end of main().

→ Yours to solve.

object 1

title
genre
rating

object 2

title
genre
rating

object 3

title
genre
rating

## Quick! Get out of main!

As long as you're in `main()`, you're not really in Objectville. It's fine for a test program to run within the `main` method, but in a true OO application, you need objects talking to other objects, as opposed to a static `main()` method creating and testing objects.

### The two uses of main:

- to **test** your real class
- to **launch/start** your Java application

A real Java application is nothing but objects talking to other objects. In this case, *talking* means objects calling methods on one another. On the previous page, and in Chapter 4, *How Objects Behave*, we look at using a `main()` method from a separate `TestDrive` class to create and test the methods and variables of another class. In Chapter 6, *Using the Java Library*, we look at using a class with a `main()` method to start the ball rolling on a *real* Java application (by making objects and then turning those objects loose to interact with other objects, etc.)

As a “sneak preview,” though, of how a real Java application might behave, here’s a little example. Because we’re still at the earliest stages of learning Java, we’re working with a small toolkit, so you’ll find this program a little clunky and inefficient. You might want to think about what you could do to improve it, and in later chapters that’s exactly what we’ll do. Don’t worry if some of the code is confusing; the key point of this example is that objects talk to objects.

## The Guessing Game

### Summary:

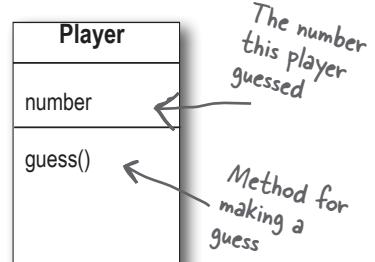
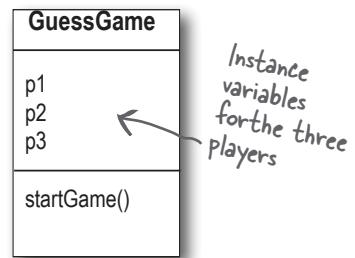
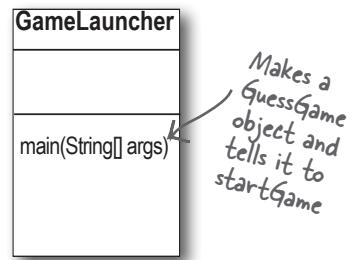
The Guessing Game involves a game object and three player objects. The game generates a random number between 0 and 9, and the three player objects try to guess it. (We didn’t say it was a really *exciting* game.)

### Classes:

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

### The Logic:

1. The `GameLauncher` class is where the application starts; it has the `main()` method.
2. In the `main()` method, a `GuessGame` object is created, and its `startGame()` method is called.
3. The `GuessGame` object’s `startGame()` method is where the entire game plays out. It creates three players and then “thinks” of a random number (the target for the players to guess). It then asks each player to guess, checks the result, and either prints out information about the winning player(s) or asks them to guess again.



```

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

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

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

        boolean plisRight = false;
        boolean p2isRight = false;
        boolean p3isRight = false;

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

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

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

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

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

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

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

            if (plisRight || p2isRight || p3isRight) {
                System.out.println("We have a winner!");
                System.out.println("Player one got it right? " + plisRight);
                System.out.println("Player two got it right? " + p2isRight);
                System.out.println("Player three got it right? " + p3isRight);
                System.out.println("Game is over.");
                break; // game over, so break out of the loop
            } else {
                // we must keep going because nobody got it right!
                System.out.println("Players will have to try again.");
            } // end if/else
        } // end loop
    } // end method
} // end class

```

*GuessGame has three instance variables for the three Player objects.*

*Create three Player objects and assign them to the three Player instance variables.*

*Declare three variables to hold the three guesses the Players make.*

*Declare three variables to hold a true or false based on the player's answer.*

*Make a 'target' number that the players have to guess.*

*Call each player's guess() method.*

*Get each player's guess (the result of their guess() method running) by accessing the number variable of each player.*

*Check each player's guess to see if it matches the target number. If a player is right, then set that player's variable to be true (remember, we set it false by default).*

*If player one OR player two OR player three is right (the || operator means OR).*

*Otherwise, stay in the loop and ask the players for another guess.*

# Running the Guessing Game



## Java takes out the Garbage

Each time an object is created in Java, it goes into an area of memory known as **The Heap**. All objects—no matter when, where, or how they're created—live on the heap. But it's not just any old memory heap; the Java heap is actually called the **Garbage-Collectible Heap**. When you create an object, Java allocates memory space on the heap according to how much that particular object needs. An object with, say, 15 instance variables, will probably need more space than an object with only two instance variables. But what happens when you need to reclaim that space? How do you get an object out of the heap when you're done with it? Java manages that memory for you! When the JVM can "see" that an object can never be used again, that object becomes *eligible for garbage collection*. And if you're running low on memory, the Garbage Collector will run, throw out the unreachable objects, and free up the space so that the space can be reused. In later chapters you'll learn more about how this works.

## Output (it will be different each time you run it)

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

## there are no Dumb Questions

**Q:** What if I need global variables and methods? How do I do that if everything has to go in a class?

**A:** There isn't a concept of "global" variables and methods in a Java OO program. In practical use, however, there are times when you want a method (or a constant) to be available to any code running in any part of your program. Think of the `random()` method in the Phrase-O-Matic app; it's a method that should be callable from anywhere. Or what about a constant like `pi`? You'll learn in Chapter 10 that marking a method as `public` and `static` makes it behave much like a "global." Any code, in any class of your application, can access a public static method. And if you mark a variable as `public`, `static`, and `final`, you have essentially made a globally available *constant*.

**Q:** Then how is this object-oriented if you can still make global functions and global data?

**A:** First of all, everything in Java goes in a class. So the constant for `pi` and the method for `random()`, although both `public` and `static`, are defined within the `Math` class. And you must keep in mind that these `static` (global-like) things are the exception rather than the rule in Java. They represent a very special case, where you don't have multiple instances/objects.

**Q:** What is a Java program? What do you actually *deliver*?

**A:** A Java program is a pile of classes (or at least one class). In a Java application, *one* of the classes must have a `main` method, used to start up the program. So as a programmer, you write one or more classes. And those classes are what you deliver. If the end user doesn't have a JVM, then you'll also need to include that with your application's classes so that they can run your program. There are a number of programs that let you bundle your classes with a JVM and create a folder or file you can share however you want (e.g., via the internet). Then the end user can install the correct version of the JVM (assuming they don't already have it on their machine).

**Q:** What if I have a hundred classes? Or a thousand? Isn't that a big pain to deliver all those individual files? Can I bundle them into one *Application Thing*?

**A:** Yes, it would be a big pain to deliver a huge bunch of individual files to your end users, but you won't have to. You can put all of your application files into a Java ARchive—a `.jar` file—that's based on the pkzip format. In the jar file, you can include a simple text file formatted as something called a *manifest*, that defines which class in that jar holds the `main()` method that should run.



### BULLET POINTS

- Object-oriented programming lets you extend a program without having to touch previously tested, working code.
- All Java code is defined in a **class**.
- A class describes how to make an object of that class type. **A class is like a blueprint**.
- An object can take care of itself; you don't have to know or care *how* the object does it.
- An object **knows** things and **does** things.
- Things an object knows about itself are called **instance variables**. They represent the *state* of an object.
- Things an object does are called **methods**. They represent the *behavior* of an object.
- When you create a class, you may also want to create a separate test class that you'll use to create objects of your new class type.
- A class can **inherit** instance variables and methods from a more abstract **superclass**.
- At runtime, a Java program is nothing more than objects "talking" to other objects.



## BE the Compiler

Each of the Java files on this page represents a complete source file. Your job is to play compiler and determine whether each of these files will compile. If they won't compile, how would you fix them, and if they do compile, what would be their output?



A

```
class StreamingSong {  
  
    String title;  
    String artist;  
    int duration;  
  
    void play() {  
        System.out.println("Playing song");  
    }  
  
    void printDetails() {  
        System.out.println("This is " + title +  
                           " by " + artist);  
    }  
}  
  
class StreamingSongTestDrive {  
    public static void main(String[] args) {  
  
        song.artist = "The Beatles";  
        song.title = "Come Together";  
        song.play();  
        song.printDetails();  
    }  
}
```

B

```
class Episode {  
  
    int seriesNumber;  
    int episodeNumber;  
  
    void skipIntro() {  
        System.out.println("Skipping intro...");  
    }  
  
    void skipToNext() {  
        System.out.println("Loading next episode...");  
    }  
}  
  
class EpisodeTestDrive {  
    public static void main(String[] args) {  
  
        Episode episode = new Episode();  
        episode.seriesNumber = 4;  
        episode.play();  
        episode.skipIntro();  
    }  
}
```

→ Answers on page 46.



## Code Magnets

A Java program is all scrambled up on the fridge. Can you reconstruct the code snippets to make a working Java program that produces the output listed below? Some of the curly braces fell on the floor and they were too small to pick up, so feel free to add as many of those as you need.

→ Answers on page 46.

d.playSnare();

DrumKit d = new DrumKit();

boolean topHat = true;  
boolean snare = true;

```
void playSnare() {  
    System.out.println("bang bang ba-bang");  
}
```

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

if (d.snare == true) {  
 d.playSnare();  
}

d.snare = false;

class DrumKitTestDrive {

d.playTopHat();

class DrumKit {

```
void playTopHat () {  
    System.out.println("ding ding da-ding");  
}
```

```
File Edit Window Help Dance  
% java DrumKitTestDrive  
bang bang ba-bang  
ding ding da-ding
```

## puzzle: Pool Puzzle



### Pool Puzzle



Your **job** is to take code snippets from the pool and place them into the blank lines in the code. You **may** use the same snippet more than once, and you won't need to use all the snippets. Your **goal** is to make classes that will compile and run and produce the output listed below. Some of the exercises and puzzles in this book might have more than one correct answer. If you find another correct answer, give yourself bonus points!

#### Output

```
File Edit Window Help Implode
%java EchoTestDrive
helloooo...
helloooo...
helloooo...
helloooo...
10
```

#### Bonus Question !

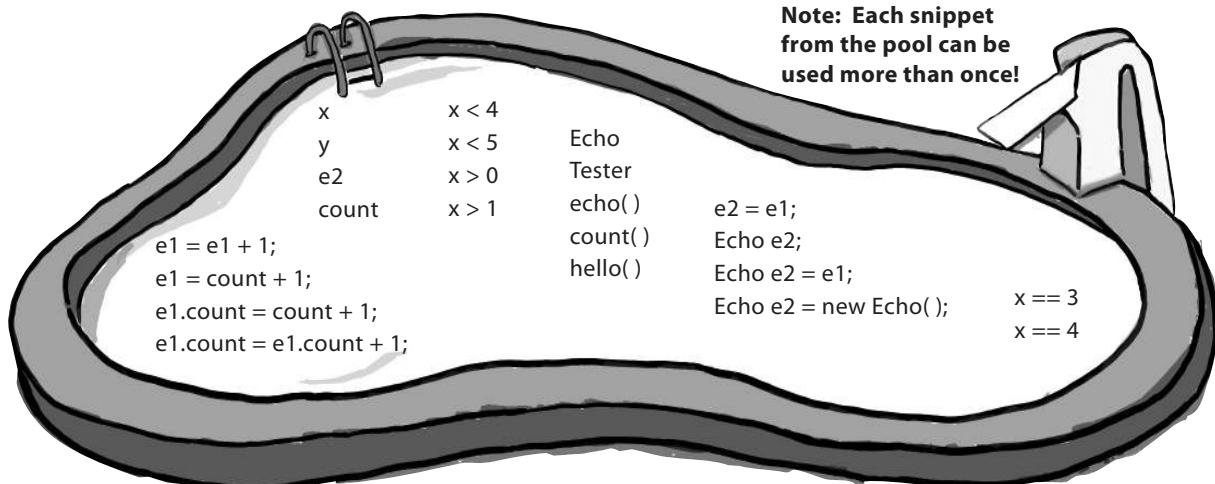
If the last line of output was **24** instead of **10**, how would you complete the puzzle?

```
public class EchoTestDrive {
    public static void main(String []
args) {
    Echo e1 = new Echo();

    int x = 0;
    while ( _____ ) {
        e1.hello();

        if ( _____ ) {
            e2.count = e2.count + 1;
        }
        if ( _____ ) {
            e2.count = e2.count + e1.count;
        }
        x = x + 1;
    }
    System.out.println(e2.count);
}
```

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





# Who Am I?

A bunch of Java components, in full costume, are playing a party game, "Who am I?" They give you a clue, and you try to guess who they are, based on what they say. Assume they always tell the truth about themselves. If they happen to say something that could be true for more than one of them, choose all for whom that sentence can apply. Fill in the blanks next to the sentence with the names of one or more attendees. The first one's on us.

**Tonight's attendees:**

**Class   Method   Object   Instance variable**

I am compiled from a .java file.

**class**

---

My instance variable values can be different from my buddy's values.

---

I behave like a template.

---

I like to do stuff.

---

I can have many methods.

---

I represent "state."

---

I have behaviors.

---

I am located in objects.

---

I live on the heap.

---

I am used to create object instances.

---

My state can change.

---

I declare methods.

---

I can change at runtime.

---

—————> Answers on page 47.

## exercise solutions



### Code Magnets (from page 43)

```
class DrumKit {  
    boolean topHat = true;  
    boolean snare = true;  
  
    void playTopHat() {  
        System.out.println("ding ding da-ding");  
    }  
  
    void playSnare() {  
        System.out.println("bang bang ba-bang");  
    }  
}  
  
class DrumKitTestDrive {  
    public static void main(String[] args) {  
        DrumKit d = new DrumKit();  
        d.playSnare();  
        d.snare = false;  
        d.playTopHat();  
  
        if (d.snare == true) {  
            d.playSnare();  
        }  
    }  
}
```

```
File Edit Window Help Dance  
% java DrumKitTestDrive  
bang bang ba-bang  
ding ding da-ding
```

### BE the Compiler (from page 42)

A

```
class StreamingSong {  
    String title;  
    String artist;  
    int duration;  
  
    void play() {  
        System.out.println("Playing song");  
    }  
  
    void printDetails() {  
        System.out.println("This is " + title +  
                           " by " + artist);  
    }  
}  
  
class StreamingSongTestDrive {  
    public static void main(String[] args) {  
  
        StreamingSong song = new StreamingSong();  
        song.artist = "The Beatles";  
        song.title = "Come Together";  
        song.play();  
        song.printDetails();  
    }  
}
```

We've got the template, now we have to make an object!

---

B

```
class Episode {  
    int seriesNumber; The line: episode.play();  
    int episodeNumber; wouldn't compile without a play  
                      method in the episode class!  
    void play() {  
        System.out.println("Playing episode " + episodeNumber);  
    }  
  
    void skipIntro() {  
        System.out.println("Skipping intro...");  
    }  
  
    void skipToNext() {  
        System.out.println("Loading next episode...");  
    }  
}  
  
class EpisodeTestDrive {  
    public static void main(String[] args) {  
        Episode episode = new Episode();  
        episode.seriesNumber = 4;  
        episode.play();  
        episode.skipIntro();  
    }  
}
```



## Puzzle Solutions

### Pool Puzzle (from page 44)

```
public class EchoTestDrive {
    public static void main(String[] args) {
        Echo e1 = new Echo();
        Echo e2 = new Echo(); // correct answer
        - or -
        Echo e2 = e1; // bonus "24" answer
        int x = 0;
        while (x < 4) {
            e1.hello();
            e1.count = e1.count + 1;
            if (x == 3) {
                e2.count = e2.count + 1;
            }
            if (x > 0) {
                e2.count = e2.count + e1.count;
            }
            x = x + 1;
        }
        System.out.println(e2.count);
    }
}

class Echo {
    int count = 0;

    void hello() {
        System.out.println("helloooo... ");
    }
}
```

```
File Edit Window Help Assimilate
%java EchoTestDrive
helloooo...
helloooo...
helloooo...
helloooo...
10
```

### Who Am I? (from page 45)

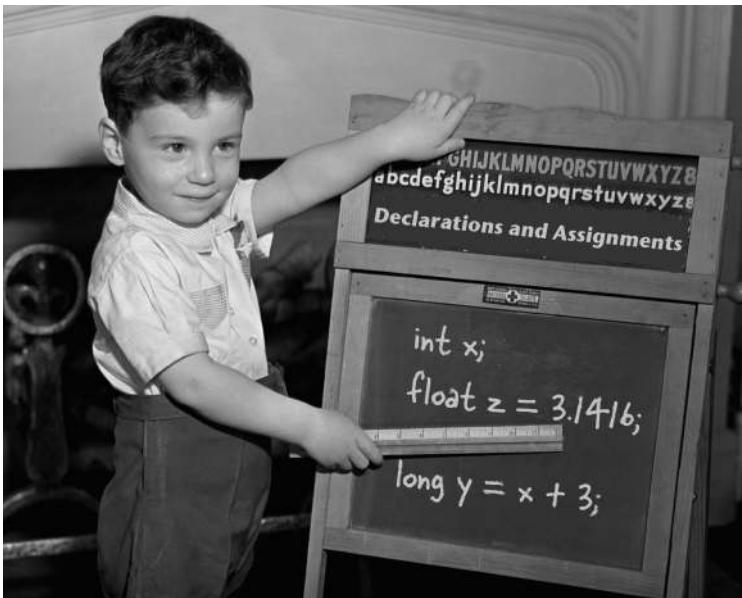
I am compiled from a .java file.	class
My instance variable values can be different from my buddy's values.	object
I behave like a template.	class
I like to do stuff.	object, method
I can have many methods.	class, object
I represent "state."	instance variable
I have behaviors.	object, class
I am located in objects.	method, instance variable
I live on the heap.	object
I am used to create object instances.	class
My state can change.	object, instance variable
I declare methods.	class
I can change at runtime.	object, instance variable

Note: both classes and objects are said to have state and behavior. They're defined in the class, but the object is also said to "have" them. Right now, we don't care where they technically live.



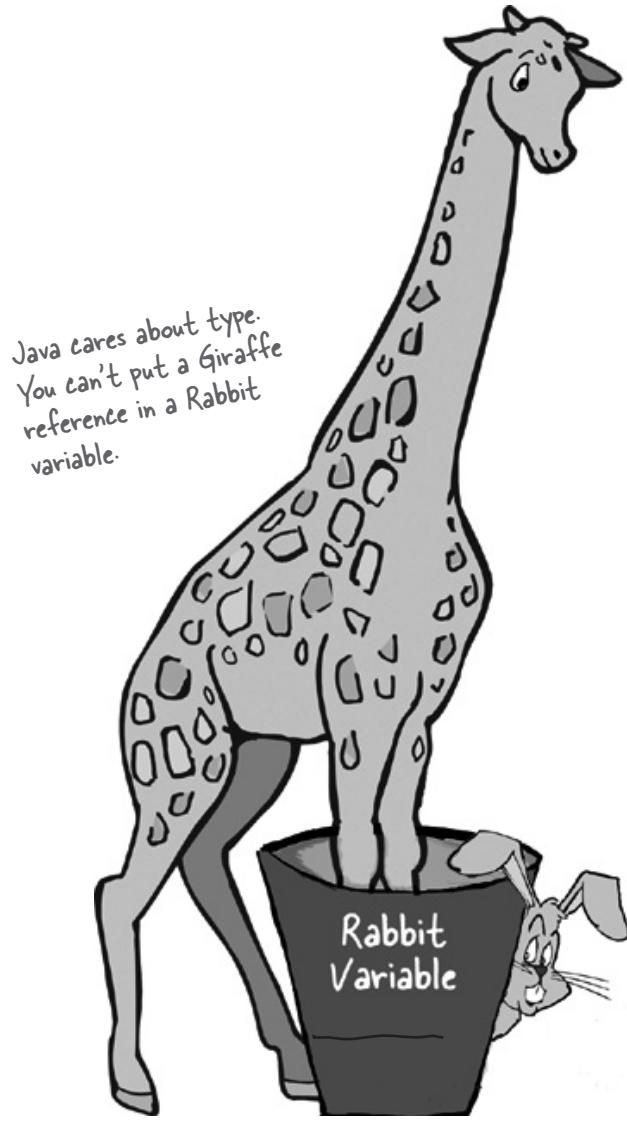
### 3 primitives and references

# Know Your Variables



**Variables can store two types of things: primitives and references.**

So far you've used variables in two places—as object **state** (instance variables) and as **local** variables (variables declared within a *method*). Later, we'll use variables as **arguments** (values sent to a method by the calling code), and as **return types** (values sent back to the caller of the method). You've seen variables declared as simple **primitive** integer values (type `int`). You've seen variables declared as something more **complex** like a `String` or an array. But **there's gotta be more to life** than integers, `Strings`, and arrays. What if you have a `PetOwner` object with a `Dog` instance variable? Or a `Car` with an `Engine`? In this chapter we'll unwrap the mysteries of Java types (like the difference between primitives and references) and look at what you can *declare* as a variable, what you can *put* in a variable, and what you can *do* with a variable. And we'll finally see what life is *truly* like on the garbage-collectible heap.



## Declaring a variable

**Java cares about type.** It won't let you do something bizarre and dangerous like stuff a Giraffe reference into a Rabbit variable—what happens when someone tries to ask the so-called *Rabbit* to *hop ()*? And it won't let you put a floating-point number into an integer variable, unless you *tell the compiler* that you know you might lose precision (like, everything after the decimal point).

The compiler can spot most problems:

```
Rabbit hopper = new Giraffe();
```

Don't expect that to compile. *Thankfully*.

For all this type-safety to work, you must declare the type of your variable. Is it an integer? a Dog? A single character? Variables come in two flavors:

**primitive** and **object reference**. Primitives hold fundamental values (think: simple bit patterns) including integers, booleans, and floating-point numbers. Object references hold, well, *references to objects* (gee, didn't *that* clear it up).

We'll look at primitives first and then move on to what an object reference really means. But regardless of the type, you must follow two declaration rules:

---

### variables must have a type

---

Besides a type, a variable needs a name so that you can use that name in code.

---

### variables must have a name

---

```
int count;
```

↑  
type      ↑  
          name

Note: When you see a statement like: “an object of **type X**,” think of *type* and *class* as synonyms. (We'll refine that a little more in later chapters.)

## "I'd like a double mocha, no, make it an int."

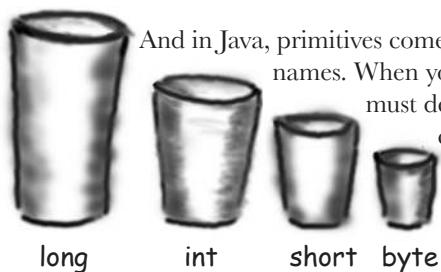
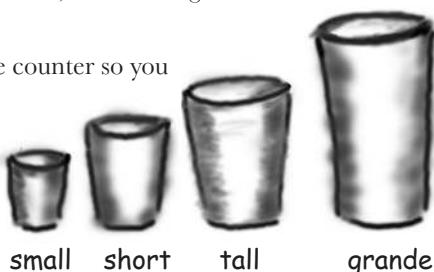
When you think of Java variables, think of cups. Coffee cups, tea cups, giant cups that hold lots and lots of your favorite drink, those big cups the popcorn comes in at the movies, cups with wonderful tactile handles, and cups with metallic trim that you learned can never, ever go in the microwave.

**A variable is just a cup. A container. It holds something.**

It has a size and a type. In this chapter, we're going to look first at the variables (cups) that hold **primitives**: then a little later we'll look at cups that hold *references to objects*. Stay with us here on the whole cup analogy—as simple as it is right now, it'll give us a common way to look at things when the discussion gets more complex. And that'll happen soon.

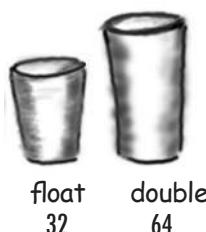
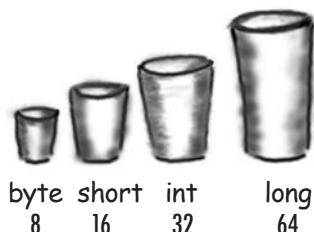
Primitives are like the cups they have at the coffee shop. If you've been to a Starbucks, you know what we're talking about here. They come in different sizes, and each has a name like "short," "tall," and, "I'd like a 'grande' mocha half-caff with extra whipped cream."

You might see the cups displayed on the counter so you can order appropriately:



And in Java, primitives come in different sizes, and those sizes have names. When you declare any variable in Java, you must declare it with a specific type. The four containers here are for the four integer primitives in Java.

Each cup holds a value, so for Java primitives, rather than saying, "I'd like a tall french roast," you say to the compiler, "I'd like an int variable with the number 90 please." Except for one tiny difference...in Java you also have to give your cup a *name*. So it's actually, "I'd like an int please, with the value of 2486, and name the variable **height**." Each primitive variable has a fixed number of bits (cup size). The sizes for the six numeric primitives in Java are shown below:



## Primitive Types

Type	Bit Depth	Value Range
------	-----------	-------------

### boolean and char

boolean (JVM-specific) **true or false**

char 16 bits 0 to 65535

### numeric (all are signed)

#### integer

byte 8 bits -128 to 127

short 16 bits -32768 to 32767

int 32 bits -2147483648 to 2147483647

long 64 bits -huge to huge

#### floating point

float 32 bits varies

double 64 bits varies

---

### Primitive declarations with assignments:

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

Note the 'f' and 'L'. With some number types, you have to specifically tell the compiler what you mean, or it might get confused between similar-looking number types. You can use upper or lowercase.

# You really don't want to spill that...

Be sure the value can fit into the variable.



Why doesn't this work, you ask? After all, the value of *x* is 24, and 24 is definitely small enough to fit into a byte. *You* know that, and *we* know that, but all the compiler cares about is that you're trying to put a big thing into a small thing, and there's the *possibility* of spilling. Don't expect the compiler to know what the value of *x* is, even if you happen to be able to see it literally in your code.

**You can assign a value to a variable in one of several ways including:**

- type a *literal* value after the equals sign (*x = 12*, *isGood = true*, etc.)
- assign the value of one variable to another (*x = y*)
- use an expression combining the two (*x = y + 43*)

In the examples below, the literal values are in bold italics:

```
int size = 32;           declare an int named size, assign it the value 32
char initial = 'j';     declare a char named initial, assign it the value 'j' 
double d = 456.709;    declare a double named d, assign it the value 456.709
boolean isLearning;       declare a boolean named isCrazy (no assignment)
isLearning = true;      assign the value true to the previously declared isCrazy
int y = x + 456;       declare an int named y, assign it the value that is the sum
                        of whatever x is now plus 456
```

## Sharpen your pencil

The compiler won't let you put a value from a large cup into a small one. But what about the other way—pouring a small cup into a big one? **No problem.**

Based on what you know about the size and type of the primitive variables, see if you can figure out which of these are legal and which aren't. We haven't covered all the rules yet, so on some of these you'll have to use your best judgment. **Tip:** The compiler always errs on the side of safety.

From the following list, **Circle** the statements that would be legal if these lines were in a single method:

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;**

→ Answers on page 68.

# Back away from that keyword!

You know you need a name and a type for your variables.

You already know the primitive types.

**But what can you use as names?** The rules are simple. You can name a class, method, or variable according to the following rules (the real rules are slightly more flexible, but these will keep you safe):

- **It must start with a letter, underscore (\_), or dollar sign (\$). You can't start a name with a number.**
- **After the first character, you can use numbers as well. Just don't start it with a number.**
- **It can be anything you like, subject to those two rules, just so long as it isn't one of Java's reserved words.**

Reserved words are keywords (and other things) that the compiler recognizes. And if you really want to play confuse-a-compiler, then just *try* using a reserved word as a name.

You've already seen some reserved words:

public    static    void

don't use any of these  
for your own names.

And the primitive types are reserved as well:

boolean    char    byte    short    int    long    float    double

But there are a lot more we haven't discussed yet. Even if you don't need to know what they mean, you still need to know you can't use 'em yourself. **Do not—under any circumstances—try to memorize these now.** To make room for these in your head, you'd probably have to lose something else. Like where your car is parked. Don't worry, by the end of the book you'll have most of them down cold.

## This table reserved

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

Java's keywords, reserved words, and special identifiers. If you use these for names, the compiler will *probably* be very, very upset.



## Controlling your Dog object

You know how to declare a primitive variable and assign it a value. But now what about non-primitive variables? In other words, *what about objects?*

- There is actually no such thing as an object variable.
- There's only an object reference variable.
- An object reference variable holds bits that represent a way to access an object.
- It doesn't hold the object itself, but it holds something like a pointer. Or an address. Except, in Java we don't really know what is inside a reference variable. We do know that whatever it is, it represents one and only one object. And the JVM knows how to use the reference to get to the object.

You can't stuff an object into a variable. We often think of it that way...we say things like, "I passed the String to the System.out.println() method." Or, "The method returns a Dog" or, "I put a new Foo object into the variable named myFoo."

But that's not what happens. There aren't giant expandable cups that can grow to the size of any object.

Objects live in one place and one place only—the garbage-collectible heap! (You'll learn more about that later in this chapter.)

Although a primitive variable is full of bits representing the actual **value** of the variable, an object reference variable is full of bits representing **a way to get to the object**.

You use the dot operator (.) on a reference variable to say, "use the thing before the dot to get me the thing after the dot." For example:

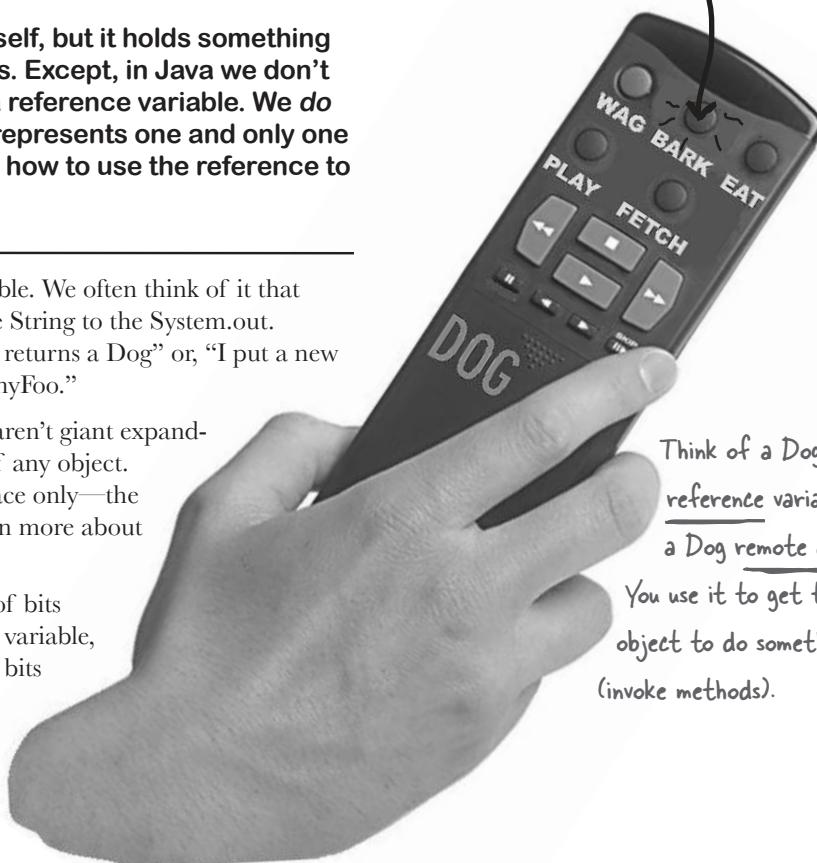
```
myDog.bark();
```

means, "use the object referenced by the variable myDog to invoke the bark() method." When you use the dot operator on an object reference variable, think of it like pressing a button on the remote control for that object.

**Dog d = new Dog();  
d.bark();**

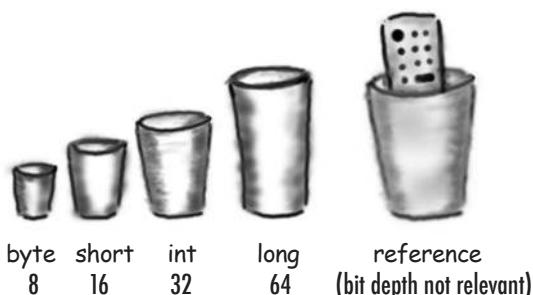
think of this

like this



Think of a Dog  
reference variable as  
a Dog remote control.

You use it to get the  
object to do something  
(invoke methods).



## An object reference is just another variable value

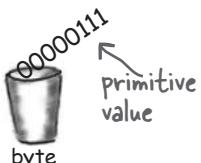
Something that goes in a cup.

Only this time, the value is a remote control.

### Primitive Variable

`byte x = 7;`

The bits representing 7 go into the variable (00000111).

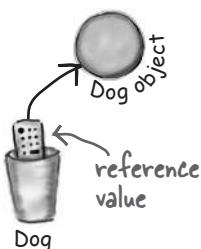


### Reference Variable

`Dog myDog = new Dog();`

The bits representing a way to get to the Dog object go into the variable.

**The Dog object itself does not go into the variable!**



With primitive variables, the value of the variable is...the value (5, -26.7, 'a').

With reference variables, the value of the variable is...bits representing a way to get to a specific object.

You don't know (or care) how any particular JVM implements object references. Sure, they might be a pointer to a pointer to...but even if you know, you still can't use the bits for anything other than accessing an object.

We don't care how many 1s and 0s there are in a reference variable. It's up to each JVM and the phase of the moon.

## The 3 steps of object declaration, creation and assignment

`Dog myDog = new Dog();`

**1** Declare a reference variable

`Dog myDog = new Dog();`

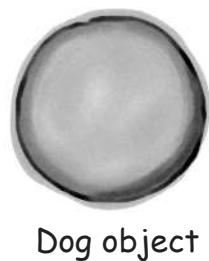
Tells the JVM to allocate space for a reference variable, and names that variable *myDog*. The reference variable is, forever, of type *Dog*. In other words, a remote control that has buttons to control a *Dog*, but not a *Cat* or a *Button* or a *Socket*.



**2** Create an object

`Dog myDog = new Dog();`

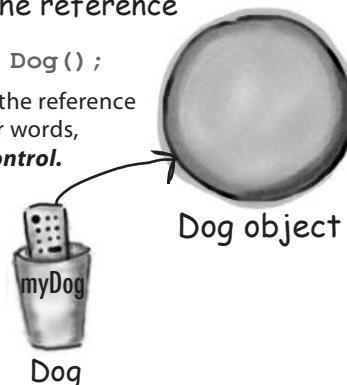
Tells the JVM to allocate space for a new *Dog* object on the heap (we'll learn a lot more about that process, especially in Chapter 9, *Life and Death of an Object*).



**3** Link the object and the reference

`Dog myDog = new Dog();`

Assigns the new *Dog* to the reference variable *myDog*. In other words, *programs the remote control*.



## there are no Dumb Questions

**Q:** How big is a reference variable?

**A:** You don't know. Unless you're cozy with someone on the JVM's development team, you don't know how a reference is represented. There are pointers in there somewhere, but you can't access them. You won't need to. (OK, if you insist, you might as well just imagine it to be a 64-bit value.) But when you're talking about memory allocation issues, your Big Concern should be about how many *objects* (as opposed to *object references*) you're creating and how big *they* (the *objects*) really are.

**Q:** So, does that mean that all object references are the same size, regardless of the size of the actual objects to which they refer?

**A:** Yep. All references for a given JVM will be the same size regardless of the objects they reference, but each JVM might have a different way of representing references, so references on one JVM may be smaller or larger than references on another JVM.

**Q:** Can I do arithmetic on a reference variable, increment it, you know—C stuff?

**A:** Nope. Say it with me again, "Java is not C."



**HeadFirst:** So, tell us, what's life like for an object reference?

**Reference:** Pretty simple, really. I'm a remote control, and I can be programmed to control different objects.

**HeadFirst:** Do you mean different objects even while you're running? Like, can you refer to a Dog and then five minutes later refer to a Car?

**Reference:** Of course not. Once I'm declared, that's it. If I'm a Dog remote control, then I'll never be able to point (oops—my bad, we're not supposed to say *point*), I mean, refer to anything but a Dog.

**HeadFirst:** Does that mean you can refer to only one Dog?

**Reference:** No. I can be referring to one Dog, and then five minutes later I can refer to some other Dog. As long as it's a Dog, I can be redirected (like reprogramming your remote to a different TV) to it. Unless...no never mind.

**HeadFirst:** No, tell me. What were you gonna say?

**Reference:** I don't think you want to get into this now, but I'll just give you the short version—if I'm marked as `final`, then once I am assigned a Dog, I can never be reprogrammed to anything else but *that* one and only Dog. In other words, no other object can be assigned to me.

**HeadFirst:** You're right, we don't want to talk about that now. OK, so unless you're `final`, then you can refer to one Dog and then refer to a different Dog later. Can you ever refer to *nothing at all*? Is it possible to not be programmed to anything?

**Reference:** Yes, but it disturbs me to talk about it.

**HeadFirst:** Why is that?

**Reference:** Because it means I'm `null`, and that's upsetting to me.

**HeadFirst:** You mean, because then you have no value?

**Reference:** Oh, `null` is a value. I'm still a remote control, but it's like you brought home a new universal remote control and you don't have a TV. I'm not programmed to control anything. They can press my buttons all day long, but nothing good happens. I just feel so...useless. A waste of bits. Granted, not that many bits, but still. And that's not the worst part. If I am the only reference to a particular object and then I'm set to `null` (deprogrammed), it means that now *nobody* can get to that object I had been referring to.

**HeadFirst:** And that's bad because...

**Reference:** You have to *ask*? Here I've developed a relationship with this object, an intimate connection, and then the tie is suddenly, cruelly, severed. And I will never see that object again, because now it's eligible for [producer, cue tragic music] *garbage collection*. Sniff. But do you think programmers ever consider *that*? Snif. Why, *why* can't I be a primitive? *I hate being a reference*. The responsibility, all the broken attachments...

## Life on the garbage-collectible heap

`Book b = new Book();`

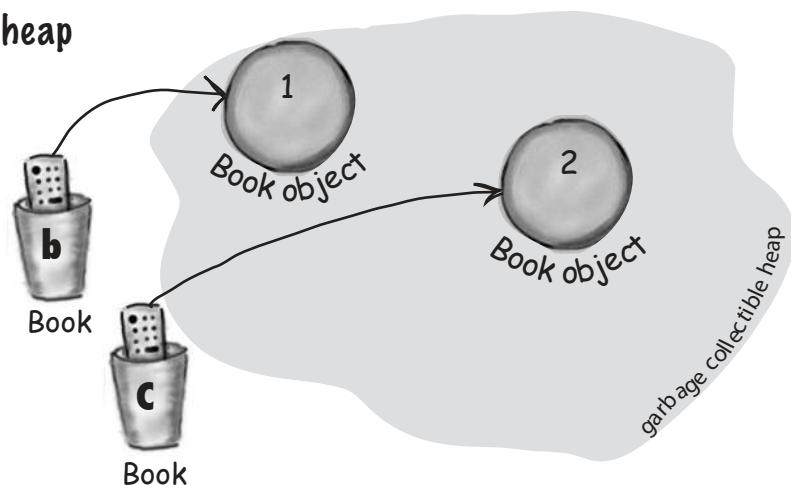
`Book c = new Book();`

Declare two Book reference variables. Create two new Book objects. Assign the Book objects to the reference variables.

The two Book objects are now living on the heap.

References: 2

Objects: 2



`Book d = c;`

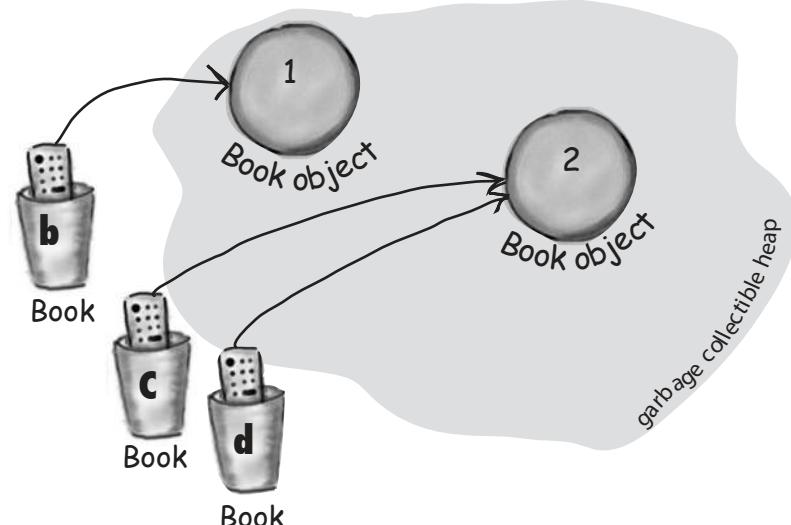
Declare a new Book reference variable. Rather than creating a new, third Book object, assign the value of variable `c` to variable `d`. But what does this mean? It's like saying "Take the bits in `c`, make a copy of them, and stick that copy into `d`".

**Both `c` and `d` refer to the same object.**

**The `c` and `d` variables hold two different copies of the same value. Two remotes programmed to one TV.**

References: 3

Objects: 2



`c = b;`

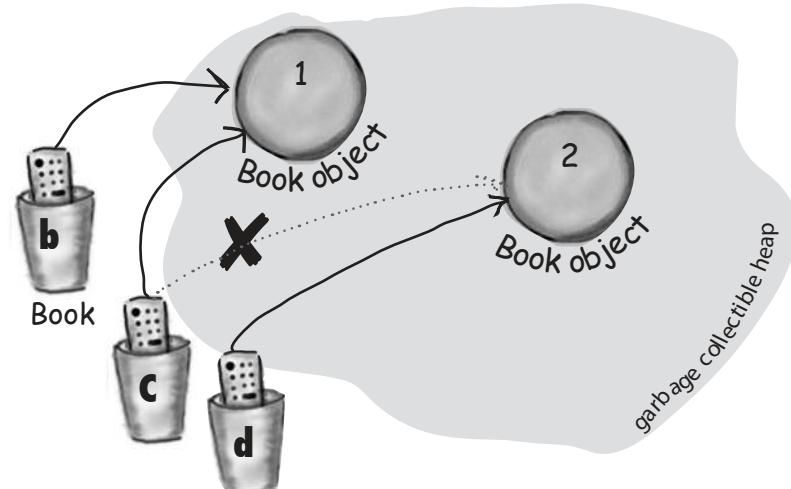
Assign the value of variable `b` to variable `c`. By now you know what this means. The bits inside variable `b` are copied, and that new copy is stuffed into variable `c`.

**Both `b` and `c` refer to the same object.**

**The `c` variable no longer refers to its old Book object.**

References: 3

Objects: 2



objects on the heap

## Life and death on the heap

`Book b = new Book();`

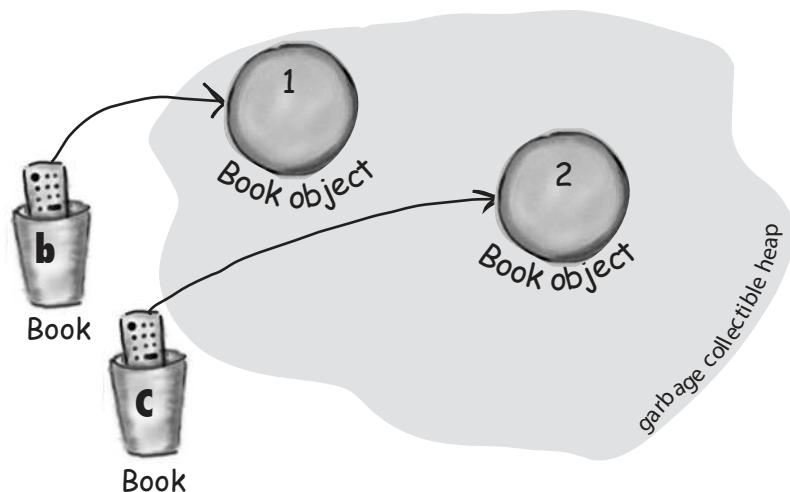
`Book c = new Book();`

Declare two Book reference variables. Create two new Book objects. Assign the Book objects to the reference variables.

The two book objects are now living on the heap.

Active References: 2

Reachable Objects: 2



`b = c;`

Assign the value of variable `c` to variable `b`. The bits inside variable `c` are copied, and that new copy is stuffed into variable `b`. Both variables hold identical values.

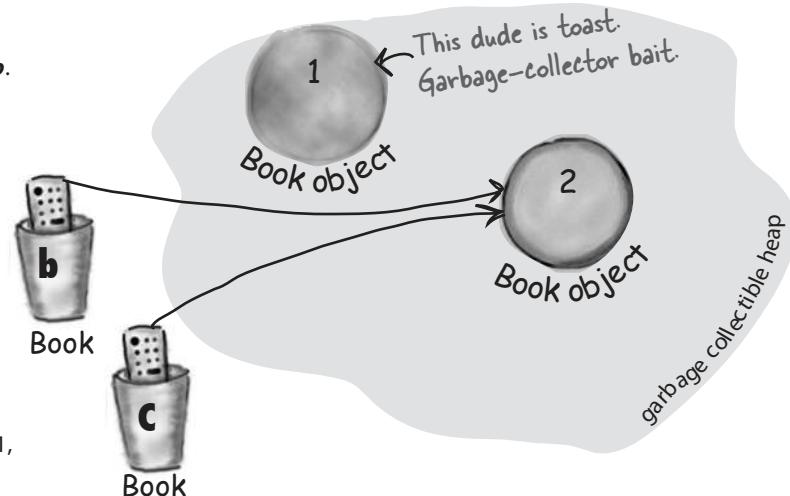
**Both `b` and `c` refer to the same object. Object 1 is abandoned and eligible for Garbage Collection (GC).**

Active References: 2

Reachable Objects: 1

Abandoned Objects: 1

The first object that `b` referenced, Object 1, has no more references. It's *unreachable*.



`c = null;`

Assign the value `null` to variable `c`. This makes `c` a *null reference*, meaning it doesn't refer to anything. But it's still a reference variable, and another Book object can still be assigned to it.

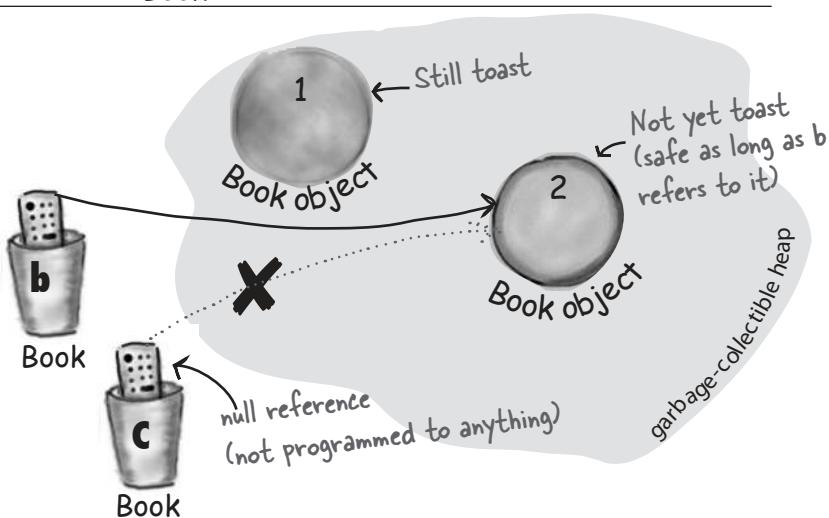
**Object 2 still has an active reference (b), and as long as it does, the object is not eligible for GC.**

Active References: 1

`null` References: 1

Reachable Objects: 1

Abandoned Objects: 1



# An array is like a tray of cups

The Java standard library includes lots of sophisticated data structures including maps, trees, and sets (see Appendix B), but arrays are great when you just want a quick, ordered, efficient list of things. Arrays give you fast random access by letting you use an index position to get to any element in the array.

Every element in an array is just a variable. In other words, one of the eight primitive variable types (think: Large Furry Dog) or a reference variable. Anything you would put in a *variable* of that type can be assigned to an

*array element* of that type. So in an array of type int (int[]), each element can hold an int. In a Dog array (Dog[]) each element can hold...a Dog? No, remember that a reference variable just holds a reference (a remote control), not the object itself. So in a Dog array, each element can hold a *remote control* to a Dog. Of course, we still have to make the Dog objects...and you'll see all that on the next page.

Be sure to notice one key thing in the picture—**the array is an object, even though it's an array of primitives.**

- 1 Declare an int array variable. An array variable is a remote control to an array object.

```
int[] nums;
```

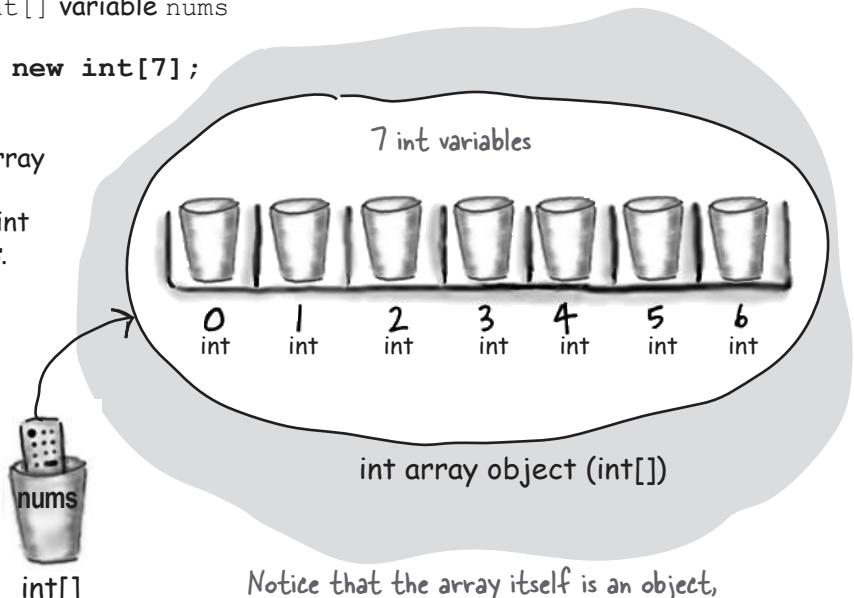
- 2 Create a new int array with a length of 7, and assign it to the previously declared int[] variable nums

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

- 3 Give each element in the array some int value.  
Remember, elements in an int array are just int variables.

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

7 int variables



Notice that the array itself is an object, even though the 7 elements are primitives.

## Arrays are objects too

You can have an array object that's declared to *hold* primitive values. In other words, the array object can have *elements* that are primitives, but the array itself is *never* a primitive.

**Regardless of what the array holds, the array itself is always an object!**

an array of objects

## Make an array of Dogs

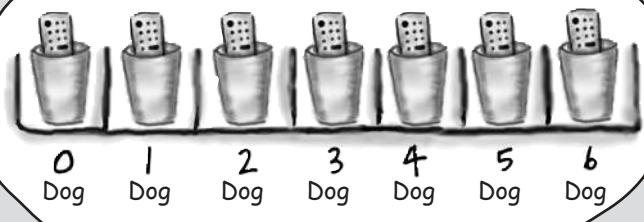
- 1 Declare a Dog array variable  
`Dog[] pets;`

- 2 Create a new Dog array with a length of 7, and assign it to the previously declared `Dog[]` variable `pets`

```
pets = new Dog[7];
```

**What's missing?**

Dogs! We have an array of Dog references, but no actual Dog objects!



Dog array object (`Dog[]`)

- 3 Create new Dog objects, and assign them to the array elements.

Remember, elements in a Dog array are just Dog reference variables. We still need Dogs!

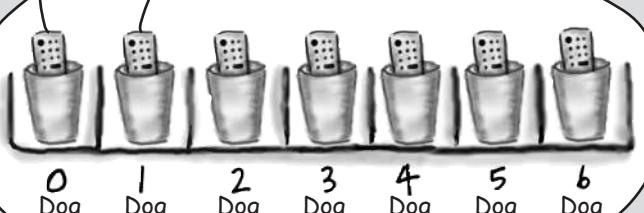
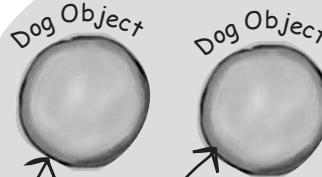
```
pets[0] = new Dog();  
pets[1] = new Dog();
```

→ Yours to solve.

**Sharpen your pencil**

What is the current value of `pets[2]`? \_\_\_\_\_

What code would make `pets[3]` refer to one of the two existing Dog objects?  
\_\_\_\_\_



Dog array object (`Dog[]`)



Dog
name
bark()
eat()
chaseCat()

### Java cares about type.

Once you've declared an array, you can't put anything in it except things that are of a compatible array type.

For example, you can't put a Cat into a Dog array (it would be pretty awful if someone thinks that only Dogs are in the array, so they ask each one to bark, and then to their horror discover there's a cat lurking.) And you can't stick a double into an int array (spillage, remember?). You can, however, put a byte into an int array, because a byte will always fit into an int-sized cup. This is known as an **implicit widening**. We'll get into the details later; for now just remember that the compiler won't let you put the wrong thing in an array, based on the array's declared type.

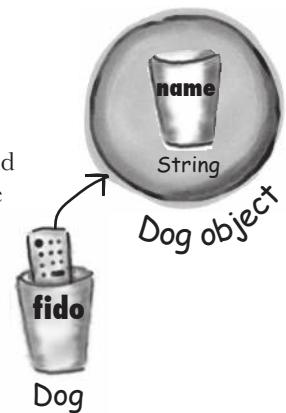
## Control your Dog (with a reference variable)

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

We created a Dog object and used the dot operator on the reference variable **fido** to access the name variable.\*

We can use the **fido** reference to get the dog to bark() or eat() or chaseCat().

```
fido.bark();
fido.chaseCat();
```



## What happens if the Dog is in a Dog array?

We know we can access the Dog's instance variables and methods using the dot operator, but *on what?*

When the Dog is in an array, we don't have an actual variable name (like **fido**). Instead we use array notation and push the remote control button (dot operator) on an object at a particular index (position) in the array:

```
Dog[] myDogs = new Dog[3];
myDogs[0] = new Dog();
myDogs[0].name = "Fido";
myDogs[0].bark();
```

\*Yes we know we're not demonstrating encapsulation here, but we're trying to keep it simple. For now. We'll do encapsulation in Chapter 4.

## using references

```
class Dog {  
    String name;  
  
    public static void main(String[] args) {  
        // make a Dog object and access it  
        Dog dog1 = new Dog();  
        dog1.bark();  
        dog1.name = "Bart"; ←  
  
        // now make a Dog array  
        Dog[] myDogs = new Dog[3];  
        // and put some dogs in it  
        myDogs[0] = new Dog();  
        myDogs[1] = new Dog();  
        myDogs[2] = dog1;  
  
        // now access the Dogs using the array  
        // references  
        myDogs[0].name = "Fred";  
        myDogs[1].name = "Marge";  
  
        // Hmm... what is myDogs[2] name?  
        System.out.print("last dog's name is ");  
        System.out.println(myDogs[2].name);  
  
        // now loop through the array  
        // and tell all dogs to bark  
        int x = 0; ←  
        while (x < myDogs.length) {  
            myDogs[x].bark();  
            x = x + 1;  
        }  
  
        public void bark() {  
            System.out.println(name + " says Ruff!");  
        }  
  
        public void eat() {}  
  
        public void chaseCat() {}  
    }
```

Strings are a special type of object. You can create and assign them as if they were primitives (even though they're references).

## A Dog example

Dog
name
bark()
eat()
chaseCat()

### Output

```
File Edit Window Help Howl  
% java Dog  
null says Ruff!  
last dog's name is Bart  
Fred says Ruff!  
Marge says Ruff!  
Bart says Ruff!
```

### BULLET POINTS

- Variables come in two flavors: primitive and reference.
- Variables must always be declared with a name and a type.
- A primitive variable value is the bits representing the value (5, 'a', true, 3.1416, etc.).
- A reference variable value is the bits representing a way to get to an object on the heap.
- A reference variable is like a remote control. Using the dot operator (.) on a reference variable is like pressing a button on the remote control to access a method or instance variable.
- A reference variable has a value of `null` when it is not referencing any object.
- An array is always an object, even if the array is declared to hold primitives. There is no such thing as a primitive array, only an array that holds primitives.



## BE the Compiler



Each of the Java files on this page represents a complete source file. Your job is to play compiler and determine whether each of these files will compile and run without exception. If they won't, how would you fix them?

**A**

```
class Books {
    String title;
    String author;
}

class BooksTestDrive {
    public static void main(String[] args) {
        Books[] myBooks = new Books[3];
        int x = 0;
        myBooks[0].title = "The Grapes of Java";
        myBooks[1].title = "The Java Gatsby";
        myBooks[2].title = "The Java Cookbook";
        myBooks[0].author = "bob";
        myBooks[1].author = "sue";
        myBooks[2].author = "ian";

        while (x < 3) {
            System.out.print(myBooks[x].title);
            System.out.print(" by ");
            System.out.println(myBooks[x].author);
            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 + " is a ");
            System.out.println("good Hobbit name");
        }
    }
}
```

→ Answers on page 68.

## exercise: Code Magnets



## Code Magnets

A working Java program is all scrambled up on the fridge. Can you reconstruct the code snippets to make a working Java program that produces the output listed below? Some of the curly braces fell on the floor and they were too small to pick up, so feel free to add as many of those as you need!

int y = 0;

ref = index[y];

islands[0] = "Bermuda";  
islands[1] = "Fiji";  
islands[2] = "Azores";  
islands[3] = "Cozumel";

int ref;

while (y < 4) {

System.out.println(islands[ref]);

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

String [] islands = new String[4];

System.out.print("island = ");

int [] index = new int[4];

y = y + 1;

```
File Edit Window Help Sunscreen  
% java TestArrays  
island = Fiji  
island = Cozumel  
island = Bermuda  
island = Azores
```

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



## Pool Puzzle



Your **job** is to take code snippets from the pool and place them into the blank lines in the code. You **may** use the same snippet more than once, and you won't need to use all the snippets. Your **goal** is to make a class that will compile and run and produce the output listed.

### Output

```
File Edit Window Help Bermuda
%java Triangle
triangle 0, area = 4.0
triangle 1, area = 10.0
triangle 2, area = 18.0
triangle 3, area = _____
y = _____
```

### Bonus Question!

For extra bonus points, use snippets from the pool to fill in the missing output (above).

```
class Triangle {
    double area;
    int height;
    int length;
```

(Sometimes we don't use a separate test class, because we're trying to save space on the page.)

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

---



---

```
while ( _____ ) {
```

---

```
    _____ .height = (x + 1) * 2;
    _____ .length = x + 4;
```

---

```
    System.out.print("triangle " + x + ", area");
    System.out.println(" = " + _____ .area);
```

---

```
}
```

---

```
x = 27;
Triangle t5 = ta[2];
ta[2].area = 343;
System.out.print("y = " + y);
System.out.println(" , t5 area = " + t5.area);
```

```
}
```

```
void setArea() {
    _____ = (height * length) / 2;
}
```

**Note:** Each snippet from the pool can be used more than once!

```
4, t5 area = 18.0
4, t5 area = 343.0
27, t5 area = 18.0
27, t5 area = 343.0
ta[x] = setArea();
ta.x = setArea();
ta[x].setArea();
Triangle [] ta = new Triangle(4);
Triangle ta = new [] Triangle[4];
Triangle [] ta = new Triangle[4];
int x;
int y;
int x = 0;
int x = 1;
int y = x;
28.0
29.0
30.0
ta = new Triangle();
ta[x] = new Triangle();
ta.x = new Triangle();
x = x + 1;
x = x + 2;
x = x - 1;
ta.x
ta[x]
x < 4
x < 5
```

## puzzle: Heap o' Trouble



## A Heap o' Trouble

A short Java program is listed to the right. When “// do stuff” is reached, some objects and some reference variables will have been created. Your task is to determine which of the reference variables refer to which objects. Not all the reference variables will be used, and some objects might be referred to more than once. Draw lines connecting the reference variables with their matching objects.

**Tip:** Unless you’re way smarter than we are, you probably need to draw diagrams like the ones on page 57–60 of this chapter. Use a pencil so you can draw and then erase reference links (the arrows going from a reference remote control to an object).

```
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];  
        // do stuff  
    }  
}
```

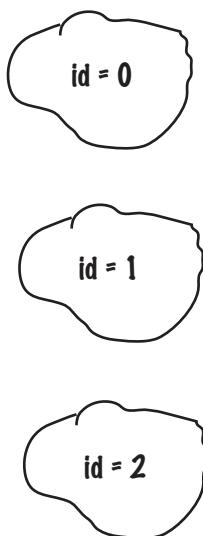
Match each reference variable with matching object(s).

You might not have to use every reference.

Reference Variables:



HeapQuiz Objects:



→ Answers on page 69.



## Five-Minute Mystery



### The case of the pilfered references

It was a dark and stormy night. Tawny strolled into the programmers' bullpen like she owned the place. She knew that all the programmers would still be hard at work, and she wanted help. She needed a new method added to the pivotal class that was to be loaded into the client's new top-secret Java-enabled cell phone. Heap space in the cell phone's memory was tight, and everyone knew it. The normally raucous buzz in the bullpen fell to silence as Tawny eased her way to the white board. She sketched a quick overview of the new method's functionality and slowly scanned the room. "Well folks, it's crunch time," she purred. "Whoever creates the most memory efficient version of this method is coming with me to the client's launch party on Maui tomorrow...to help me install the new software."

The next morning Tawny glided into the bullpen. "Ladies and Gentlemen," she smiled, "the plane leaves in a few hours, show me what you've got!" Bob went first; as he began to sketch his design on the white board, Tawny said, "Let's get to the point Bob, show me how you handled updating the list of contact objects." Bob quickly drew a code fragment on the board:

```
Contact [] contacts = new Contact[10];
while (x < 10) {    // make 10 contact objects
    contacts[x] = new Contact();
    x = x + 1;
}
// do complicated Contact list updating with contacts
```

"Tawny, I know we're tight on memory, but your spec said that we had to be able to access individual contact information for all ten allowable contacts; this was the best scheme I could cook up," said Bob. Kate was next, already imagining coconut cocktails at the party, "Bob," she said, "your solution's a bit kludgy, don't you think?" Kate smirked, "Take a look at this baby":

```
Contact contactRef;
while (x < 10) {    // make 10 contact objects
    contactRef = new Contact();
    x = x + 1;
}
// do complicated Contact list updating with contactRef
```

"I saved a bunch of reference variables worth of memory, Bob-o-rino, so put away your sunscreen," mocked Kate. "Not so fast Kate!" said Tawny, "you've saved a little memory, but Bob's coming with me."

*Why did Tawny choose Bob's method over Kate's, when Kate's used less memory?*

→ Answers on page 69.

## exercise solutions



### Exercise Solutions

#### Sharpen your pencil (from page 52)

- |   |   |
|---|---|
| 1. int x = 34.5; <input checked="" type="checkbox"/>    | 7. s = y; <input checked="" type="checkbox"/>         |
| 2. boolean boo = x; <input checked="" type="checkbox"/> | 8. byte b = 3; <input checked="" type="checkbox"/>    |
| 3. int g = 17; <input checked="" type="checkbox"/>      | 9. byte v = b; <input checked="" type="checkbox"/>    |
| 4. int y = g; <input checked="" type="checkbox"/>       | 10. short n = 12; <input checked="" type="checkbox"/> |
| 5. y = y + 10; <input checked="" type="checkbox"/>      | 11. v = n; <input checked="" type="checkbox"/>        |
| 6. short s; <input checked="" type="checkbox"/>         | 12. byte k = 128; <input checked="" type="checkbox"/> |

#### Code Magnets (from page 64)

```
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[] islands = new String[4];
        islands[0] = "Bermuda";
        islands[1] = "Fiji";
        islands[2] = "Azores";
        islands[3] = "Cozumel";
        int y = 0;
        int ref;
        while (y < 4) {
            ref = index[y];
            System.out.print("island = ");
            System.out.println(islands[ref]);
            y = y + 1;
        }
    }
}
```

```
File Edit Window Help Sunscreen
% java TestArrays
island = Fiji
island = Cozumel
island = Bermuda
island = Azores
```

#### BE the Compiler (from page 63)

**A**

```
class Books {
    String title;
    String author;
}

class BooksTestDrive {
    public static void main(String[] args) {
        Books[] myBooks = new Books[3];
        int x = 0;
        myBooks[0] = new Books(); Remember: We have to
        myBooks[1] = new Books(); actually make the Book
        myBooks[2] = new Books(); objects!
        myBooks[0].title = "The Grapes of Java";
        myBooks[1].title = "The Java Gatsby";
        myBooks[2].title = "The Java Cookbook";
        myBooks[0].author = "bob";
        myBooks[1].author = "sue";
        myBooks[2].author = "ian";
        while (x < 3) {
            System.out.print(myBooks[x].title);
            System.out.print(" by ");
            System.out.println(myBooks[x].author);
            x = x + 1;
        }
    }
}
```

**B**

```
class Hobbits {
    String name;

    public static void main(String[] args) {
        Hobbits[] h = new Hobbits[3];
        int z = -1;
        while (z < 2) { Remember: arrays start
            with element 0!
            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 + " is a ");
            System.out.println("good Hobbit name");
        }
    }
}
```



## Puzzle Solutions

### Pool Puzzle (from page 65)

```
class Triangle {
    double area;
    int height;
    int length;

    public static void main(String[] args) {
        int x = 0;
        Triangle[] ta = new Triangle[4];
        while (x < 4) {
            ta[x] = new Triangle();
            ta[x].height = (x + 1) * 2;
            ta[x].length = x + 4;
            ta[x].setArea();
            System.out.print("triangle " + x +
                ", area");
            System.out.println(" = " + ta[x].area);
            x = x + 1;
        }
        int y = x;
        x = 27;
        Triangle t5 = ta[2];
        ta[2].area = 343;
        System.out.print("y = " + y);
        System.out.println(", t5 area = " +
            t5.area);
    }

    void setArea() {
        area = (height * length) / 2;
    }
}
```

```
File Edit Window Help Bermuda
%java Triangle
triangle 0, area = 4.0
triangle 1, area = 10.0
triangle 2, area = 18.0
triangle 3, area = 28.0
y = 4, t5 area = 343.0
```

### Five-Minute Mystery (from page 67)

#### The case of the pilfered references

Tawny could see that Kate's method had a serious flaw. It's true that she didn't use as many reference variables as Bob, but there was no way to access any but the last of the Contact objects that her method created. With each trip through the loop, she was assigning a new object to the one reference variable, so the previously referenced object was abandoned on the heap—*unreachable*. Without access to nine of the ten objects created, Kate's method was useless.

(The software was a huge success, and the client gave Tawny and Bob an extra week in Hawaii. We'd like to tell you that by finishing this book you too will get stuff like that.)

### A Heap o' Trouble (from page 66)

#### Reference Variables:



hq[0]



hq[1]



hq[2]



hq[3]



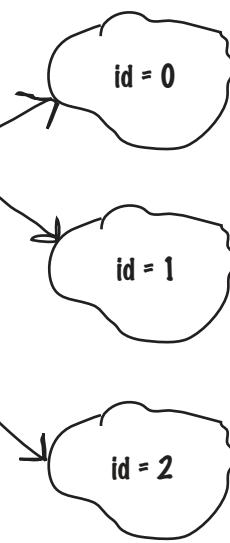
hq[4]

#### HeapQuiz Objects:

*id = 0*

*id = 1*

*id = 2*





## 4 methods use instance variables

# How Objects Behave



**State affects behavior, behavior affects state.** We know that objects have **state** and **behavior**, represented by **instance variables** and **methods**. But until now, we haven't looked at how state and behavior are *related*. We already know that each instance of a class (each object of a particular type) can have its own unique values for its instance variables. Dog A can have a *name* "Fido" and a *weight* of 70 pounds. Dog B is "Killer" and weighs 9 pounds. And if the Dog class has a method `makeNoise()`, well, don't you think a 70-pound dog barks a bit deeper than the little 9-pounder? (Assuming that annoying yippy sound can be considered a *bark*.) Fortunately, that's the whole point of an object—it has *behavior* that acts on its *state*. In other words, **methods use instance variable values**. Like, "if dog is less than 14 pounds, make yippy sound, else..." or "increase weight by 5." **Let's go change some state.**

objects have state and behavior

## Remember: a class describes what an object knows and what an object does

**A class is the blueprint for an object.** When you write a class, you're describing how the JVM should make an object of that type. You already know that every object of that type can have different *instance variable* values. But what about the methods?

**Can every object of that type have different method behavior?**

Well...*sort of*\*

Every instance of a particular class has the same methods, but the methods can *behave* differently based on the value of the instance variables.

The Song class has two instance variables, *title* and *artist*. When you call the *play()* method on an instance, it will play the song represented by the value of the *title* and *artist* instance variables for that instance. So, if you call the *play()* method on one instance, you'll hear the song "Havana" by Cabello, while another instance plays "Sing" by Travis. The method code, however, is the same.

```
void play() {  
    soundPlayer.playSound(title, artist);  
}
```

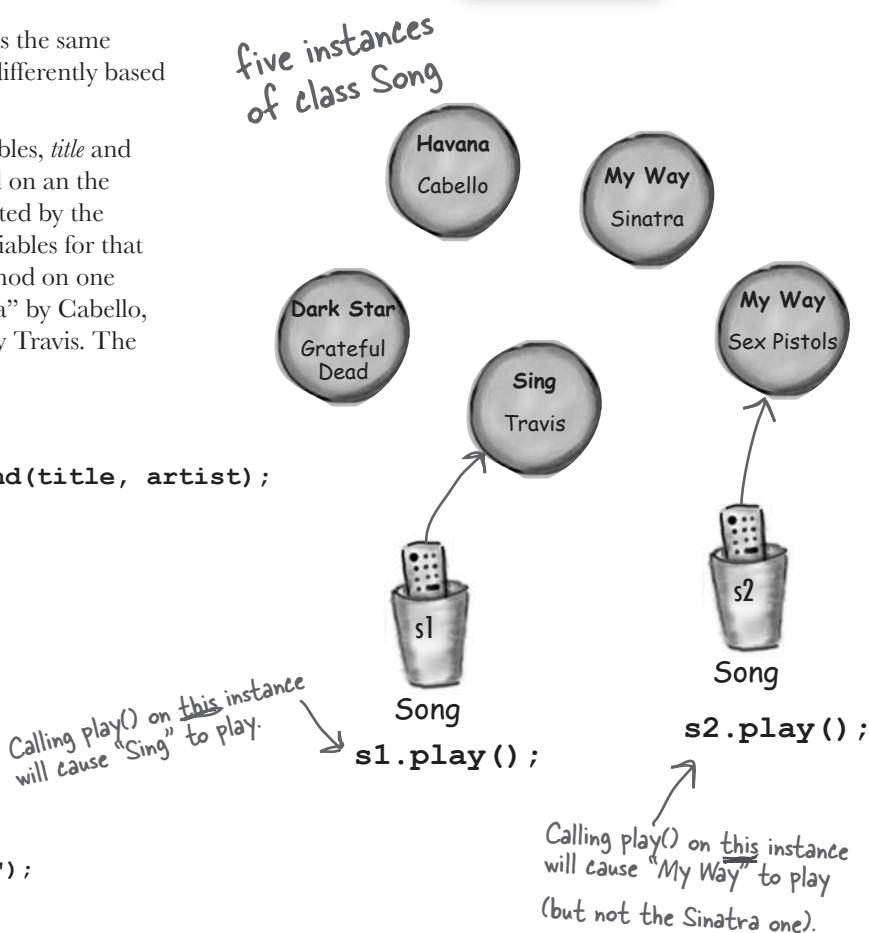
```
Song song1 = new Song();  
song1.setArtist("Travis");  
song1.setTitle("Sing");  
Song song2 = new Song();  
song2.setArtist("Sex Pistols");  
song2.setTitle("My Way");
```

Song	
title	
artist	

**instance variables (state)**

**methods (behavior)**

**knows**  
**does**



\*Yes, another stunningly clear answer!

# The size affects the bark

A small Dog's bark is different from a big Dog's bark.

The Dog class has an instance variable *size* that the *bark()* method uses to decide what kind of bark sound to make.

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

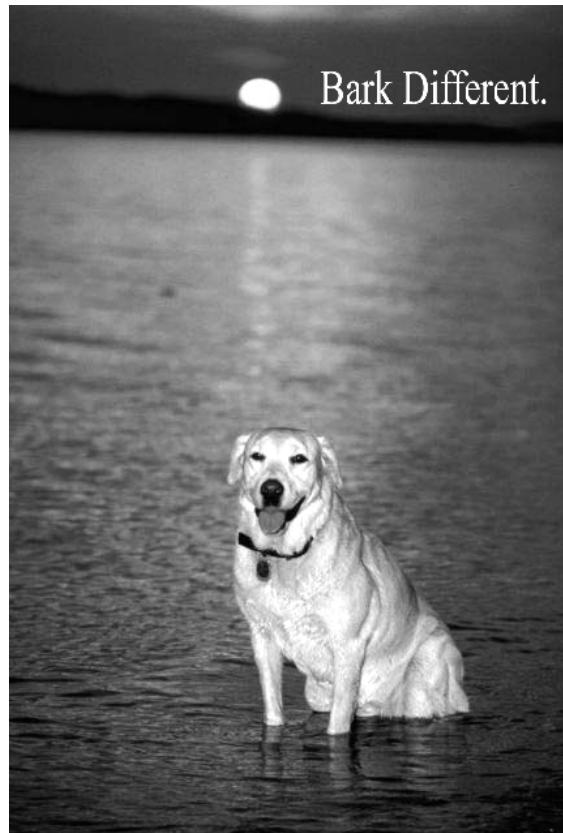
    void bark() {
        if (size > 60) {
            System.out.println("Wooof! Wooof!");
        } else if (size > 14) {
            System.out.println("Ruff! Ruff!");
        } else {
            System.out.println("Yip! Yip!");
        }
    }
}
```

---

```
class DogTestDrive {

    public static void main(String[] args) {
        Dog one = new Dog();
        one.size = 70;
        Dog two = new Dog();
        two.size = 8;
        Dog three = new Dog();
        three.size = 35;
    }
}
```

one.bark(); two.bark(); three.bark(); }	File Edit Window Help Playdead %java DogTestDrive Wooof! Wooof! Yip! Yip! Ruff! Ruff!
--	---



## You can send things to a method

Just as you expect from any programming language, you can pass values into your methods. You might, for example, want to tell a Dog object how many times to bark by calling:

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

Depending on your programming background and personal preferences, *you* might use the term *arguments* or perhaps *parameters* for the values passed into a method.

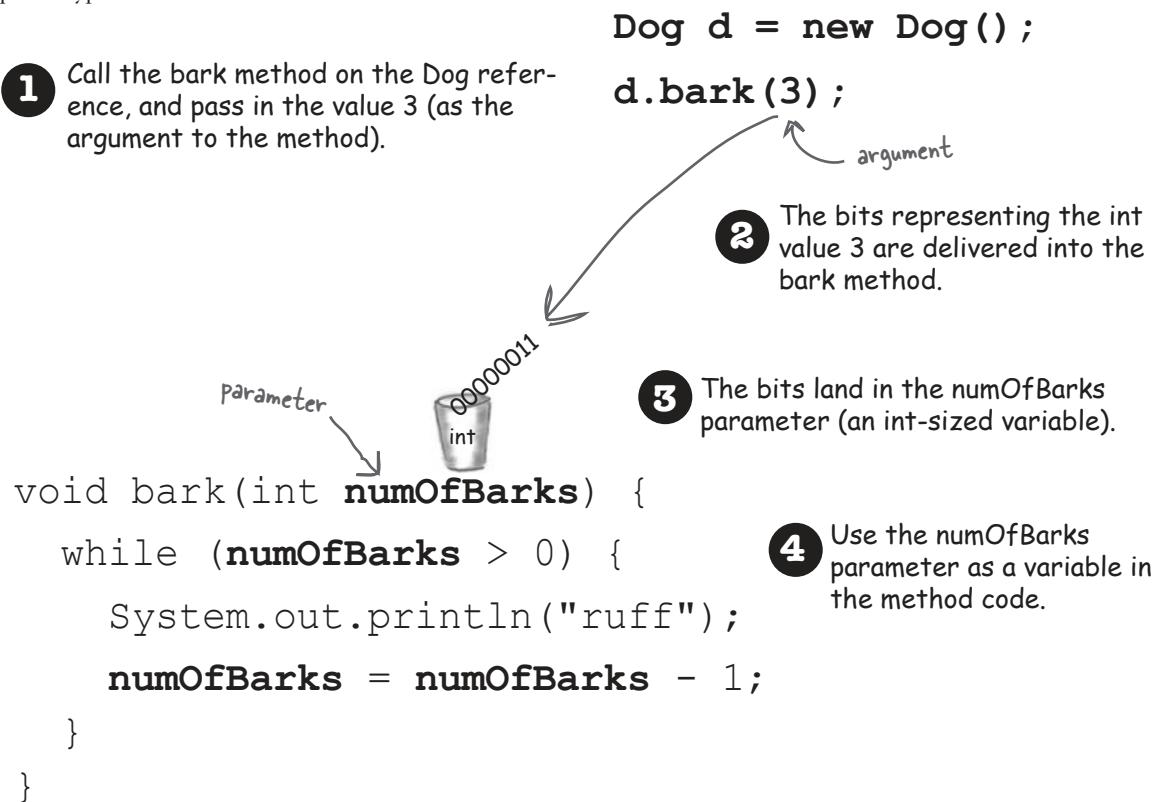
Although there *are* formal computer science distinctions that people who wear lab coats (and who will almost certainly not read this book) make, we have bigger fish to fry in this book. So *you* can call them whatever you like (arguments, donuts, hair-balls, etc.) but we're doing it like this:

**A caller passes arguments. A method takes parameters.**

Arguments are the things you pass into the methods. An **argument** (a value like 2, Foo, or a reference to a Dog) lands face-down into a...wait for it...**parameter**.

And a parameter is nothing more than a local variable. A variable with a type and a name that can be used inside the body of the method.

But here's the important part: **If a method takes a parameter, you must pass it something when you call it.** And that something must be a value of the appropriate type.



# You can get things back from a method

Methods can also *return* values. Every method is declared with a return type, but until now we've made all of our methods with a **void** return type, which means they don't give anything back.

```
void go() {  
}
```

But we can declare a method to give a specific type of value back to the caller, such as:

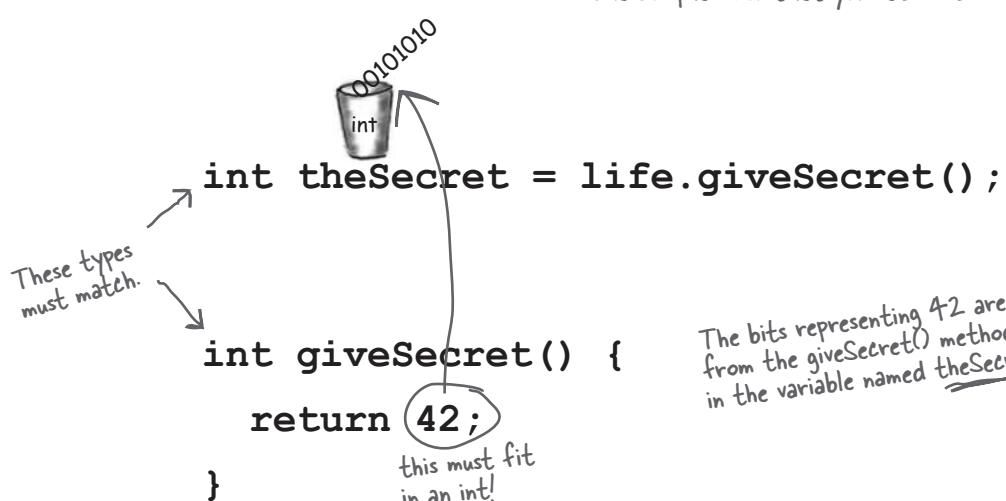
```
int giveSecret() {  
    return 42;  
}
```

If you declare a method to return a value, you *must* return a value of the declared type! (Or a value that is *compatible* with the declared type. We'll get into that more when we talk about polymorphism in Chapters 7 and 8.)

**Whatever you say  
you'll give back, you  
better give back!**



The compiler won't let you return the wrong type of thing.



The bits representing 42 are returned from the giveSecret() method, and land in the variable named theSecret.

## multiple arguments

# You can send more than one thing to a method

Methods can have multiple parameters. Separate them with commas when you declare them, and separate the arguments with commas when you pass them. Most importantly, if a method has parameters, you *must* pass arguments of the right type and order.

## Calling a two-parameter method and sending it two arguments

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

The arguments you pass land in the same order you passed them. First argument lands in the first parameter, second argument in the second parameter, and so on.

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

## You can pass variables into a method, as long as the variable type matches the parameter type

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

The values of foo and bar land in the x and y parameters. So now the bits in x are identical to the bits in foo (the bit pattern for the integer '7'), and the bits in y are identical to the bits in bar.

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

What's the value of z? It's the same result you'd get if you added foo + bar at the time you passed them into the takeTwo method.



`int x = 7;`



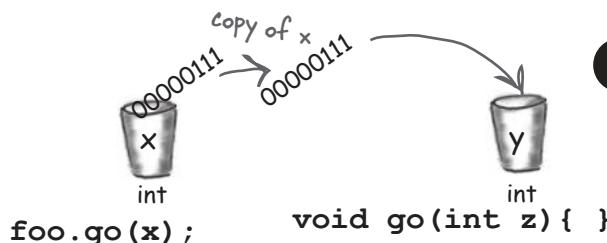
Java is pass-by-value.  
That means pass-by-copy.

`void go(int z) { }`

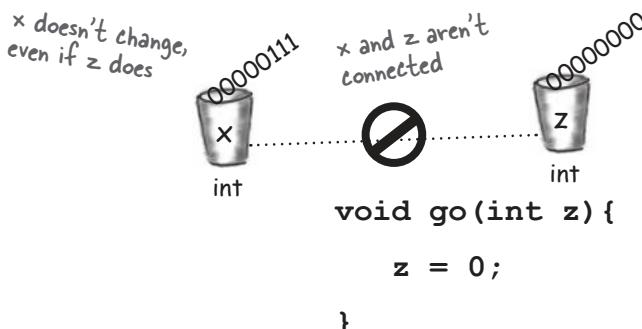


- 1** Declare an int variable and assign it the value '7'. The bit pattern for 7 goes into the variable named x.

- 2** Declare a method with an int parameter named z.

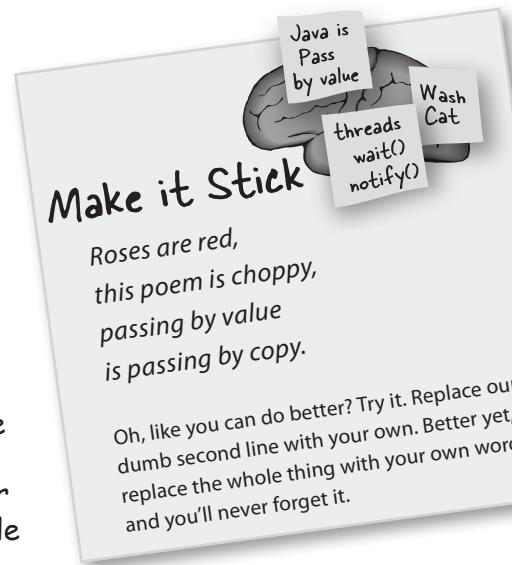


- 3** Call the go() method, passing the variable x as the argument. The bits in x are copied, and the copy lands in z.



- 4** Change the value of z inside the method. The value of x doesn't change! The argument passed to the z parameter was only a copy of x.

The method can't change the bits that were in the calling variable x.



## there are no Dumb Questions

**Q:** What happens if the argument you want to pass is an object instead of a primitive?

**A:** You'll learn more about this in later chapters, but you already know the answer. Java passes *everything* by value. **Everything**. But...*value* means *bits inside the variable*. And remember, you don't stuff objects into variables; the variable is a remote control—a *reference to an object*. So if you pass a reference to an object into a method, you're passing a *copy of the remote control*. Stay tuned, though, we'll have lots more to say about this.

**Q:** Can a method declare multiple return values? Or is there some way to return more than one value?

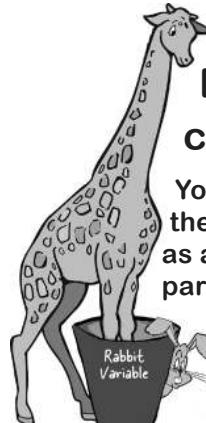
**A:** Sort of. A method can declare only one return value. BUT...if you want to return, say, three int values, then the declared return type can be an int *array*. Stuff those ints into the array, and pass it on back. It's a little more involved to return multiple values with different types; we'll be talking about that in a later chapter when we talk about ArrayList.

**Q:** Do I have to return the exact type I declared?

**A:** You can return anything that can be *implicitly* promoted to that type. So, you can pass a byte where an int is expected. The caller won't care, because the byte fits just fine into the int the caller will use for assigning the result. You must use an *explicit cast* when the declared type is *smaller* than what you're trying to return (we'll see these in Chapter 5).

**Q:** Do I have to do something with the return value of a method? Can I just ignore it?

**A:** Java doesn't require you to acknowledge a return value. You might want to call a method with a non-void return type, even though you don't care about the return value. In this case, you're calling the method for the work it does *inside* the method, rather than for what the method gives *returns*. In Java, you don't have to assign or use the return value.



## Reminder: Java cares about type!

You can't return a Giraffe when the return type is declared as a Rabbit. Same thing with parameters. You can't pass a Giraffe into a method that takes a Rabbit.

### BULLET POINTS

- Classes define what an object knows and what an object does.
- Things an object knows are its **instance variables** (state).
- Things an object does are its **methods** (behavior).
- Methods can use instance variables so that objects of the same type can behave differently.
- A method can have parameters, which means you can pass one or more values in to the method.
- The number and type of values you pass in must match the order and type of the parameters declared by the method.
- Values passed in and out of methods can be implicitly promoted to a larger type or explicitly cast to a smaller type.
- The value you pass as an argument to a method can be a literal value (2, 'c', etc.) or a variable of the declared parameter type (for example, x where x is an int variable). (There are other things you can pass as arguments, but we're not there yet.)
- A method *must* declare a return type. A void return type means the method doesn't return anything.
- If a method declares a non-void return type, it *must* return a value compatible with the declared return type.

# Cool things you can do with parameters and return types

Now that we've seen how parameters and return types work, it's time to put them to good use: let's create **Getters** and **Setters**. If you're into being all formal about it, you might prefer to call them *Accessors* and *Mutators*. But that's a waste of perfectly good syllables. Besides, Getters and Setters fits a common Java naming convention, so that's what we'll call them.

Getters and Setters let you, well, *get and set things*. Instance variable values, usually. A Getter's sole purpose in life is to send back, as a return value, the value of whatever it is that particular Getter is supposed to be Getting. And by now, it's probably no surprise that a Setter lives and breathes for the chance to take an argument value and use it to *set* the value of an instance variable.

```
class ElectricGuitar {
    String brand;
    int numOfPickups;
    boolean rockStarUsesIt;

    String getBrand() {
        return brand;
    }

    void setBrand(String aBrand) {
        brand = aBrand;
    }

    int getNumOfPickups() {
        return numOfPickups;
    }

    void setNumOfPickups(int num) {
        numOfPickups = num;
    }

    boolean getRockStarUsesIt() {
        return rockStarUsesIt;
    }

    void setRockStarUsesIt(boolean yesOrNo) {
        rockStarUsesIt = yesOrNo;
    }
}
```

ElectricGuitar
brand
numOfPickups
rockStarUsesIt
getBrand()
setBrand()
getNumOfPickups()
setNumOfPickups()
getRockStarUsesIt()
setRockStarUsesIt()

Note: Using these naming conventions means you're following a standard that you'll see in lots of Java code



# Encapsulation

## Do it or risk humiliation and ridicule.

Until this most important moment, we've been committing one of the worst OO faux pas (and we're not talking minor violation like showing up without the "B" in BYOB). No, we're talking Faux Pas with a capital "F." And "P."

Our shameful transgression?

Exposing our data!

Here we are, just humming along without a care in the world leaving our data out there for *anyone* to see and even touch.

You may have already experienced that vaguely unsettling feeling that comes with leaving your instance variables exposed.

Exposed means reachable with the dot operator, as in:

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

Think about this idea of using our remote control to make a direct change to the Cat object's size instance variable. In the hands of the wrong person, a reference variable (remote control) is quite a dangerous weapon. Because what's to prevent:

```
theCat.height = 0; ← Oh my goodness! We  
can't let this happen!
```

This would be a Bad Thing. We need to build setter methods for all the instance variables, and find a way to force other code to call the setters rather than access the data directly.



By forcing everybody to call a setter method, we can protect the cat from unacceptable size changes.

```
public void setHeight(int ht) {
```

```
    if (ht > 9) {
```

```
        height = ht;
```

```
}
```

← We put in checks to guarantee a minimum cat height.

## Hide the data

Yes, it *is* that simple to go from an implementation that's just begging for bad data to one that protects your data *and* protects your right to modify your implementation later.

OK, so how exactly do you *hide* the data? With the **public** and **private** access modifiers. You're familiar with **public**—we use it with every main method.

Here's an encapsulation *starter* rule of thumb (all standard disclaimers about rules of thumb are in effect): mark your instance variables **private** and provide **public** getters and setters for access control. When you have more design and coding savvy in Java, you will probably do things a little differently, but for now, this approach will keep you safe.

**Mark instance variables **private**.**

**Mark getters and setters **public**.**

**"Sadly, Bill forgot to encapsulate his Cat class and ended up with a flat cat."**

(overheard at the water cooler)



This week's interview:  
**An Object gets candid about encapsulation.**

**HeadFirst:** What's the big deal about encapsulation?

**Object:** OK, you know that dream where you're giving a talk to 500 people when you suddenly realize you're *naked*?

**HeadFirst:** Yeah, we've had that one. It's right up there with the one about the Pilates machine and...no, we won't go there. OK, so you feel naked. But other than being a little exposed, is there any danger?

**Object:** Is there any danger? Is there any *danger*? [starts laughing] Hey, did all you other instances hear that, "*Is there any danger?*" he asks? [falls on the floor laughing]

**HeadFirst:** What's funny about that? Seems like a reasonable question.

**Object:** OK, I'll explain it. It's [bursts out laughing again, uncontrollably]

**HeadFirst:** Can I get you anything? Water?

**Object:** Whew! Oh boy. No I'm fine, really. I'll be serious. Deep breath. OK, go on.

**HeadFirst:** So what does encapsulation protect you from?

**Object:** Encapsulation puts a force-field around my instance variables, so nobody can set them to, let's say, something *inappropriate*.

**HeadFirst:** Can you give me an example?

**Object:** Happy to. Most instance variable values are coded with certain assumptions about their boundaries. Like, think of all the things that would break if negative numbers were allowed. Number of bathrooms in an office. Velocity of an airplane. Birthdays. Barbell weight. Phone numbers. Microwave oven power.

**HeadFirst:** I see what you mean. So how does encapsulation let you set boundaries?

**Object:** By forcing other code to go through setter methods. That way, the setter method can validate the parameter and decide if it's doable. Maybe the method will reject it and do nothing, or maybe it'll throw an Exception (like if it's a null Social Security number for a credit card application), or maybe the method will round the parameter sent in to the nearest acceptable value. The point is, you can do whatever you want in the setter method, whereas you can't do *anything* if your instance variables are public.

**HeadFirst:** But sometimes I see setter methods that simply set the value without checking anything. If you have an instance variable that doesn't have a boundary, doesn't that setter method create unnecessary overhead? A performance hit?

**Object:** The point to setters (and getters, too) is that ***you can change your mind later, without breaking anybody else's code!*** Imagine if half the people in your company used your class with public instance variables, and one day you suddenly realized, "Oops—there's something I didn't plan for with that value, I'm going to have to switch to a setter method." You break everyone's code. The cool thing about encapsulation is that *you get to change your mind*. And nobody gets hurt. The performance gain from using variables directly is so minuscule and would rarely—if ever—be worth it.

# Encapsulating the GoodDog class

*Make the instance variable private.*

*Make the getter and setter methods public.*

```
class GoodDog {
    private int size;

    public int getSize() {
        return size;
    }

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

GoodDog
size
getSize()
setSize()
bark()

Even though the methods don't really add new functionality, the nice thing is that you can change your mind later. You can come back and make a method safer, faster, better.

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

```
class GoodDogTestDrive {

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

**Any place where a particular value can be used, a *method call* that returns that type can be used.**

**instead of:**

int x = 3 + 24;

**you can say:**

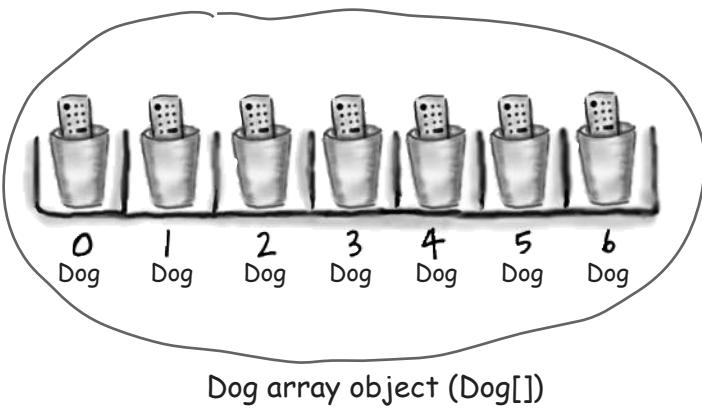
int x = 3 + one.getSize();

# How do objects in an array behave?

Just like any other object. The only difference is how you *get* to them. In other words, how you get the remote control. Let's try calling methods on Dog objects in an array.

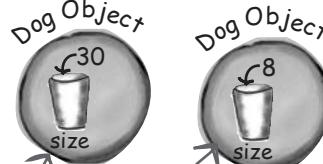
- 1 Declare and create a Dog array to hold seven Dog references.

```
Dog[] pets;
pets = new Dog[7];
```



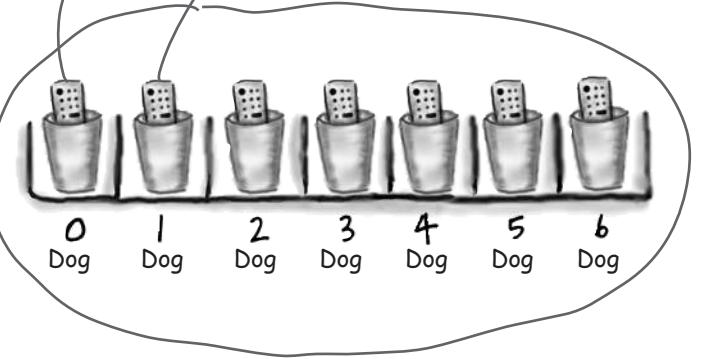
- 2 Create two new Dog objects, and assign them to the first two array elements.

```
pets[0] = new Dog();
pets[1] = new Dog();
```



- 3 Call methods on the two Dog objects.

```
pets[0].setSize(30);
int x = pets[0].getSize();
pets[1].setSize(8);
```



# Declaring and initializing instance variables

You already know that a variable declaration needs at least a name and a type:

```
int size;
String name;
```

And you know that you can initialize (assign a value to) the variable at the same time:

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

But when you don't initialize an instance variable, what happens when you call a getter method? In other words, what is the *value* of an instance variable *before* you initialize it?

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

    public int getSize() {
        return size;
    }

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

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

```
File Edit Window Help CallVet
% java PoorDogTestDrive
Dog size is 0
Dog name is null
```

**Instance variables always get a default value. If you don't explicitly assign a value to an instance variable or you don't call a setter method, the instance variable still has a value!**

integers	0
floating points	0.0
booleans	false
references	null

What do you think? Will this even compile?  
 You don't have to initialize instance variables, because they always have a default value. Number primitives (including char) get 0, booleans get false, and object reference variables get null.  
 (Remember, null just means a remote control that isn't controlling / programmed to anything. A reference, but no actual object.)

# The difference between instance and local variables

- 1** **Instance** variables are declared inside a class but not within a method.

```
class Horse {
    private double height = 15.2;
    private String breed;
    // more code...
}
```

- 2** **Local** variables are declared within a method.

```
class AddThing {
    int a;           } INSTANCE variables
    int b = 12;

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

- 3** **Local** variables MUST be initialized before use!

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

← *Won't compile!! You can declare x without a value, but as soon as you try to USE it, the compiler freaks out.*

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

**Local variables do NOT get a default value! The compiler complains if you try to use a local variable before the variable is initialized.**

there are no  
**Dumb Questions**

**Q:** What about method parameters? How do the rules about local variables apply to them?

**A:** Method parameters are virtually the same as local variables—they’re declared *inside* the method (well, technically they’re declared in the *argument list* of the method rather than within the *body* of the method, but they’re still local variables as opposed to instance variables). But method parameters will never be uninitialized, so you’ll never get a compiler error telling you that a parameter variable might not have been initialized.

Instead, the compiler will give you an error if you try to invoke a method without giving the arguments that the method needs. So parameters are *always* initialized, because the compiler guarantees that methods are always called with arguments that match the parameters. The arguments are assigned (automatically) to the parameters.

# Comparing variables (primitives or references)

Sometimes you want to know if two *primitives* are the same; for example, you might want to check an int result with some expected integer value. That's easy enough: just use the `==` operator. Sometimes you want to know if two reference variables refer to a single object on the heap; for example, is this Dog object exactly the same Dog object I started with? Easy as well: just use the `==` operator. But sometimes you want to know if two *objects* are equal. And for that, you need the `.equals()` method.

The idea of equality for objects depends on the type of object. For example, if two different String objects have the same characters (say, "my name"), they are meaningfully equivalent, regardless of whether they are two distinct objects on the heap. But what about a Dog? Do you want to treat two Dogs as being equal if they happen to have the same size and weight? Probably not. So whether two different objects should be treated as equal depends on what makes sense for that particular object type. We'll explore the notion of object equality again in later chapters, but for now, we need to understand that the `==` operator is used *only* to compare the bits in two variables. What those bits represent doesn't matter. The bits are either the same, or they're not.

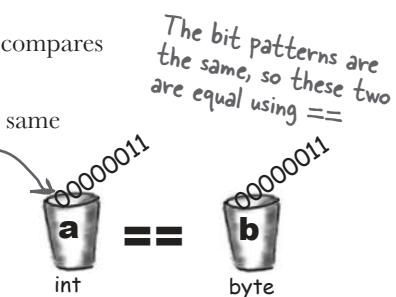
## To compare two primitives, use the `==` operator

The `==` operator can be used to compare two variables of any kind, and it simply compares the bits.

if (`a == b`) {...} looks at the bits in `a` and `b` and returns true if the bit pattern is the same (although all the extra zeros on the left end don't matter).

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

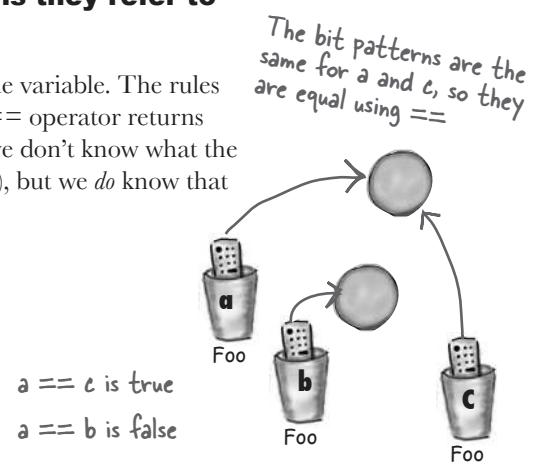
(There are more zeros on  
the left side of the int,  
but we don't care about  
that here)

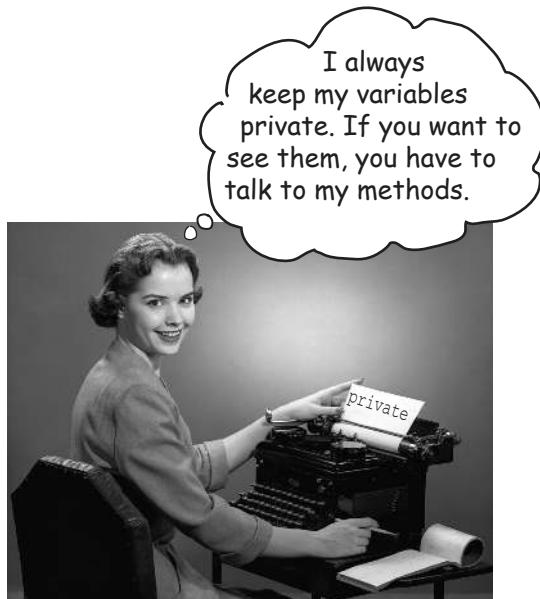


## To see if two references are the same (which means they refer to the same object on the heap) use the `==` operator

Remember, the `==` operator cares only about the pattern of bits in the variable. The rules are the same whether the variable is a reference or primitive. So the `==` operator returns true if two reference variables refer to the same object! In that case, we don't know what the bit pattern is (because it's dependent on the JVM and hidden from us), but we *do* know that whatever it looks like, *it will be the same for two references to a single object*.

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



**BULLET POINTS**

- Encapsulation gives you control over who changes the data in your class and how.
- Make an instance variable *private* so it can't be changed by accessing the variable directly.
- Create a public mutator method, e.g., a setter, to control how other code interacts with your data. For example, you can add validation code inside a setter to make sure the value isn't changed to something invalid.
- *Instance* variables are assigned values by default, even if you don't set them explicitly.
- *Local* variables, e.g., variables inside methods, are not assigned a value by default. You always need to initialize them.
- Use == to check if two primitives are the same value.
- Use == to check if two references are the same, i.e., two object variables are actually the same object.
- Use .equals() to see if two objects are equivalent (but not necessarily the same object), e.g., to check if two String values contain the same characters.

**Sharpen your pencil****What's legal?**

Given the method below, which of the method calls listed on the right are legal?

Put a checkmark next to the ones that are legal. (Some statements are there to assign values used in the method calls.)

```
int calcArea(int height, int width) {
    return height * width;
}
```



```
int a = calcArea(7, 12);
short c = 7;
calcArea(c, 15);

int d = calcArea(57);
calcArea(2, 3);

long t = 42;
int f = calcArea(t, 17);

int g = calcArea();

calcArea();

byte h = calcArea(4, 20);

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

→ Answers on page 93.



## BE the Compiler



Each of the Java files on this page represents a complete source file. Your job is to play compiler and determine whether each of these files will compile. If they won't compile, how would you fix them, and if they do compile, what would be their output?

**A**

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

**B**

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

→ Answers on page 93.



# Who Am I?



A class can have any number of these.

---



---

A method can have only one of these.

---



---

This can be implicitly promoted.

---



---

I prefer my instance variables private.

---



---

It really means “make a copy.”

---



---

Only setters should update these.

---



---

A method can have many of these.

---



---

I return something by definition.

---



---

I shouldn't be used with instance variables.

---



---

I can have many arguments.

---



---

By definition, I take one argument.

---



---

These help create encapsulation.

---



---

I always fly solo.

---



---

→ Answers on page 93.

## puzzle: Mixed Messages



A short Java program is listed to your right. Two blocks of the program are missing. Your challenge is to **match the candidate blocks of code** (below) with the output that you'd see if the blocks were inserted.

Not all the lines of output will be used, and some of the lines of output might be used more than once. Draw lines connecting the candidate blocks of code with their matching command-line output.

### Candidates:

i < 9

index < 5

i < 20

index < 5

i < 7

index < 7

i < 19

index < 1

### Possible output:

14 7

9 5

19 1

14 1

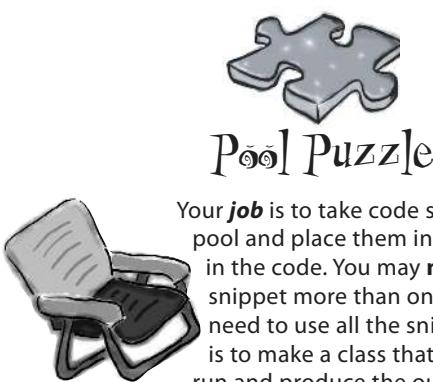
25 1

7 7

20 1

20 5

```
public class Mix4 {  
    int counter = 0;  
  
    public static void main(String[] args) {  
        int count = 0;  
        Mix4[] mixes = new Mix4[20];  
        int i = 0;  
        while ( [ ] ) {  
            mixes[i] = new Mix4();  
            mixes[i].counter = mixes[i].counter + 1;  
            count = count + 1;  
            count = count + mixes[i].maybeNew(i);  
            i = i + 1;  
        }  
        System.out.println(count + " " +  
                           mixes[1].counter);  
    }  
  
    public int maybeNew(int index) {  
        if ( [ ] ) {  
            Mix4 mix = new Mix4();  
            mix.counter = mix.counter + 1;  
            return 1;  
        }  
        return 0;  
    }  
}
```



Your **job** is to take code snippets from the pool and place them into the blank lines in the code. You may **not** use the same snippet more than once, and you won't need to use all the snippets. Your **goal** is to make a class that will compile and run and produce the output listed.

→ Answers on page 94.

### Output

```
File Edit Window Help BellyFlop
%java Puzzle4
result 543345
```

**Note:** Each snippet from the pool can be used only once!

```
Puzzle4 [] values = new Puzzle4[6];
Value [] values = new Value[6];
Value [] values = new Puzzle4[6];

intValue = i;    values[i].doStuff(i);
values.intValue = i;    values[i].intValue = i;
values[i].intValue = number;    factor
                                public
                                private
doStuff(i);    values.doStuff(i);
values.doStuff(i);    values[i].doStuff(factor);

intValue + factor;    Puzzle4    int
intValue * (2 + factor);    Value    short
intValue * (5 - factor);    Value()    short
intValue * factor;    values [i] = new Value(i);
values [i] = new Value();    values [i] = new Value();
values [i] = new Value();    values = new Value();
```

```
public class Puzzle4 {
    public static void main(String [] args) {

        int number = 1;
        int i = 0;
        while (i < 6) {

            _____
            _____
            number = number * 10;
            _____
        }

        int result = 0;
        i = 6;
        while (i > 0) {

            result = result + _____
        }
        System.out.println("result " + result);
    }
}

class _____ {
    int intValue;
    _____ doStuff(int _____) {
        if (intValue > 100) {
            return _____
        } else {
            return _____
        }
    }
}
```



## Fast Times in Stim-City

When Buchanan roughly grabbed Jai's arm from behind, Jai froze. Jai knew that Buchanan was as stupid as he was ugly and he didn't want to spook the big guy. Buchanan ordered Jai into his boss's office, but Jai'd done nothing wrong (lately), so he figured a little chat with Buchanan's boss Leveler couldn't be too bad. He'd been moving lots of neural-stimmers in the west side lately, and he figured Leveler would be pleased. Black market stimmers weren't the best money pump around, but they were pretty harmless. Most of the stim-junkies he'd seen tapped out after a while and got back to life, maybe just a little less focused than before.

# Five-Minute Mystery

Leveler's "office" was a skungy-looking skimmer, but once Buchanan shoved him in, Jai could see that it'd been modified to provide all the extra speed and armor that a local boss like Leveler could hope for. "Jai my boy," hissed Leveler, "pleasure to see you again." "Likewise I'm sure..." said Jai, sensing the malice behind Leveler's greeting, "We should be square Leveler, have I missed something?" "Ha! You're making it look pretty good, Jai. Your volume is up, but I've been experiencing, shall we say, a little 'breach' lately," said Leveler.



Jai winced involuntarily; he'd been a top drawer jack-hacker in his day. Anytime someone figured out how to break a street-jack's security, unwanted attention turned toward Jai. "No way it's me man," said Jai, "not worth the downside. I'm retired from hacking, I just move my stuff and mind my own business." "Yeah, yeah," laughed Leveler, "I'm sure you're clean on this one, but I'll be losing big margins until this new jack-hacker is shut out!" "Well, best of luck, Leveler. Maybe you could just drop me here and I'll go move a few more 'units' for you before I wrap up today," said Jai.

"I'm afraid it's not that easy, Jai. Buchanan here tells me that word is you're current on Java NE 37.3.2," insinuated Leveler. "Neural edition? Sure, I play around a bit, so what?" Jai responded, feeling a little queasy. "Neural edition's how I let the stim-junkies know where the next drop will be," explained Leveler. "Trouble is, some stim-junkie's stayed straight long enough to figure out how to hack into my Warehousing database." "I need a quick thinker like yourself, Jai, to take a look at my StimDrop Java NE class; methods, instance variables, the whole enchilada, and figure out how they're getting in. It should..." "HEY!" exclaimed Buchanan, "I don't want no scum hacker like Jai nosin' around my code!" "Easy big guy," Jai saw his chance, "I'm sure you did a top rate job with your access modi..." "Don't tell me, bit twiddler!" shouted Buchanan, "I left all of those junkie-level methods public so they could access the drop site data, but I marked all the critical Warehousing methods private. Nobody on the outside can access those methods, buddy, nobody!"

"I think I can spot your leak, Leveler. What say we drop Buchanan here off at the corner and take a cruise around the block?" suggested Jai. Buchanan clenched his fists and started toward Jai, but Leveler's stunner was already on Buchanan's neck. "Let it go, Buchanan," sneered Leveler, "Keep your hands where I can see them and step outside. I think Jai and I have some plans to make."

***What did Jai suspect?***

***Will he get out of Leveler's skimmer with all his bones intact?***

—————> **Answers on page 94.**



## Exercise Solutions

### Sharpen your pencil (from page 87)

```

int a = calcArea(7, 12);
short c = 7;
calcArea(c, 15);      ✓

int d = calcArea(57);

calcArea(2, 3);      ✓

long t = 42;
int f = calcArea(t, 17);

int g = calcArea();

calcArea();

byte h = calcArea(4, 20);

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

```

### Who Am I? (from page 89)

- A class can have any number of these.
- A method can have only one of these.
- This can be implicitly promoted.
- I prefer my instance variables private.
- It really means “make a copy.”
- Only setters should update these.
- A method can have many of these.
- I return something by definition.
- I shouldn't be used with instance variables
- I can have many arguments.
- By definition, I take one argument.
- These help create encapsulation.
- I always fly solo.

**A**

Class 'XCopy' compiles and runs as it stands! The output is: '42 84'. Remember, Java is pass by value, (which means pass by copy), and the variable 'orig' is not changed by the go( ) method.

### BE the Compiler (from page 88)

**B**

```

class Clock {
    String time;

    void setTime(String t) {
        time = t;
    }

    String getTime() {
        return time;
    }
}

Note: 'Getter' methods have a
return type by definition.

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

```

- instance variables, getter, setter, method
- return
- return, argument
- encapsulation
- pass by value
- instance variables
- argument
- getter
- public
- method
- setter
- getter, setter, public, private
- return

## Puzzle Solutions

### Pool Puzzle (from page 91)

```
public class Puzzle4 {
    public static void main(String[] args) {
        Value[] values = new Value[6];
        int number = 1;
        int i = 0;
        while (i < 6) {
            values[i] = new Value();
            values[i].intValue = number;
            number = number * 10;
            i = i + 1;
        }

        int result = 0;
        i = 6;
        while (i > 0) {
            i = i - 1;
            result = result + values[i].doStuff(i);
        }
        System.out.println("result " + result);
    }
}

class Value {
    int intValue;

    public int doStuff(int factor) {
        if (intValue > 100) {
            return intValue * factor;
        } else {
            return intValue * (5 - factor);
        }
    }
}
```

**Output**

File Edit Window Help BellyFlop
% java Puzzle4
result 543345

### Five-Minute Mystery (from page 92)

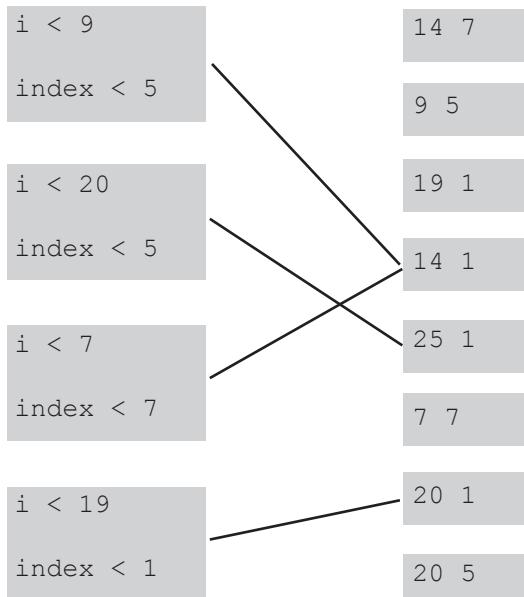
#### What did Jai suspect?

Jai knew that Buchanan wasn't the sharpest pencil in the box. When Jai heard Buchanan talk about his code, Buchanan never mentioned his instance variables. Jai suspected that while Buchanan did in fact handle his methods correctly, he failed to mark his instance variables `private`. That slip-up could have easily cost Leveler thousands.

### Mixed Messages (from page 90)

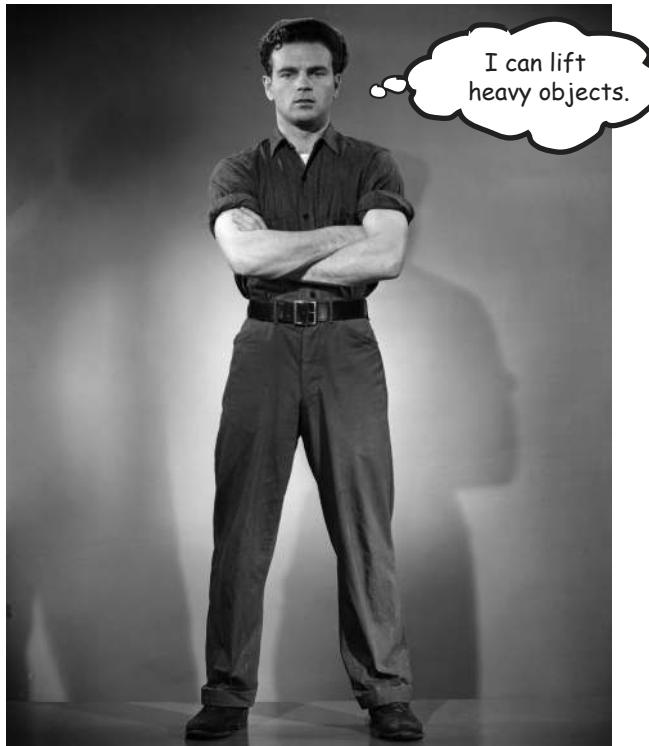
#### Candidates:

#### Possible output:



## 5 writing a program

# Extra-Strength Methods



**Let's put some muscle in our methods.** We dabbled with variables, played with a few objects, and wrote a little code. But we were weak. We need more tools. Like **operators**. We need more operators so we can do something a little more interesting than, say, *bark*. And **loops**. We need loops, but what's with the wimpy *while* loops? We need **for** loops if we're really serious. Might be useful to **generate random numbers**. Better learn that too. And why don't we learn it all by *building* something real, to see what it's like to write (and test) a program from scratch. **Maybe a game**, like Battleships. That's a heavy-lifting task, so it'll take two chapters to finish. We'll build a simple version in this chapter and then build a more powerful deluxe version in Chapter 6, *Using the Java Library*.

# Let's build a Battleship-style game: "Sink a Startup"

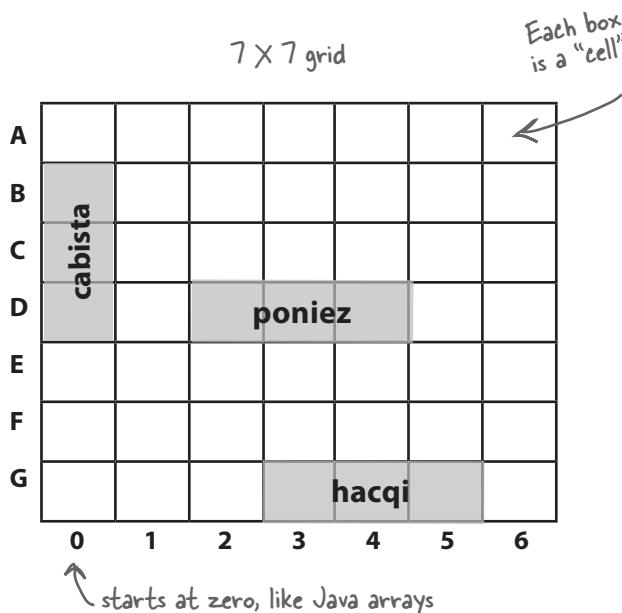
It's you against the computer, but unlike the real Battleship game, in this one you don't place any ships of your own. Instead, your job is to sink the computer's ships in the fewest number of guesses.

Oh, and we aren't sinking ships. We're killing ill-advised, Silicon Valley Startups (thus establishing business relevancy so you can expense the cost of this book).

**Goal:** Sink all of the computer's Startups in the fewest number of guesses. You're given a rating or level, based on how well you perform.

**Setup:** When the game program is launched, the computer places three Startups on a **virtual 7 x 7 grid**. When that's complete, the game asks for your first guess.

**How you play:** We haven't learned to build a GUI yet, so this version works at the command line. The computer will prompt you to enter a guess (a cell) that you'll type at the command line as "A3," "C5," etc.). In response to your guess, you'll see a result at the command-line, either "hit," "miss," or "You sunk poniez" (or whatever the lucky Startup of the day is). When you've sent all three Startups to that big 404 in the sky, the game ends by printing out your rating.



**You're going to build the Sink a Startup game, with a 7 x 7 grid and three Startups. Each Startup takes up three cells.**

part of a game interaction

```
File Edit Window Help Sell
%java StartupBust
Enter a guess  A3
miss
Enter a guess  B2
miss
Enter a guess  C4
miss
Enter a guess  D2
hit
Enter a guess  D3
hit
Enter a guess  D4
Ouch! You sunk poniez  :(
kill
Enter a guess  G3
hit
Enter a guess  G4
hit
Enter a guess  G5
Ouch! You sunk hacqi  :(
All Startups are dead! Your stock is now worthless
Took you long enough. 62 guesses.
```

# First, a high-level design

We know we'll need classes and methods, but what should they be? To answer that, we need more information about what the game should do.

First, we need to figure out the general flow of the game. Here's the basic idea:

## 1 User starts the game.

- A** Game creates three Startups
- B** Game places the three Startups onto a virtual grid

## 2 Game play begins.

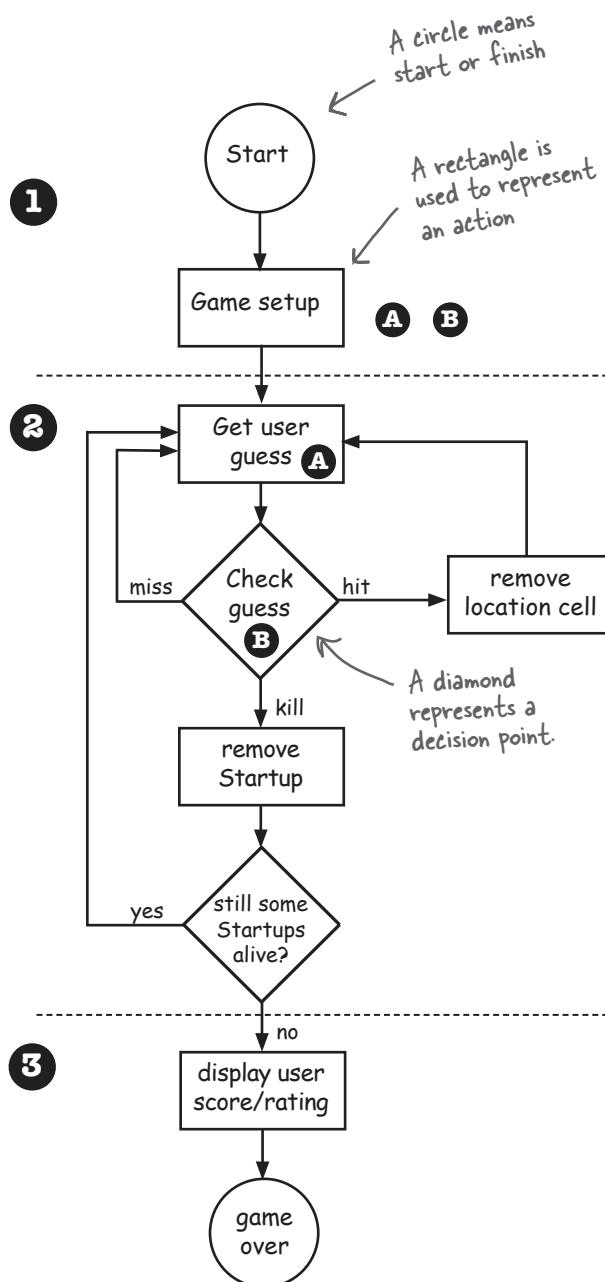
Repeat the following until there are no more Startups:

- A** Prompt user for a guess ("A2," "C0," etc.)
- B** Check the user guess against all Startups to look for a hit, miss, or kill. Take appropriate action: if a hit, delete cell (A2, D4, etc.). If a kill, delete Startup.

## 3 Game finishes.

Give the user a rating based on the number of guesses.

Now we have an idea of the kinds of things the program needs to do. The next step is figuring out what kind of **objects** we'll need to do the work. Remember, think like Brad rather than Laura (who we met in Chapter 2, *A Trip to Objectville*); focus first on the **things** in the program rather than the **procedures**.



Whoa. A real flow chart.

## The “Simple Startup Game” a gentler introduction

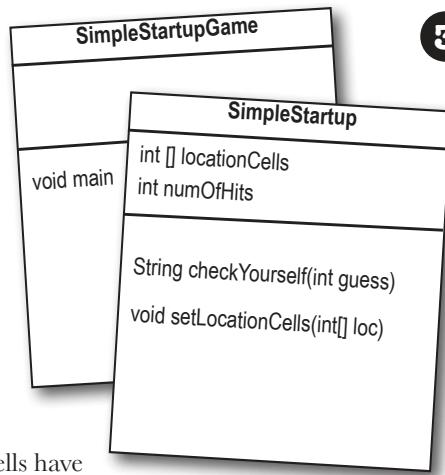
It looks like we’re gonna need at least two classes, a Game class and a Startup class. But before we build the full-monty **Sink a Startup** game, we’ll start with a stripped-down, simplified version, **Simple Startup Game**. We’ll build the simple version in *this* chapter, followed by the deluxe version that we build in the *next* chapter.

Everything is simpler in this game. Instead of a 2-D grid, we hide the Startup in just a single *row*. And instead of *three* Startups, we use *one*.

The goal is the same, though, so the game still needs to make a Startup instance, assign it a location somewhere in the row, get user input, and when all of the Startup’s cells have been hit, the game is over. This simplified version of the game gives us a big head start on building the full game. If we can get this small one working, we can scale it up to the more complex one later.

In this simple version, the game class has no instance variables, and all the game code is in the main() method. In other words, when the program is launched and main() begins to run, it will make the one and only Startup instance, pick a location for it (three consecutive cells on the single virtual seven-cell row), ask the user for a guess, check the guess, and repeat until all three cells have been hit.

Keep in mind that the virtual row is...*virtual*. In other words, it doesn’t exist anywhere in the program. As long as both the game and the user know that the Startup is hidden in three consecutive cells out of a possible seven (starting at zero), the row itself doesn’t have to be represented in code. You might be tempted to build an array of seven ints and then assign the Startup to three of the seven elements in the array, but you don’t need to. All we need is an array that holds just the three cells the Startup occupies.



1

**Game starts** and creates ONE Startup and gives it a location on three cells in the single row of seven cells.

Instead of “A2,” “C4,” and so on, the locations are just integers (for example: 1,2,3 are the cell locations in this picture):



2

**Game play begins.** Prompt user for a guess; then check to see if it hit any of the Startup’s three cells. If a hit, increment the numOfHits variable.

3

**Game finishes** when all three cells have been hit (the numOfHits variable value is 3), and the user is told how many guesses it took to sink the Startup.

### A complete game interaction

```

File Edit Window Help Destroy
%java SimpleStartupGame
enter a number 2
hit
enter a number 3
hit
enter a number 4
miss
enter a number 1
kill
You took 4 guesses

```

# Developing a Class

As a programmer, you probably have a methodology/process/approach to writing code. Well, so do we. Our sequence is designed to help you see (and learn) what we're thinking as we work through coding a class. It isn't necessarily the way we (or *you*) write code in the Real World. In the Real World, of course, you'll follow the approach your personal preferences, project, or employer dictate. We, however, can do pretty much whatever we want. And when we create a Java class as a "learning experience," we usually do it like this:

- Figure out what the class is supposed to *do*.
- List the **instance variables and methods**.
- Write **prep code** for the methods. (You'll see this in just a moment.)
- Write **test code** for the methods.
- Implement** the class.
- Test** the methods.
- Debug and reimplement** as needed.
- Express gratitude that we don't have to test our so-called *learning experience* app on actual live users.



Flex those dendrites.

**How would you decide which class or classes to build first, when you're writing a program? Assuming that all but the tiniest programs need more than one class (if you're following good OO principles and not having one class do many different jobs), where do you start?**

The three things we'll write for each class:

**prep code    test code    real code**

This bar is displayed on the next set of pages to tell you which part you're working on. For example, if you see this picture at the top of a page, it means you're working on prep code for the SimpleStartup class.

SimpleStartup class

**prep code    test code    real code**

## prep code

A form of pseudocode, to help you focus on the logic without stressing about syntax.

## test code

A class or methods that will test the real code and validate that it's doing the right thing.

## real code

The actual implementation of the class. This is where we write real Java code.

## To Do:

### SimpleStartup class

- write prep code
- write test code
- write final Java code

### SimpleStartupGame class

- write prep code
- write test code [not needed]
- write final Java code

## SimpleStartup class

prep code test code real code

SimpleStartup
int [] locationCells
int numOfHits
String checkYourself(int guess)
void setLocationCells(int[] loc)

You'll get the idea of how prep code (our version of pseudocode) works as you read through this example. It's sort of halfway between real Java code and a plain English description of the class. Most prep code includes three parts: instance variable declarations, method declarations, method logic. The most important part of prep code is the method logic, because it defines *what* has to happen, which we later translate into *how* when we actually write the method code.

**DECLARE** an *int array* to hold the location cells. Call it *locationCells*.

**DECLARE** an *int* to hold the number of hits. Call it *numOfHits* and **SET** it to 0.

---

**DECLARE** a *checkYourself()* method that takes a *int* for the user's guess (1, 3, etc.), checks it, and returns a result representing a "hit," "miss," or "kill."

**DECLARE** a *setLocationCells()* setter method that takes an *int array* (which has the three cell locations as *ints* (2, 3, 4, etc.)).

---

**METHOD:** *String checkYourself(int userGuess)*

**GET** the user guess as an *int* parameter

— **REPEAT** with each of the location cells in the *int* array

    // **COMPARE** the user guess to the location cell

    — **IF** the user guess matches

**INCREMENT** the number of hits

        // **FIND OUT** if it was the last location cell:

        — **IF** number of hits is 3, **RETURN** "kill" as the result

        — **ELSE** it was not a kill, so **RETURN** "hit"

    — **END IF**

    — **ELSE** the user guess did not match, so **RETURN** "miss"

— **END IF**

— **END REPEAT**

END METHOD

**METHOD:** *void setLocationCells(int[] cellLocations)*

**GET** the cell locations as an *int array* parameter

**ASSIGN** the cell locations parameter to the cell locations instance variable

END METHOD

prep code test code real code

## Writing the method implementations

**Let's write the real method code now and get this puppy working.**

Before we start coding the methods, though, let's back up and write some code to *test* the methods. That's right, we're writing the test code *before* there's anything to test!

The concept of writing the test code first is one of the practices of Test-Driven Development (TDD), and it can make it easier (and faster) for you to write your code. We're not necessarily saying you should use TDD, but we do like the part about writing tests first. And TDD just *sounds* cool.



## Test-Driven Development (TDD)

Back in 1999, Extreme Programming (XP) was a newcomer to the software development methodology world. One of the central ideas in XP was to write test code before writing the actual code. Since then, the idea of writing test code first has spun off of XP and become the core of a newer, more popular subset of XP called TDD. (Yes, yes, we know we've just grossly oversimplified this, please cut us a little slack here.)

TDD is a **LARGE** topic, and we're only going to scratch the surface in this book. But we hope that the way we're going about developing the "Sink a Startup" game gives you some sense of TDD.

Check out *Test Driven Development: By Example* by Kent Beck if you want to learn more about how TDD works.

Here is a partial list of key ideas in TDD:

- Write the test code first.
- Develop in iteration cycles.
- Keep it (the code) simple.
- Refactor (improve the code) whenever and wherever you notice the opportunity.
- Don't release anything until it passes all the tests.
- Don't put in anything that's not in the spec (no matter how tempted you are to put in functionality "for the future").
- No killer schedules; work regular hours.

# Writing test code for the SimpleStartup class

We need to write test code that can make a SimpleStartup object and run its methods. For the SimpleStartup class, we really care about only the *checkYourself()* method, although we *will* have to implement the *setLocationCells()* method in order to get the *checkYourself()* method to run correctly.

Take a good look at the prep code below for the *checkYourself()* method (the *setLocationCells()* method is a no-brainer setter method, so we're not worried about it, but in a “real” application we might want a more robust “setter” method, which we *would* want to test).

Then ask yourself, “If the *checkYourself()* method were implemented, what test code could I write that would prove to me the method is working correctly?”

## Based on this prep code:

```
METHOD String checkYourself(int userGuess)
  GET the user guess as an int parameter
  REPEAT with each of the location cells in the int array
    // COMPARE the user guess to the location cell
    IF the user guess matches
      INCREMENT the number of hits
      // FIND OUT if it was the last location cell:
      IF number of hits is 3, RETURN "Kill" as the result
      ELSE it was not a kill, so RETURN "Hit"
      END IF
      ELSE the user guess did not match, so RETURN "Miss"
    END IF
  END REPEAT
END METHOD
```

## Here's what we should test:

1. Instantiate a SimpleStartup object.
2. Assign it a location (an array of 3 ints, like {2, 3, 4}).
3. Create an int to represent a user guess (2, 0, etc.).
4. Invoke the *checkYourself()* method passing it the fake user guess.
5. Print out the result to see if it's correct (“passed” or “failed”).

## there are no Dumb Questions

**Q:** Maybe I'm missing something here, but how exactly do you run a test on something that doesn't yet exist?

**A:** You don't. We never said you start by *running* the test; you start by *writing* the test. At the time you write the test code, you won't have anything to run it against, so you probably won't be able to compile it until you write "stub" code that can compile, but that will always cause the test to fail (like, return null).

**Q:** Then I still don't see the point. Why not wait until the code is written, and then whip out the test code?

**A:** The act of thinking through (and writing) the test code helps clarify your thoughts about what the method itself needs to do.

As soon as your implementation code is done, you already have test code just waiting to validate it. Besides, you *know* if you don't do it now, you'll *never* do it. There's always something more interesting to do.

Ideally, write a little test code, then write *only* the implementation code you need in order to pass that test. Then write a little *more* test code and write *only* the new implementation code needed to pass *that* new test. At each test iteration, you run *all* the previously written tests to prove that your latest code additions don't break previously tested code.

## Test code for the SimpleStartup class

```
public class SimpleStartupTestDrive {
    public static void main(String[] args) {
        SimpleStartup dot = new SimpleStartup(); ← Instantiate a SimpleStartup object.

        int[] locations = {2, 3, 4}; ← Make an int array for the location of the Startup (3 consecutive ints out of a possible 7).

        dot.setLocationCells(locations); ← Invoke the setter method on the Startup.

        int userGuess = 2; ← Make a fake user guess.

        String result = dot.checkYourself(userGuess); ← Invoke the checkYourself() method on the Startup object, and pass it the fake guess.

        String testResult = "failed";
        if (result.equals("hit")) {
            testResult = "passed"; ← If the fake guess (2) gives back a "hit", it's working.
        }

        System.out.println(testResult); ← Print out the test result ("passed" or "failed").
    }
}
```



### Sharpen your pencil

→ Yours to solve.

In the next couple of pages we implement the SimpleStartup class, and then later we return to the test class. Looking at our test code above, what else should be added? What are we *not* testing in this code that we *should* be testing for? Write your ideas (or lines of code) below:

## The checkYourself() method

There isn't a perfect mapping from prep code to Java code; you'll see a few adjustments. The prep code gave us a much better idea of *what* the code needs to do, and now we have to figure out the Java code that can do the *how*.

In the back of your mind, be thinking about parts of this code you might want (or need) to improve. The numbers ① are for things (syntax and language features) you haven't seen yet. They're explained on the opposite page.

**GET** the user guess

**REPEAT** with each cell in the int array

**IF** the user guess matches

**INCREMENT** the number of hits

**// FIND OUT** if it was the last cell

**IF** number of hits is 3,

**RETURN** "kill" as the result

**ELSE** it was not a kill, so

**RETURN** "hit"

**ELSE**

**RETURN** "miss"

```

public String checkYourself(int guess) {
    String result = "miss"; ← Make a variable to hold the result we'll
                           return. put "miss" in as the default
                           (i.e., we assume a "miss")

    ① for (int cell : locationCells) { ← Repeat with each cell in the locationCells
                                         array (each cell location of the object)
        if (guess == cell) { ← Compare the user guess to this
                             element (cell) in the array
            result = "hit"; ← We got a hit!
            ② numOfHits++; ← Get out of the loop, no need
                                to test the other cells
            ③ break; ← to test the other cells
        } // end if
    } // end for

    if (numOfHits == locationCells.length) {
        result = "kill"; ← We're out of the loop, but let's see if we're
                           now 'dead' (hit 3 times) and change the
                           result String to "Kill"
    } // end if

    System.out.println(result); ← Display the result for the user ("miss",
                               unless it was changed to "hit" or "kill")
    return result; ← Return the result back to
                    the calling method
} // end method

```

## Just the new stuff

The things we haven't seen before are on this page. Stop worrying! There are more details later in the chapter. This is just enough to get you going.

### ① The for loop

Read this for loop declaration as "repeat for each element in the 'locationCells' array: take the next element in the array and assign it to the int variable 'cell'."

The colon (:) means "in", so the whole thing means "for each int value IN locationCells..."

**for (int cell : locationCells) { }**

Declare a variable that will hold one element from the array. Each time through the loop, this variable (in this case an int variable named "cell") will hold a different element from the array, until there are no more elements (or the code does a "break" ... see #4 below).

The array to iterate over in the loop. Each time through the loop, the next element in the array will be assigned to the variable "cell". (More on this at the end of this chapter.)

### ② The post-increment operator

**numOfHits++**

The ++ means add 1 to whatever's there (in other words, increment by 1).

numOfHits++ is the same (in this case) as saying numOfHits = numOfHits + 1, with less typing.

### ③ break statement

**break;**

Gets you out of a loop. Immediately. Right here. No iteration, no boolean test, just get out now!

## SimpleStartup class

prep code test code real code

there are no  
Dumb Questions

**Q:** In the beginning of the book, there was an example of a *for* loop that was really different from this one—are there two different styles of *for* loops?

**A:** Yes! From the first version of Java there has been a single kind of *for* loop (explained later in this chapter) that looks like this:

```
for (int i = 0; i < 10; i++)  
{  
    // do something 10 times  
}
```

You can use this format for any kind of loop you need. But... since Java 5, you can also use the *enhanced for* loop (that's the official description) when your loop needs to iterate over the elements in an array (or *another* kind of collection, as you'll see in the *next* chapter). You can always use the plain old *for* loop to iterate over an array, but the *enhanced for* loop makes it easier.

**Q:** If you can add one to an int by using `++`, can you also subtract one in some way?

**A:** Yep absolutely. Hopefully it's not too surprising to find out that the syntax is `--` (two minuses), like this:

```
countdown = i--;
```

## Final code for SimpleStartup and SimpleStartupTestDrive

```
public class SimpleStartupTestDrive {  
    public static void main(String[] args) {  
        SimpleStartup dot = new SimpleStartup();  
        int[] locations = {2, 3, 4};  
        dot.setLocationCells(locations);  
        int userGuess = 2;  
        String result = dot.checkYourself(userGuess);  
        String testResult = "failed";  
        if (result.equals("hit")) {  
            testResult = "passed";  
        }  
        System.out.println(testResult);  
    }  
}
```

```
class SimpleStartup {  
    private int[] locationCells;  
    private int numHits = 0;  
  
    public void setLocationCells(int[] locs) {  
        locationCells = locs;  
    }  
  
    public String checkYourself(int guess) {  
        String result = "miss";  
        for (int cell : locationCells) {  
            if (guess == cell) {  
                result = "hit";  
                numHits++;  
                break;  
            } // end if  
        } // end for  
        if (numHits ==  
            locationCells.length) {  
            result = "kill";  
        } // end if  
        System.out.println(result);  
        return result;  
    } // end method  
} // close class
```

There's a little bug lurking here. It compiles and runs, but...don't worry about it for now, but we will have to face it a little later.

**What should we see when we run this code?**

The test code makes a `SimpleStartup` object and gives it a location at 2,3,4. Then it sends a fake user guess of "2" into the `checkYourself()` method. If the code is working correctly, we should see the result print out:

```
% java SimpleStartupTestDrive  
hit  
passed
```



## Sharpen your pencil

We built the test class and the SimpleStartup class. But we still haven't made the actual *game*. Given the code on the opposite page and the spec for the actual game, write in your ideas for prep code for the game class. We've given you a few lines here and there to get you started. The actual game code is on the next page, so ***don't turn the page until you do this exercise!***

You should have somewhere between 12 and 18 lines (including the ones we wrote, but *not* including lines that have only a curly brace).

**METHOD `public static void main (String [] args)`**

**DECLARE** an int variable to hold the number of user guesses, named `numOfGuesses`

**COMPUTE** a random number between 0 and 4 that will be the starting location cell position

**WHILE** the Startup is still alive:

**GET** user input from the command line

**The SimpleStartupGame needs to do this:**

1. Make the single SimpleStartup object.
2. Make a location for it (three consecutive cells on a single row of seven virtual cells).
3. Ask the user for a guess.
4. Check the guess.
5. Repeat until the Startup is sunk.
6. Tell the user how many guesses it took.

**A complete game interaction**

```

File Edit Window Help Runaway
%java SimpleStartupGame
enter a number 2
hit
enter a number 3
hit
enter a number 4
miss
enter a number 1
kill
You took 4 guesses

```

→ Yours to solve.

## Prep code for the SimpleStartupGame class

### Everything happens in main()

There are some things you'll have to take on faith. For example, we have one line of prep code that says "GET user input from command line." Let me tell you, that's a little more than we want to implement from scratch right now. But happily, we're using OO. And that means you get to ask some *other* class/object to do something for you, without worrying about **how** it does it. When you write prep code, you should assume that *somewhat* you'll be able to do whatever you need to do, so you can put all your brainpower into working out the logic.

#### **public static void main (String [] args)**

**DECLARE** an int variable to hold the number of user guesses, named *numOfGuesses*, and set it to 0

**MAKE** a new SimpleStartup instance

**COMPUTE** a random number between 0 and 4 that will be the starting location cell position

**MAKE** an int array with 3 ints using the randomly generated number, that number incremented by 1, and that number incremented by 2 (example: 3,4,5)

**INVOK**E the *setLocationCells()* method on the SimpleStartup instance

**DECLARE** a boolean variable representing the state of the game, named *isAlive*. **SET** it to true

**WHILE** the Startup is still alive (*isAlive* == true):

**GET** user input from the command line

**// CHECK** the user guess

**INVOK**E the *checkYourself()* method on the SimpleStartup instance

**INCREMENT** *numOfGuesses* variable

**// CHECK** for Startup death

**IF** result is "kill"

**SET** *isAlive* to false (which means we won't enter the loop again)

**PRINT** the number of user guesses

END IF

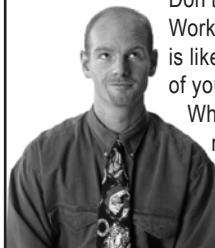
END WHILE

END METHOD

### metacognitive tip

Don't work one part of the brain for too long a stretch at one time. Working just the left side of the brain for more than 30 minutes is like working just your left arm for 30 minutes. Give each side of your brain a break by switching sides at regular intervals.

When you shift to one side, the other side gets to rest and recover. Left-brain activities include things like step-by-step sequences, logical problem-solving, and analysis, while the right-brain kicks in for metaphors, creative problem-solving, pattern-matching, and visualizing.



**BULLET POINTS**

- Your Java program should start with a high-level design.
- Typically you'll write three things when you create a new class:
  - ***prep code***
  - ***test code***
  - ***real (Java) code***
- Prep code should describe *what* to do, not *how* to do it. Implementation comes later.
- Use the prep code to help design the test code.
- A class can have one superclass only.
- Write test code *before* you implement the methods.
- Choose *for* loops over *while* loops when you know how many times you want to repeat the loop code.
- The enhanced for loop is an easy way to loop over an array or collection.
- Use the *increment* operator to add 1 to a variable (`x++;`).
- Use the *decrement* operator to subtract 1 from a variable (`x--;`).
- Use *break* to leave a loop early (i.e., even if the boolean test condition is still true).



# The game's main() method

Just as you did with the SimpleStartup class, be thinking about parts of this code you might want (or need) to improve. The numbered things ① are for stuff we want to point out. They're explained on the opposite page. Oh, if you're wondering why we skipped the test code phase for this class, we don't need a test class for the game. It has only one method, so what would you do in your test code? Make a separate class that would call main() on this class? We didn't bother, we'll just run this to test it.

```

public static void main(String[] args) {
    int numOfGuesses = 0;           ← Make a variable to track how many guesses the user makes.
    GameHelper helper = new GameHelper(); ← This is a special class we wrote that has the method for getting user input. For now, pretend it's part of Java.
}

DECLARE a variable to hold user guess count, and set it to 0

MAKE a Simple-Startup object

COMPUTE a random number between 0 and 4

MAKE an int array with the 3 cell locations, and

INVOKE setLocationCells on the Startup object

DECLARE a boolean isAlive

WHILE the Startup is still alive

GET user input

// CHECK it

INVOKE checkYourself() on Startup

INCREMENT numOfGuesses

IF result is "kill"

SET isAlive to false

PRINT the number of user guesses

```

```

    public static void main(String[] args) {
        int numOfGuesses = 0;           ← Make a variable to track how many guesses the user makes.
        GameHelper helper = new GameHelper(); ← This is a special class we wrote that has the method for getting user input. For now, pretend it's part of Java.

        SimpleStartup theStartup = new SimpleStartup(); Make the Startup object.
        int randomNum = (int) (Math.random() * 5); Make a random number for the first cell, and use it to make the cell locations array.
        int[] locations = {randomNum, randomNum + 1, randomNum + 2};
        theStartup.setLocationCells(locations); ← Give the Startup its locations (the array).

        boolean isAlive = true;          ← Make a boolean variable to track whether the game is still alive, to use in the while loop test. repeat while game is still alive.
        while (isAlive) {               ← Get user guess.
            int guess = helper.getUserInput("enter a number");

            String result = theStartup.checkYourself(guess);
            numOfGuesses++;             ← Increment guess count by one.
            ← Ask the Startup to check the guess; save the returned result.

            if (result.equals("kill")) {
                isAlive = false;         ← Was it a "kill"? if so, set isAlive to false (so we won't re-enter the loop) and print user guess count.
            }
            System.out.println("You took " + numOfGuesses + " guesses");
        } // close if
    } // close while
}

```

## random() and getUserInput()

Two things that need a bit more explaining are on this page. This is just a quick look to keep you going; more details on the GameHelper class are at the end of this chapter.

- ① Make a random number

```
int randomNum = (int) (Math.random() * 5)
```

We declare an int variable to hold the random number we get back.

A class that comes with Java.

A static method of the Math class.

This is a 'cast', and it forces the thing immediately after it to become the type of the cast (i.e., the type in the brackets). Math.random returns a double, so we have to cast it to an int (we want a nice whole number between 0 and 4). In this case, the cast chops off the fractional part of the double.

The Math.random method returns a number from zero to just less than one. So this formula (with the cast) returns a number from 0 to 4 (i.e., 0 - 4.999.., cast to an int).

Math.random() has been around forever, so you'll see code like this in the Real World. These days you can use java.util.Random's nextInt() method instead, which is more convenient (you don't have to cast the result to an int).

The Random class is in a different package. Since we haven't covered importing packages yet (it's in the next chapter), we've used Math.random() instead.

- ② Getting user input using the GameHelper class

An instance we made earlier of a class that we built to help with the game. It's called GameHelper and you haven't seen it yet (you will).

This method takes a String argument that it uses to prompt the user at the command line. Whatever you pass in here gets displayed in the terminal just before the method starts looking for user input.

```
int guess = helper.getUserInput("enter a number");
```

We declare an int variable to hold the user input we get back (3, 5, etc.).

A method of the GameHelper class that asks the user for command-line input, reads it in after the user hits RETURN, and gives back the result as an int.

## One last class: GameHelper

We made the *Startup* class.

We made the *game* class.

All that's left is the **helper class**—the one with the `getUserInput()` method. The code to get command-line input is more than we want to explain right now. It opens up topics best left for later. (Later, as in Chapter 16, *Saving Objects*.)



Whenever you see this logo,  
you're seeing code that you  
have to type as-is and take  
on faith. Trust it. You'll learn  
how that code works later.

```
import java.util.Scanner;

public class GameHelper {
    public int getUserInput(String prompt) {
        System.out.print(prompt + ": ");
        Scanner scanner = new Scanner(System.in);
        return scanner.nextInt();
    }
}
```

Just copy\* the code below and compile it into a class named GameHelper. Drop all three class files (SimpleStartup, SimpleStartupGame, GameHelper) into the same directory, and make it your working directory.

Yes, we WILL  
take a little more  
of your delicious  
Ready-Bake Code,  
thank you very much!



\*We know how much you enjoy typing, but for those rare moments when you'd rather do something else, we've made the Ready-Bake Code available on [https://oreil.ly/hfJava\\_3e\\_examples](https://oreil.ly/hfJava_3e_examples).

## Let's play

Here's what happens when we run it and enter the numbers 1,2,3,4,5,6. Lookin' good.

### A complete game interaction (your mileage may vary)

```
File Edit Window Help Smile
%java SimpleStartupGame
enter a number 1
miss
enter a number 2
miss
enter a number 3
miss
enter a number 4
hit
enter a number 5
hit
enter a number 6
kill
You took 6 guesses
```

## What's this? A bug?

*Gasp!*

Here's what happens when we enter 1,1,1.

### A different game interaction (yikes)

```
File Edit Window Help Faint
%java SimpleStartupGame
enter a number 1
hit
enter a number 1
hit
enter a number 1
kill
You took 3 guesses
```



**Sharpen your pencil**

**It's a cliff-hanger!**

Will we **find** the bug?  
Will we **fix** the bug?

Stay tuned for the next chapter, where we answer these questions and more...

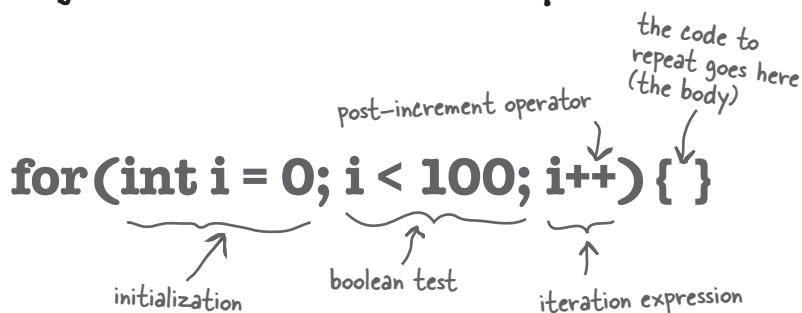
And in the meantime, see if you can come up with ideas for what went wrong and how to fix it.

→ Yours to solve.

## More about for loops

We've covered all the game code for *this* chapter (but we'll pick it up again to finish the deluxe version of the game in the next chapter). We didn't want to interrupt your work with some of the details and background info, so we put it back here. We'll start with the details of for loops, and if you've seen this kind of syntax in another programming language, just skim these last few pages...

### Regular (non-enhanced) for loops



repeat for 100 reps:



**What it means in plain English:** "Repeat 100 times."

**How the compiler sees it:**

- create a variable *i* and set it to 0.
- repeat while *i* is less than 100.
- at the end of each loop iteration, add 1 to *i*.

#### Part One: *initialization*

Use this part to declare and initialize a variable to use within the loop body. You'll most often use this variable as a counter. You can actually initialize more than one variable here, but it's much more common to use a single variable.

#### Part Two: *boolean test*

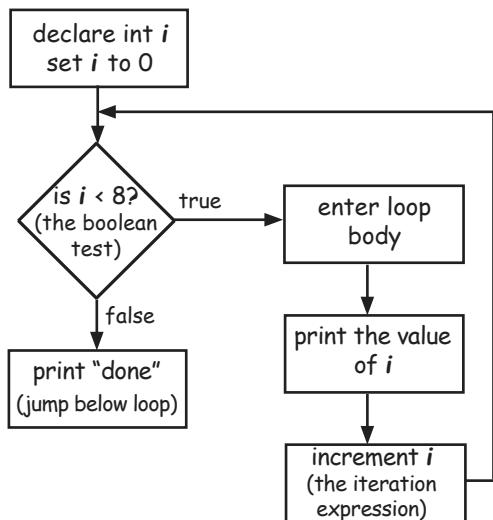
This is where the conditional test goes. Whatever's in there, it *must* resolve to a boolean value (you know, **true** or **false**). You can have a test, like `(x >= 4)`, or you can even invoke a method that returns a boolean.

#### Part Three: *iteration expression*

In this part, put one or more things you want to happen with each trip through the loop. Keep in mind that this stuff happens at the *end* of each loop.

# Trips through a loop

```
for (int i = 0; i < 8; i++) {
    System.out.println(i);
}
System.out.println("done");
```



## Difference between for and while

A *while* loop has only the boolean test; it doesn't have a built-in initialization or iteration expression. A *while* loop is good when you don't know how many times to loop and just want to keep going while some condition is true. But if you *know* how many times to loop (e.g., the length of an array, 7 times, etc.), a *for* loop is cleaner. Here's the loop above rewritten using *while*:

```
int i = 0;           ← we have to declare and
                     initialize the counter
while (i < 8) {
    System.out.println(i);
    i++;             ← we have to increment
                     the counter
}
System.out.println("done");
```

## output:

```
File Edit Window Help Repeat
%java Test
0
1
2
3
4
5
6
7
done
```

++    --

## Pre and Post Increment/Decrement Operator

The shortcut for adding or subtracting 1 from a variable:

**x++;**

is the same as:

**x = x + 1;**

They both mean the same thing in this context:

"add 1 to the current value of x" or "**increment** x by 1"

And:

**x--;**

is the same as:

**x = x - 1;**

Of course that's never the whole story. The placement of the operator (either before or after the variable) can affect the result. Putting the operator *before* the variable (for example, **++x**), means, "first, increment x by 1, and *then* use this new value of x." This only matters when the **++x** is part of some larger expression rather than just a single statement.

**int x = 0;      int z = ++x;**

produces: x is 1, z is 1

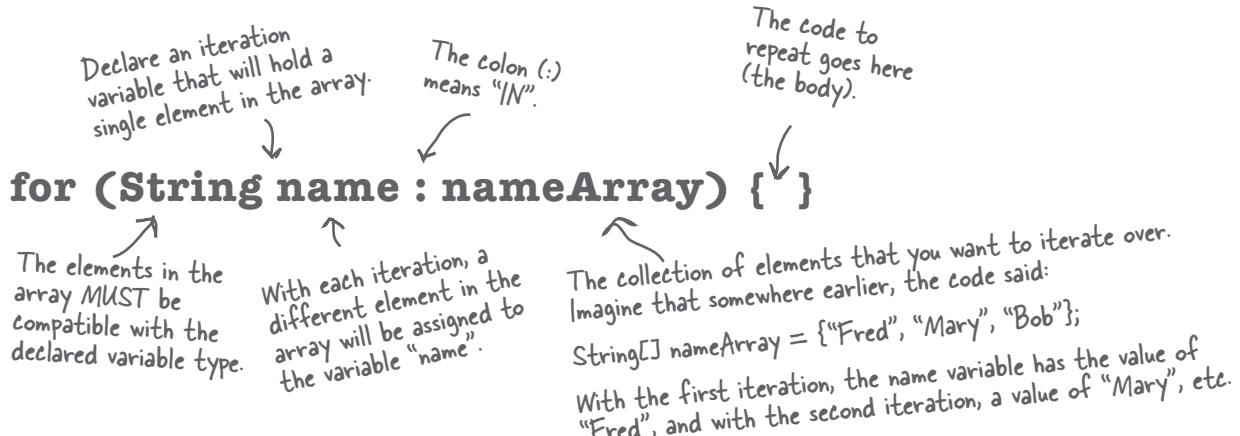
But putting the *++* *after* the x gives you a different result:

**int x = 0;      int z = x++;**

Once this code has run, x is 1, but **z is 0!** z gets the value of x, and *then* x is incremented.

## The enhanced for loop

The Java language added a second kind of *for* loop called the *enhanced for* back in Java 5. This makes it easier to iterate over all the elements in an array or other kinds of collections (you'll learn about *other* collections in the next chapter). That's really all that the enhanced for gives you—a simpler way to walk through all the elements in the collection. We'll see the enhanced for loop in the next chapter too, when we talk about collections that *aren't* arrays.



**What it means in plain English:** “For each element in `nameArray`, assign the element to the ‘`name`’ variable, and run the body of the loop.”

### How the compiler sees it:

- Create a String variable called `name` and set it to null.
- Assign the first value in `nameArray` to `name`.
- Run the body of the loop (the code block bounded by curly braces).
- Assign the next value in `nameArray` to `name`.
- Repeat while *there are still elements in the array*.

Note: depending on the programming language they've used in the past, some people refer to the enhanced for as the “for each” or the “for in” loop, because that's how it reads: “for EACH thing IN the collection...”

### Part One: iteration variable declaration

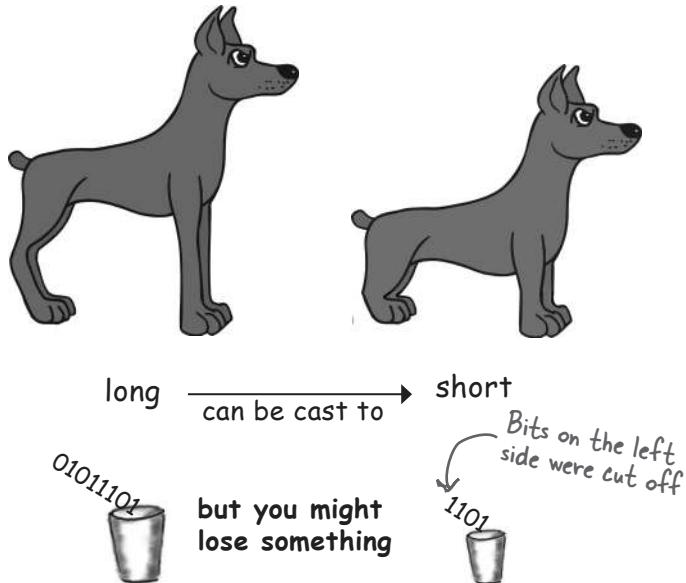
Use this part to declare and initialize a variable to use within the loop body. With each iteration of the loop, this variable will hold a different element from the collection. The type of this variable must be compatible with the elements in the array! For example, you can't declare an `int` iteration variable to use with a `String[]` array.

### Part Two: the actual collection

This must be a reference to an array or other collection. Again, don't worry about the *other* non-array kinds of collections yet—you'll see them in the next chapter.

# Casting primitives

Before we finish the chapter, we want to tie up a loose end. When we used `Math.random()`, we had to *cast* the result to an `int`. Casting one numeric type to another can change the value itself. It's important to understand the rules so you're not surprised by this.



In Chapter 3, *Know Your Variables*, we talked about the sizes of the various primitives and how you can't shove a big thing directly into a small thing:

```
long y = 42;
int x = y;           // won't compile
```

A long is bigger than an int, and the compiler can't be sure where that long has been. It might have been out partying with the other longs, and taking on really big values. To force the compiler to jam the value of a bigger primitive variable into a smaller one, you can use the cast operator. It looks like this:

```
long y = 42;      // so far so good
int x = (int) y; // x = 42 cool!
```

Putting in the cast tells the compiler to take the value of `y`, chop it down to `int` size, and set `x` equal to whatever is left. If the value of `y` was bigger than the maximum value of `x`, then what's left will be a weird (but calculable\*) number:

```
long y = 40002;      // 40002 exceeds the 16-bit limit of a short
short x = (short) y; // x now equals -25534!
```

Still, the point is that the compiler lets you do it. And let's say you have a floating-point number and you just want to get at the whole number (`int`) part of it:

```
float f = 3.14f;
int x = (int) f; // x will equal 3
```

And don't even think about casting anything to a boolean or vice versa—just walk away.

\*It involves sign bits, binary, “two’s complement,” and other geekery.



## BE the JVM



The Java file on this page represents a complete source file. Your job is to play JVM and determine what would be the output when the program runs.

```
class Output {  
    public static void main(String[] args) {  
        Output output = new Output();  
        output.go();  
    }  
  
    void go() {  
        int value = 7;  
        for (int i = 1; i < 8; i++) {  
            value++;  
            if (i > 4) {  
                System.out.print(++value + " ");  
            }  
            if (value > 14) {  
                System.out.println(" i = " + i);  
                break;  
            }  
        }  
    }  
}
```

```
File Edit Window Help OM  
% java Output  
12 14
```

-or-

```
File Edit Window Help Incense  
% java Output  
12 14 x = 6
```

-or-

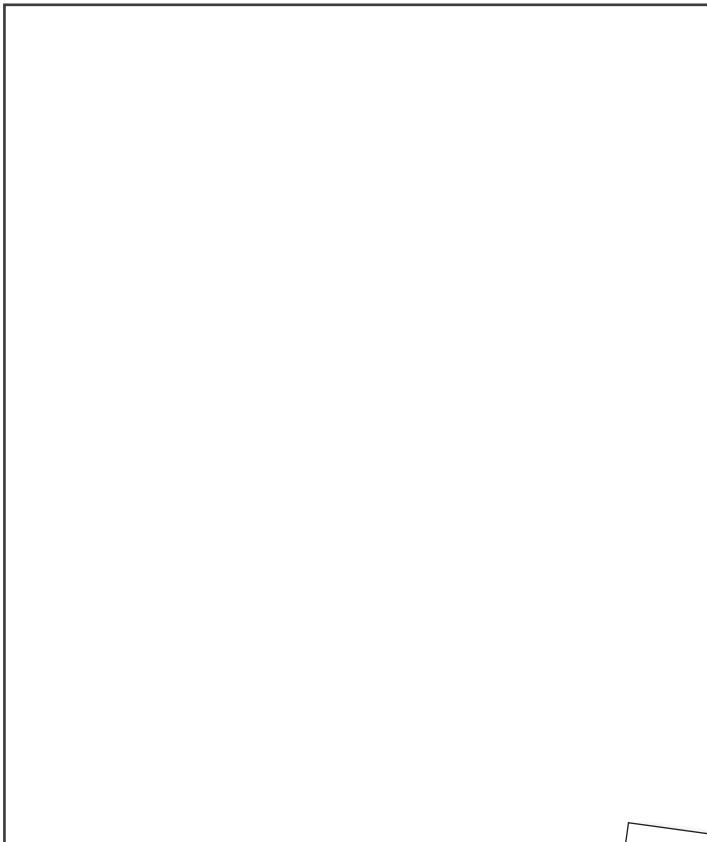
```
File Edit Window Help Believe  
% java Output  
13 15 x = 6
```

→ Answers on page 122.



## Code Magnets

A working Java program is all scrambled up on the fridge. Can you reconstruct the code snippets to make a working Java program that produces the output listed below? Some of the curly braces fell on the floor and they were too small to pick up, so feel free to add as many of those as you need!



i++;

if (i == 1) {

System.out.println(i + " " + j);

class MultiFor {

for(int j = 4; j > 2; j--) {

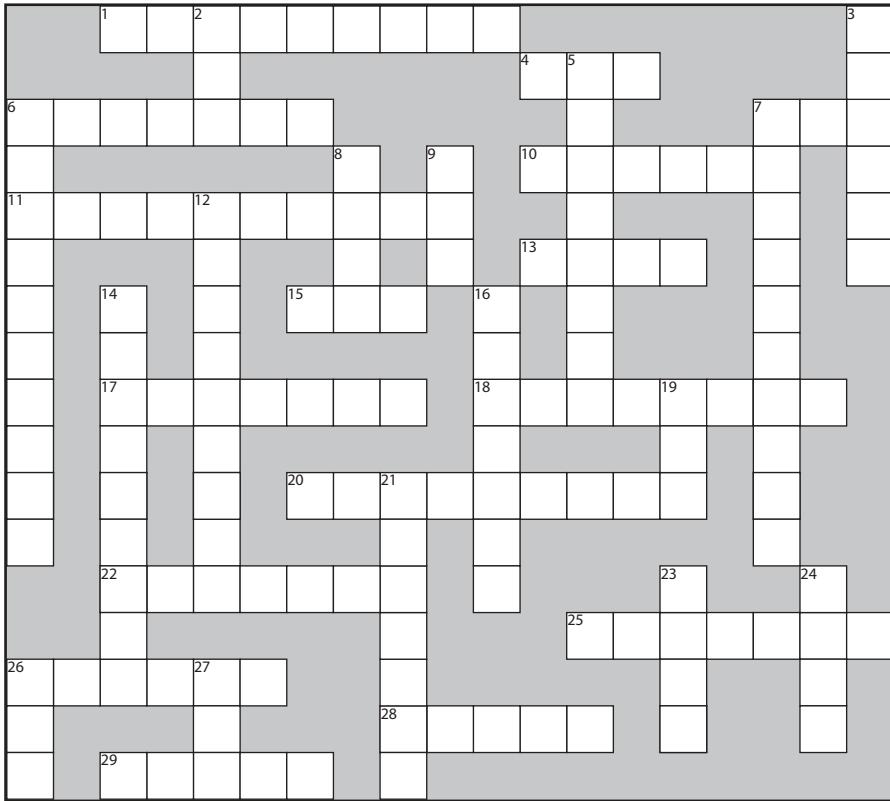
for(int i = 0; i < 4; i++) {

public static void main(String[] args) {

```
File Edit Window Help Raid
% java MultiFor
0 4
0 3
1 4
1 3
3 4
3 3
```

—————> Answers on page 122.

## puzzle: JavaCross



# JavaCross

How does a crossword puzzle help you learn Java? Well, all of the words **are** Java related. In addition, the clues provide metaphors, puns, and the like. These mental twists and turns burn alternate routes to Java knowledge right into your brain!

### Across

- 1. Fancy computer word for build
- 4. Multipart loop
- 6. Test first
- 7. 32 bits
- 10. Method's answer
- 11. Prep code-esque
- 13. Change
- 15. The big toolkit
- 17. An array unit
- 18. Instance or local

### Down

- 20. Automatic toolkit
- 22. Looks like a primitive, but..
- 25. Un-castable
- 26. Math method
- 28. Iterate over me
- 29. Leave early
- 2. Increment type
- 3. Class's workhorse
- 5. Pre is a type of \_\_\_\_\_
- 6. For's iteration \_\_\_\_\_
- 7. Establish first value
- 8. While or For
- 9. Update an instance variable
- 12. Toward blastoff
- 14. A cycle
- 16. Talkative package
- 19. Method messenger (abbrev.)
- 21. As if
- 23. Add after
- 24. Pi house
- 26. Compile it and \_\_\_\_\_
- 27. ++ quantity

→ Answers on page 123.



A short Java program is listed below. One block of the program is missing. Your challenge is to **match the candidate block of code** (on the left) **with the output** that you'd see if the block were inserted. Not all the lines of output will be used, and some of the lines of output might be used more than once. Draw lines connecting the candidate blocks of code with their matching command-line output.

→ Answers on page 123.

```
public static void main(String[] args) {
    int x = 0;
    int y = 30;
    for (int outer = 0; outer < 3; outer++) {
        for (int inner = 4; inner > 1; inner--) {
              ← Candidate code goes here
            y = y - 2;
            if (x == 6) {
                break;
            }
            x = x + 3;
        }
        y = y - 2;
    }
    System.out.println(x + " " + y);
}
```

#### Candidates:

x = x + 3;

x = x + 6;

x = x + 2;

x++;

x--;

x = x + 0;

#### Possible output:

45 6

36 6

54 6

60 10

18 6

6 14

Match each candidate with one of the possible outputs



### Be the JVM (from page 118)

```
class Output {  
  
    public static void main(String[] args) {  
        Output output = new Output();  
        output.go();  
    }  
  
    void go() {  
        int value = 7;  
        for (int i = 1; i < 8; i++) {  
            value++;  
            if (i > 4) {  
                System.out.print(++value + " ");  
            }  
            if (value > 14) {  
                System.out.println(" i = " + i);  
                break;  
            }  
        }  
    }  
}
```

**Did you remember to factor in the break statement? How did that affect the output?**

File Edit Window Help MotorcycleMaintenance

```
% java Output  
13 15 x = 6
```

### Code Magnets (from page 119)

```
class MultiFor {  
  
    public static void main(String[] args) {  
        for (int i = 0; i < 4; i++) {  
  
            for (int j = 4; j > 2; j--) {  
                System.out.println(i + " " + j);  
            }  
  
            if (i == 1) {  
                i++;  
            }  
        }  
    }  
}
```

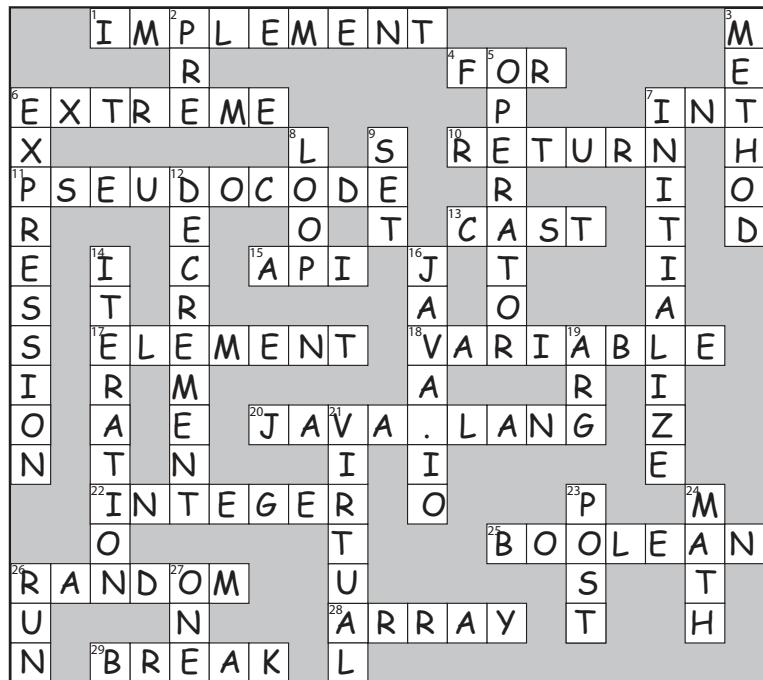
**What would happen if this code block came before the 'j' for loop?**

File Edit Window Help Monopole

```
% java MultiFor  
0 4  
0 3  
1 4  
1 3  
3 4  
3 3
```



## Puzzle Solutions



## JavaCross (from page 120)

## Mixed Messages (from page 121)

### Candidates:

```
x = x + 3;
```

```
x = x + 6;
```

```
x = x + 2;
```

```
x++;
```

```
x--;
```

```
x = x + 0;
```

### Possible output:

```
45 6
```

```
36 6
```

```
54 6
```

```
60 10
```

```
18 6
```

```
6 14
```

```
12 14
```



# Using the Java Library



**Java ships with hundreds of prebuilt classes.** You don't have to reinvent the wheel if you know how to find what you need in the Java library, known as the **Java API**. *You've got better things to do.* If you're going to write code, you might as well write *only* the parts that are truly custom for your application. You know those programmers who walk out the door each night at 5 PM? The ones who don't even show up until 10 AM? **They use the Java API.** And about eight pages from now, so will you. The core Java library is a giant pile of classes just waiting for you to use like building blocks, to assemble your own program out of largely prebuilt code. The Ready-Bake Java we use in this book is code you don't have to create from scratch, but you still have to type it. The Java API is full of code you don't even have to type. All you need to do is learn to use it.

we still have a bug

# In our last chapter, we left you with the cliff-hanger: a bug

## How it's supposed to look

Here's what happens when we run it and enter the numbers 1,2,3,4,5,6. Lookin' good.

### A complete game interaction (your mileage may vary)

```
File Edit Window Help Smile
%java SimpleStartupGame
enter a number 1
miss
enter a number 2
miss
enter a number 3
miss
enter a number 4
hit
enter a number 5
hit
enter a number 6
kill
You took 6 guesses
```

## How the bug looks

Here's what happens when we enter 2,2,2.

### A different game interaction (yikes)

```
File Edit Window Help Faint
%java SimpleStartupGame
enter a number 2
hit
enter a number 2
hit
enter a number 2
kill
You took 3 guesses
```

In the current version, once you get a hit, you can simply repeat that hit two more times for the kill!

# So what happened?

Here's where it goes wrong. We counted a hit every time the user guessed a cell location, even if that location had already been hit!

We need a way to know that when a user makes a hit, they haven't previously hit that cell. If they have, then we don't want to count it as a hit.

```

public String checkYourself(int guess) {
    String result = "miss"; ← Make a variable to hold the result
    we'll return. Put "miss" in as the
    default (i.e., we assume a "miss").

    for (int cell : locationCells) { ← Repeat with each
        thing in the array.

            if (guess == cell) { ← Compare the user
                guess to this element
                (cell), in the array.

                    result = "hit"; ← we got a hit!
                    numOfHits++; ←

                    break; ← Get out of the loop; no need
                    to test the other cells.

                } // end if
            }
        } // end for

        if (numOfHits == locationCells.length) { ← We're out of the loop, but
            let's see if we're now 'dead'
            (hit 3 times) and change the
            result String to "kill".
            result = "kill";
        }
    } // end if

    System.out.println(result); ← Display the result for the user ("miss"
    unless it was changed to "hit" or "kill").

    return result; ← Return the result back to
    the calling method.
} // end method

```

# How do we fix it?

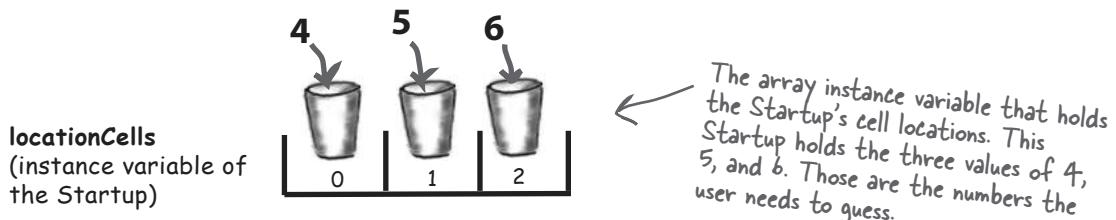
We need a way to know whether a cell has already been hit. Let's run through some possibilities, but first, we'll look at what we know so far...

We have a virtual row of seven cells, and a Startup will occupy three consecutive cells somewhere in that row. This virtual row shows a Startup placed at cell locations 4, 5, and 6.



The virtual row, with the 3 cell locations for the Startup object.

The Startup has an instance variable—an int array—that holds that Startup object's cell locations.

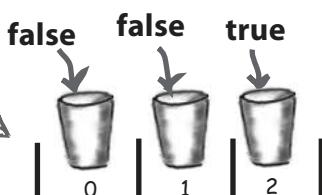


The array instance variable that holds the Startup's cell locations. This array holds the three values of 4, 5, and 6. Those are the numbers the user needs to guess.

## ① Option one

We could make a second array, and each time the user makes a hit, we store that hit in the second array, and then check that array each time we get a hit, to see if that cell has been hit before.

hitCells array  
(this would be a new boolean array instance variable of the Startup)



A 'true' in a particular index in the array means that the cell location at that same index in the OTHER array (locationCells) has been hit.

This array holds three values representing the 'state' of each cell in the Startup's location cells array. For example, if the cell at index 2 is hit, then set index 2 in the "hitCells" array to 'true'.

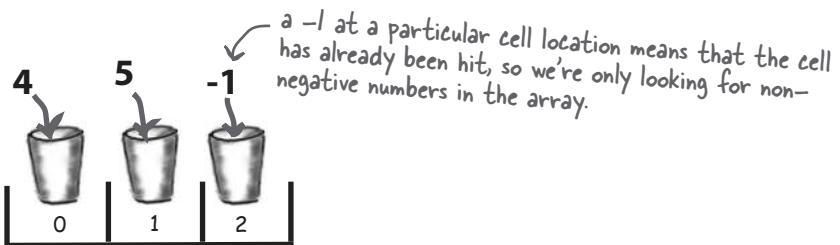
## Option one is too clunky

Option one seems like more work than you'd expect. It means that each time the user makes a hit, you have to change the state of the *second* array (the hitCells array), oh—but first you have to *CHECK* the hitCells array to see if that cell has already been hit anyway. It would work, but there's got to be something better...

### ② Option two

We could just keep the one original array but change the value of any hit cells to -1. That way, we only have **ONE** array to check and manipulate.

`locationCells`  
(instance variable of  
the Startup)



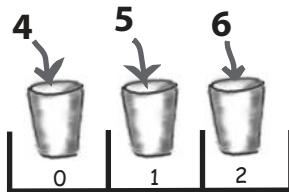
## Option two is a little better, but still pretty clunky

Option two is a little less clunky than option one, but it's not very efficient. You'd still have to loop through all three slots (index positions) in the array, even if one or more are already invalid because they've been "hit" (and have a -1 value). There has to be something better...

### ③ Option three

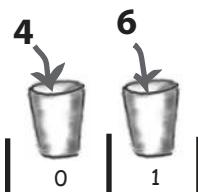
We delete each cell location as it gets hit and then modify the array to be smaller. Except arrays can't change their size, so we have to make a **new** array and copy the remaining cells from the old array into the new smaller array.

**locationCells** array  
BEFORE any cells  
have been hit



The array starts out with a size of 3, and we loop through all 3 cells (positions in the array) to look for a match between the user guess and the cell value (4,5,6).

**locationCells** array  
AFTER cell '5', which  
was at index 1 in the  
array, has been hit



When cell '5' is hit, we make a new, smaller array with only the remaining cell locations, and assign it to the original **locationCells** reference.

Option three would be much better if the array could shrink so that we wouldn't have to make a new smaller array, copy the remaining values in, and reassign the reference.

The original prep code for part of the **checkYourself()** method:

```
REPEAT with each of the location cells in the int array →
    // COMPARE the user guess to the location cell
    IF the user guess matches
        INCREMENT the number of hits
        // FIND OUT if it was the last location cell:
        IF number of hits is 3, RETURN "kill"
        ELSE it was not a kill, so RETURN "hit"
    END IF
    ELSE user guess did not match, so RETURN "miss"
END IF
END REPEAT
```

Life would be good if only we could change it to:

```
REPEAT with each of the remaining location cells →
    // COMPARE the user guess to the location cell
    IF the user guess matches
        REMOVE this cell from the array
        // FIND OUT if it was the last location cell:
        IF the array is now empty, RETURN "kill"
        ELSE it was not a kill, so RETURN "hit"
    END IF
    ELSE user guess did not match, so RETURN "miss"
END IF
END REPEAT
```



If only I could find an array  
that could **shrink** when you **remove**  
something. And one that you didn't have  
to loop through to check each element, but  
instead you could just **ask it if it contains**  
what you're looking for. And it would let you  
**get** things out of it, without having to know  
exactly which slot the things are in.  
That would be dreamy. But I know it's  
just a fantasy...

when arrays aren't enough

# Wake up and smell the library

**As if by magic, there really *is* such a thing.**

**But it's not an array, it's an *ArrayList*.**

**A class in the core Java library (the API).**

The Java Platform, Standard Edition (Java SE) ships with hundreds of pre-built classes. Just like our Ready-Bake Code. Except that these built-in classes are already compiled.

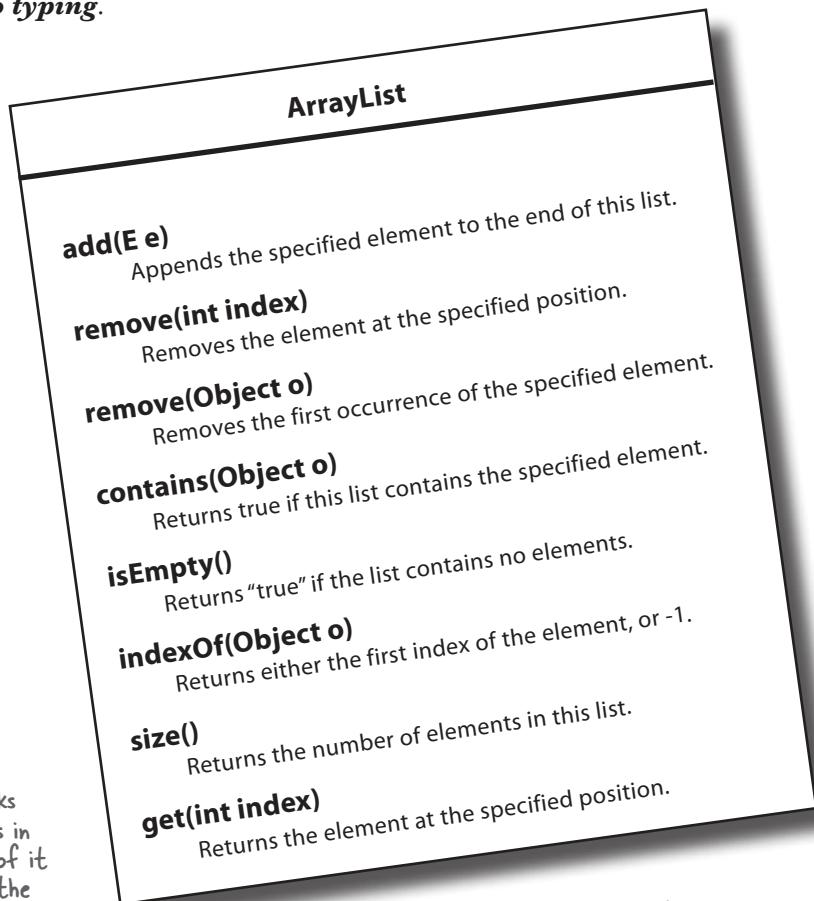
***That means no typing.***

Just use 'em.

ArrayList is one of a gazillion classes in the Java library.

You can use it in your code as if you wrote it yourself.

(Note: the add(E e) method looks a little strange...we'll get to this in Chapter 11. For now, just think of it as an add() method that takes the object you want to add.)



This is just a sample of SOME of the methods in ArrayList.

# Some things you can do with ArrayList

## ① Make one

```
ArrayList<Egg> myList = new ArrayList<Egg>();
```

Don't worry about this new <Egg> angle-bracket syntax, right now; it just means "make this a list of Egg objects."

A new ArrayList object is created on the heap. It's little because it's empty.

## ② Put something in it

```
Egg egg1 = new Egg();
```

```
myList.add(egg1);
```

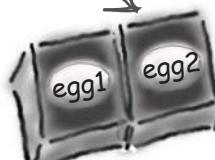


Now the ArrayList grows a "box" to hold the Egg object.

## ③ Put another thing in it

```
Egg egg2 = new Egg();
```

```
myList.add(egg2);
```



The ArrayList grows again to hold the second Egg object.

## ④ Find out how many things are in it

```
int theSize = myList.size();
```

The ArrayList is holding 2 objects so the size() method returns 2:

## ⑤ Find out if it contains something

```
boolean isIn = myList.contains(egg1);
```

The ArrayList DOES contain the Egg object referenced by 'egg1', so contains() returns true.

## ⑥ Find out where something is (i.e., its index)

```
int idx = myList.indexOf(egg2);
```

ArrayList is zero-based (means first index is 0) and since the object referenced by 'egg2' was the second thing in the list, indexOf() returns 1.

## ⑦ Find out if it's empty

```
boolean empty = myList.isEmpty();
```

it's definitely NOT empty, so isEmpty() returns false.



Hey look — it shrank!

## ⑧ Remove something from it

```
myList.remove(egg1);
```

## when arrays aren't enough



Fill in the rest of the table below by looking at the ArrayList code on the left and putting in what you think the code might be if it were using a regular array instead. We don't expect you to get all of them exactly right, so just make your best guess.

ArrayList	Regular array
ArrayList<String> myList = new ArrayList<String>();	String [] myList = new String[2];
String a = "whooahoo"; myList.add(a);	String a = "whooahoo";
String b = "Frog"; myList.add(b);	String b = "Frog";
int theSize = myList.size();	
String str = myList.get(1);	
myList.remove(1);	
boolean isIn = myList.contains(b);	

there are no  
Dumb Questions

**Q:** So ArrayList is cool, but how would I know it exists?

**A:** The question is really, "How do I know what's in the API?" and that's the key to your success as a Java programmer. Not to mention your key to being as lazy as possible while still managing to build software. You might be amazed at how much time you can save when somebody else has already done most of the heavy lifting and all you have to do is step in and create the fun part.

But we digress...the short answer is that you spend some time learning what's in the core API. The long answer is at the end of this chapter, where you'll learn how to do that.

**Q:** But that's a pretty big issue. Not only do I need to know that the Java library comes with ArrayList, but more importantly I have to know that ArrayList is the thing that can do what I want! So how do I go from a need-to-do-something to a-way-to-do-it using the API?

**A:** Now you're really at the heart of it. By the time you've finished this book, you'll have a good grasp of the language, and the rest of your learning curve really is about knowing how to get from a problem to a solution, with you writing the least amount of code. If you can be patient for a few more pages, we start talking about it at the end of this chapter.



**HeadFirst:** So, ArrayLists are like arrays, right?

**ArrayList:** In their dreams! *I* am an *object*, thank you very much.

**HeadFirst:** If I'm not mistaken, arrays are objects too. They live on the heap right there with all the other objects.

**ArrayList:** Sure arrays go on the heap, *duh*, but an array is still a wanna-be ArrayList. A poser. Objects have state *and* behavior, right? We're clear on that. But have you actually tried calling a method on an array?

**HeadFirst:** Now that you mention it, can't say I have. But what method would I call, anyway? I only care about calling methods on the stuff I put *in* the array, not the array itself. And I can use array syntax when I want to put things in and take things out of the array.

**ArrayList:** Is that so? You mean to tell me you actually *removed* something from an array? (Sheesh, where do they *train* you guys?)

**HeadFirst:** Of course I take something out of the array. I say Dog d = dogArray[1], and I get the Dog object at index 1 out of the array.

**ArrayList:** Alright, I'll try to speak slowly so you can follow along. You were *not*, I repeat *not*, removing that Dog from the array. All you did was make a copy of the reference to the Dog and assign it to another Dog variable.

**HeadFirst:** Oh, I see what you're saying. No, I didn't actually remove the Dog object from the array. It's still there. But I can just set its reference to null, I guess.

**ArrayList:** But I'm a first-class object, so I have methods, and I can actually, you know, *do* things like remove the Dog's reference from myself, not just set it to null. And I can change my size, *dynamically* (look it up). Just try to get an *array* to do that!

**HeadFirst:** Gee, hate to bring this up, but the rumor is that you're nothing more than a glorified but less-efficient array. That in fact you're just a wrapper for an array, adding extra methods for things like resizing that I would have had to write myself. And while we're at it, *you can't even hold primitives!* Isn't that a big limitation?

**ArrayList:** I can't *believe* you buy into that urban legend. No, I am *not* just a less-efficient array. I will admit that there are a few *extremely* rare situations where an array might be just a tad, I repeat, *tad* bit faster for certain things. But is it worth the *minuscule* performance gain to give up all this *power*? Still, look at all this *flexibility*. And as for the primitives, of course you can put a primitive in an ArrayList, as long as it's wrapped in a primitive wrapper class (you'll see a lot more on that in Chapter 10). And if you're using Java 5 or above, that wrapping (and unwrapping when you take the primitive out again) happens automatically. And alright, I'll *acknowledge* that yes, if you're using an ArrayList of *primitives*, it probably is faster with an array, because of all the wrapping and unwrapping, but still...who really uses primitives *these* days?

Oh, look at the time! *I'm late for Pilates.* We'll have to do this again sometime.

# Solution



## ArrayList

```
ArrayList<String> myList = new
ArrayList<String>();
```

```
String a = "whoohoo";
myList.add(a);
```

```
String b = "Frog";
myList.add(b);
```

```
int theSize = myList.size();
```

```
String str = myList.get(1);
```

```
myList.remove(1);
```

```
boolean isIn = myList.contains(b);
```

```
String [] myList = new String[2];
```

```
String a = "whoohoo";
myList[0] = a;
```

```
String b = "Frog";
myList[1] = b;
```

```
int theSize = myList.length;
```

```
String str = myList[1];
```

```
myList[1] = null;
```

```
boolean isIn = false;
for (String item : myList) {
    if (b.equals(item)) {
        isIn = true;
        break;
    }
}
```

Here's where it  
starts to look  
really different...

Notice how with ArrayList, you're working with an object of type ArrayList, so you're just invoking regular old methods on a regular old object, using the regular old dot operator.

With an *array*, you use *special array syntax* (like myList[0] = foo) that you won't use anywhere else except with arrays. Even though an array *is* an object, it lives in its own special world, and you can't invoke any methods on it, although you can access its one and only instance variable, *length*.

# Comparing ArrayList to a regular array

## ① A plain old array has to know its size at the time it's created.

But for ArrayList, you just make an object of type ArrayList. Every time. It never needs to know how big it should be, because it grows and shrinks as objects are added or removed.

`new String[2]` Needs a size.

`new ArrayList<String>()`

No size required (although you can give it an initial size if you want to).

## ② To put an object in a regular array, you must assign it to a specific location.

(An index from 0 to one less than the length of the array.)

`myList[1] = b;`

Needs an index.

If that index is outside the boundaries of the array (like the array was declared with a size of 2, and now you're trying to assign something to index 3), it blows up at runtime.

With ArrayList, you can specify an index using the `add(anInt, anObject)` method, or you can just keep saying `add(anObject)` and the ArrayList will keep growing to make room for the new thing.

`myList.add(b);`

No index.

## ③ Arrays use array syntax that's not used anywhere else in Java.

But ArrayLists are plain old Java objects, so they have no special syntax.

`myList[1]`

The array brackets [ ] are special syntax used only for arrays.

## ④ ArrayLists are parameterized.

We just said that unlike arrays, ArrayLists have no special syntax. But they *do* use something special—**parameterized types**.\*

`ArrayList<String>`

The `<String>` in angle brackets is a “type parameter.” `ArrayList<String>` means simply “a list of Strings,” as opposed to `ArrayList<Dog>`, which means, “a list of Dogs.”

Using the `<TypeGoesHere>` syntax, we can declare and create an ArrayList that knows (and restricts) the types of objects it can hold. We'll look at the details of parameterized types in ArrayLists in Chapter 11, *Data Structures*, so for now, don't think too much about the angle bracket `<>` syntax you see when we use ArrayLists. Just know that it's a way to force the compiler to allow only a specific type of object (*the type in angle brackets*) in the ArrayList.

\*Parameterized types were added to Java in Java 5, which came out so long ago that you are almost definitely using a version that supports them!

# Let's fix the Startup code

Remember, this is how the buggy version looks:

```
class Startup {  
    private int[] locationCells;  
    private int numOfHits = 0;  
  
    public void setLocationCells(int[] locs) {  
        locationCells = locs;  
    }  
  
    public String checkYourself(int guess) {  
        String result = "miss";  
        for (int cell : locationCells) {  
            if (guess == cell) {  
                result = "hit";  
                numOfHits++;  
                break;  
            }  
        } // end for  
        if (numOfHits == locationCells.length) {  
            result = "kill";  
        } // end if  
        System.out.println(result);  
        return result;  
    } // end method  
} // close class
```

We've renamed the class `Startup` now (instead of `SimpleStartup`), for the new advanced version, but this is the same code you saw in the last chapter.

Where it all went wrong. We counted each guess as a hit, without checking whether that cell had already been hit.

## New and improved Startup class

```

import java.util.ArrayList;           ← Ignore this line for
                                         now; we talk about
                                         it at the end of the
                                         chapter.

public class Startup {

    private ArrayList<String> locationCells;
    // private int numofHits;           ← Change the int array to an ArrayList that holds Strings.
    // don't need to track this now

    public void setLocationCells(ArrayList<String> locs) {
        locationCells = locs;
    }

    public String checkYourself(String userInput) {
        String result = "miss";
        int index = locationCells.indexOf(userInput);

        if (index >= 0) {             ← If index is greater than or equal to zero,
                                      the user guess is definitely in the list, so
                                      remove it.
            locationCells.remove(index); ←

            if (locationCells.isEmpty()) { ← If the list is empty, this
                result = "kill";
            } else {
                result = "hit";
            } // end if
        } // end outer if
        return result;
    } // end method
} // close class

```

**Now with  
ArrayList  
power!**

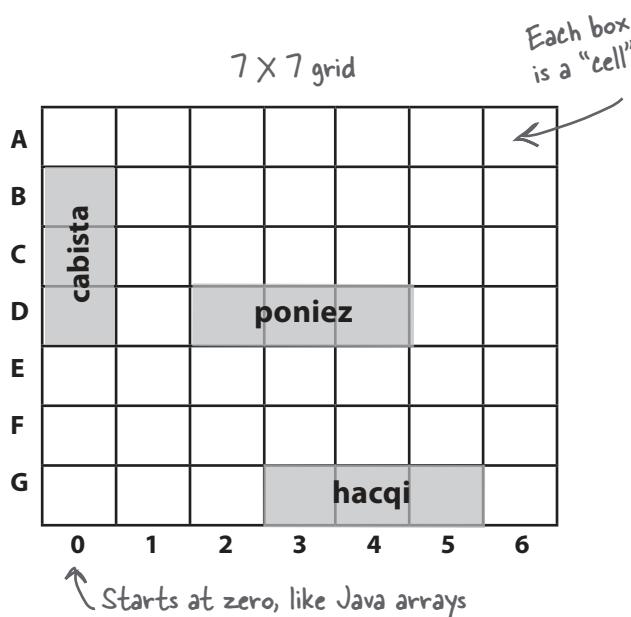
# Let's build the REAL game: “Sink a Startup”

We've been working on the “simple” version, but now let's build the real one. Instead of a single row, we'll use a grid. And instead of one Startup, we'll use three.

**Goal:** Sink all of the computer's Startups in the fewest number of guesses. You're given a rating level based on how well you perform.

**Setup:** When the game program is launched, the computer places three Startups, randomly, on the **virtual 7 x 7 grid**. When that's complete, the game asks for your first guess.

**How you play:** We haven't learned to build a GUI yet, so this version works at the command line. The computer will prompt you to enter a guess (a cell), which you'll type at the command line (as “A3,” “C5,” etc.). In response to your guess, you'll see a result at the command-line, either “hit,” “miss,” or “You sunk poniez” (or whatever the lucky Startup of the day is). When you've sent all three Startups to that big 404 in the sky, the game ends by printing out your rating.



**You're going to build the Sink a Startup game, with a 7 x 7 grid and three Startups. Each Startup takes up three cells.**

part of a game interaction

```

File Edit Window Help Sell
%java StartupBust
Enter a guess  A3
miss
Enter a guess  B2
miss
Enter a guess  C4
miss
Enter a guess  D2
hit
Enter a guess  D3
hit
Enter a guess  D4
Ouch! You sunk poniez :(
kill
Enter a guess  G3
hit
Enter a guess  G4
hit
Enter a guess  G5
Ouch! You sunk hacqi :(
All Startups are dead! Your stock
is now worthless
Took you long enough. 62 guesses.

```

# What needs to change?

We have three classes that need to change: the Startup class (which is now called Startup instead of SimpleStartup), the game class (StartupBust), and the game helper class (which we won't worry about now).

## A Startup class

- Add a *name* variable**

to hold the name of the Startup ("poniez," "cabista," etc.) so each Startup can print its name when it's killed (see the output screen on the opposite page).

## B StartupBust class (the game)

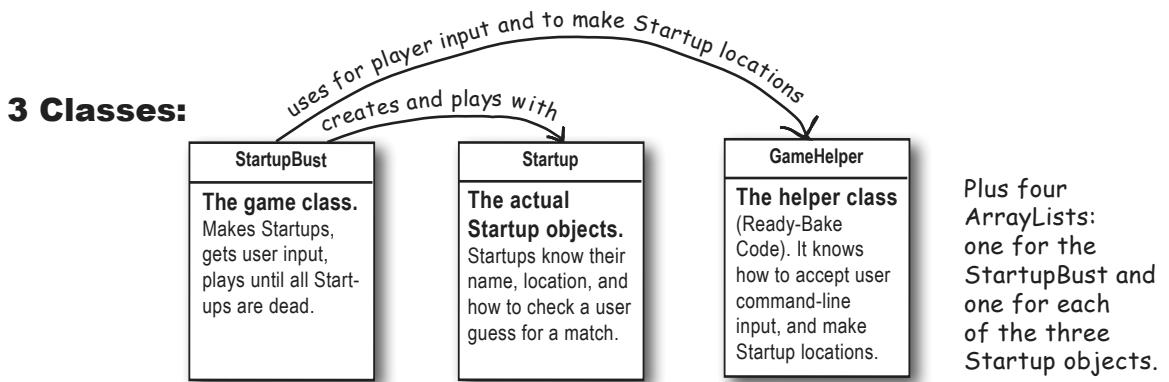
- Create *three* Startups instead of one.**
- Give each of the three Startups a *name*.**  
Call a setter method on each Startup instance so that the Startup can assign the name to its name instance variable.

## StartupBust class continued...

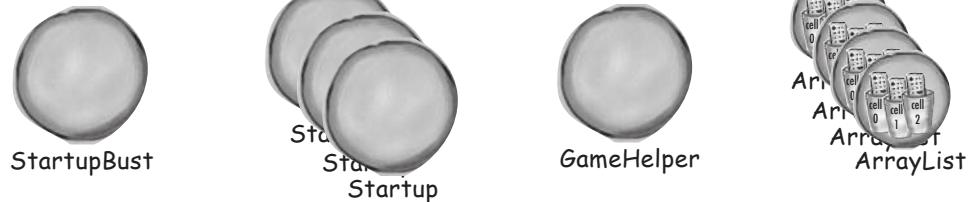
- Put the Startups on a grid rather than just a single row, and do it for all three Startups.**

This step is now way more complex than before, if we're going to place the Startups randomly. Since we're not here to mess with the math, we put the algorithm for giving the Startups a location into the GameHelper (Ready-Bake Code) class.

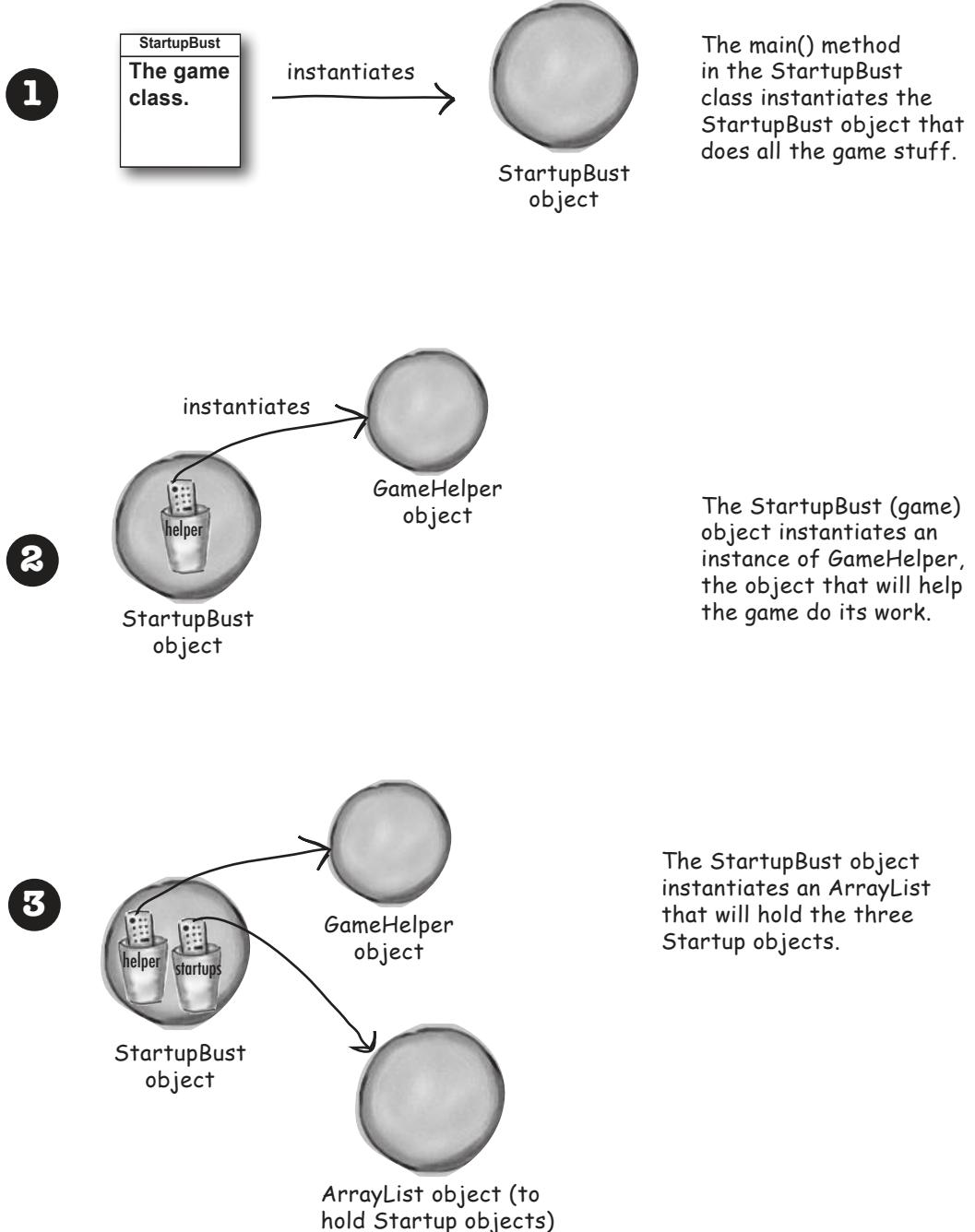
- Check each user guess with all three Startups, instead of just one.**
- Keep playing the game** (i.e., accepting user guesses and checking them with the remaining Startups) **until there are no more live Startups.**
- Get out of main.** We kept the simple one in main just to...keep it simple. But that's not what we want for the *real* game.

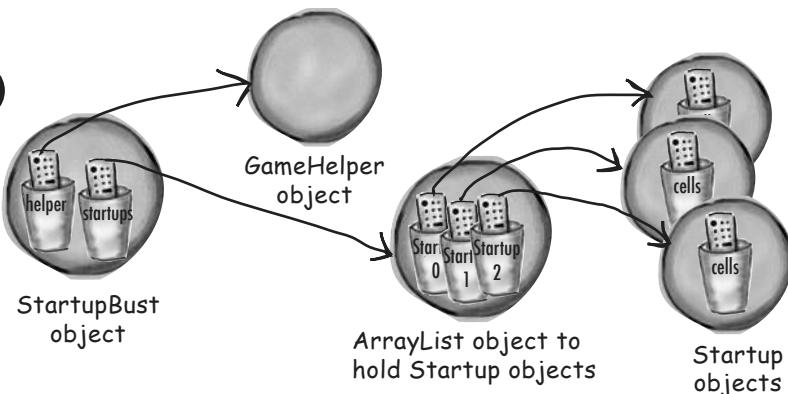


## 5 Objects:

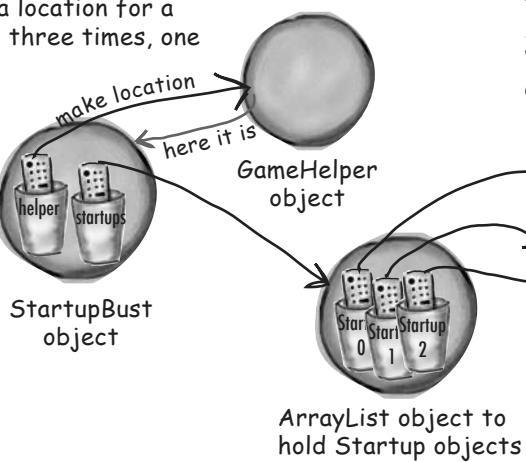


## Who does what in the StartupBust game (and when)



**4**

The `StartupBust` object creates three `Startup` objects (and puts them in the `ArrayList`).

**5**

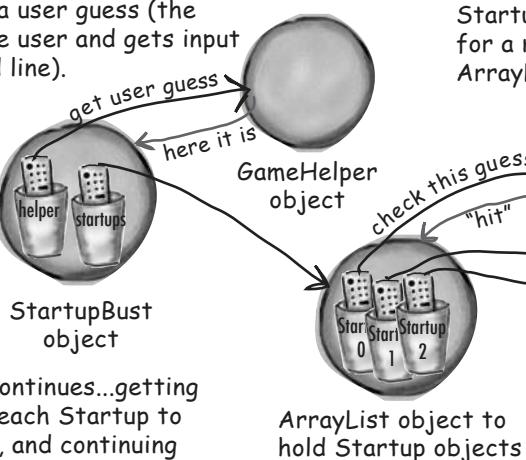
The `StartupBust` object gives each of the `Startup` objects a location (which the `StartupBust` got from the `helper` object) like "A2," "B2," etc. Each `Startup` object puts his own three location cells in an `ArrayList`.

`ArrayList` object (to hold Startup cell locations)

`ArrayList` object

`ArrayList` object

The `StartupBust` object asks the `helper` object for a user guess (the `helper` prompts the user and gets input from the command line).

**6**

The `StartupBust` object loops through the list of `Startups`, and asks each one to check the user guess for a match. Each `Startup` checks its locations `ArrayList` and returns a result ("hit," "miss," etc.).

`ArrayList` object (to hold Startup cell locations)

`ArrayList` object

`ArrayList` object

And so the game continues...getting user input, asking each `Startup` to check for a match, and continuing until all `Startups` are dead

## the StartupBust class (the game)

prep code   test code   real code

StartupBust
GameHelper helper ArrayList startups int numOfGuesses
setUpGame() startPlaying() checkUserGuess() finishGame()

## Prep code for the real StartupBust class

The StartupBust class has three main jobs: set up the game, play the game until the Startups are dead, and end the game. Although we could map those three jobs directly into three methods, we split the middle job (play the game) into *two* methods to keep the granularity smaller. Smaller methods (meaning smaller chunks of functionality) help us test, debug, and modify the code more easily.

## Variable Declarations

**DECLARE** and instantiate the *GameHelper* instance variable, named *helper*.

**DECLARE** and instantiate an *ArrayList* to hold the list of Startups (initially three). Call it *startups*.

**DECLARE** an int variable to hold the number of user guesses (so that we can give the user a score at the end of the game). Name it *numOfGuesses* and set it to 0.

## Method Declarations

**DECLARE** a *setUpGame()* method to create and initialize the Startup objects with names and locations. Display brief instructions to the user.

**DECLARE** a *startPlaying()* method that asks the player for guesses and calls the *checkUserGuess()* method until all the Startup objects are removed from play.

**DECLARE** a *checkUserGuess()* method that loops through all remaining Startup objects and calls each Startup object's *checkYourself()* method.

**DECLARE** a *finishGame()* method that prints a message about the user's performance, based on how many guesses it took to sink all of the Startup objects.

### METHOD: void *setUpGame()*

// make three Startup objects and name them

**CREATE** three Startup objects.

**SET** a name for each Startup.

**ADD** the Startups to *startups* (the *ArrayList*).

**REPEAT** with each of the Startup objects in the *startups* List:

**CALL** the *placeStartup()* method on the *helper* object, to get a randomly-selected location for this Startup (three cells, vertically or horizontally aligned, on a 7 X 7 grid).

**SET** the location for each Startup based on the result of the *placeStartup()* call.

END REPEAT

END METHOD

## Method implementations continued:

### METHOD: void startPlaying()

**REPEAT** while any Startups exist.

**GET** user input by calling the helper `getUserInput()` method.

**EVALUATE** the user's guess by `checkUserGuess()` method.

END REPEAT

END METHOD

### METHOD: void checkUserGuess(String userGuess)

// find out if there's a hit (and kill) on any Startup

**INCREMENT** the number of user guesses in the `numOfGuesses` variable.

**SET** the local `result` variable (a `String`) to "miss", assuming that the user's guess will be a miss.

**REPEAT** with each of the Startup objects in the `startups` List.

**EVALUATE** the user's guess by calling the Startup object's `checkYourself()` method.

**SET** the result variable to "hit" or "kill" if appropriate.

**IF** the result is "kill", **REMOVE** the Startup from the `startups` List.

END REPEAT

**DISPLAY** the `result` value to the user.

END METHOD

### METHOD: void finishGame()

**DISPLAY** a generic "game over" message, then:

**IF** number of user guesses is small,

**DISPLAY** a congratulations message.

**ELSE**

**DISPLAY** an insulting one.

END IF

END METHOD



### Sharpen your pencil

How should we go from prep code to the final code? First we start with test code, and then test and build up our methods bit by bit. We won't keep showing you test code in this book, so now it's up to you to think about what you'd need to know to test these

→ Yours to solve.

methods. And which method do you test and write first? See if you can work out some prep code for a set of tests. Prep code or even bullet points are good enough for this exercise, but if you want to try to write the *real* test code (in Java), knock yourself out.

## the StartupBust code (the game)

prep code test code real code

```
import java.util.ArrayList;

public class StartupBust {
    private GameHelper helper = new GameHelper();
    private ArrayList<Startup> startups = new ArrayList<Startup>();
    private int numOfGuesses = 0;

    private void setUpGame() {
        // first make some Startups and give them locations
        Startup one = new Startup();
        one.setName("poniez");
        Startup two = new Startup();
        two.setName("hacqi");
        Startup three = new Startup();
        three.setName("cabista");
        startups.add(one);
        startups.add(two);
        startups.add(three);
    } // close setUpGame method

    private void startPlaying() {
        while (!startups.isEmpty()) { // 7
            String userGuess = helper.getUserInput("Enter a guess"); // 8
            checkUserGuess(userGuess); // 9
        } // close while
        finishGame(); // 10
    } // close startPlaying method
}
```

- Declare and initialize the variables we'll need
- Get user input
- Ask the helper for a Startup location
- Print brief instructions for user
- Call the setter method on this Startup to give it the location you just got from the helper
- Repeat with each Startup in the list
- Call our own checkUserGuess method
- Call our own finishGame method
- Make three Startup objects, give 'em names, and stick 'em in the ArrayList
- As long as the Startup list is NOT empty



Annotate the code yourself!

Match the annotations at the bottom of each page with the numbers in the code. Write the number in the slot in front of the corresponding annotation.

You'll use each annotation just once, and you'll need all of the annotations.



**prep code** **test code** **real code**

```
private void checkUserGuess(String userGuess) {
    numOfGuesses++; (11)
    String result = "miss"; (12)

    for (Startup startupToTest : startups) { (13)
        result = startupToTest.checkYourself(userGuess); (14)

        if (result.equals("hit")) {
            break; (15)
        }
        if (result.equals("kill")) {
            startups.remove(startupToTest); (16)
            break;
        }
    } // close for

    System.out.println(result); (17)
} // close method
```

**Whatever you do,  
DON'T turn the  
page!**

**Not until you've  
finished this  
exercise.**

**Our version is on  
the next page.**



```
private void finishGame() {
    System.out.println("All Startups are dead! Your stock is now worthless");
    if (numOfGuesses <= 18) {
        System.out.println("It only took you " + numOfGuesses + " guesses.");
        System.out.println("You got out before your options sank.");
    } else {
        System.out.println("Took you long enough. " + numOfGuesses + " guesses.");
        System.out.println("Fish are dancing with your options");
    }
} // close method

public static void main(String[] args) {
    StartupBust game = new StartupBust(); (19)
    game.setUpGame(); (20)
    game.startPlaying(); (21)
} // close method
}
```

**(18)**

- Repeat with all Startups in the list
- This one's dead, so take it out of the Startups list then get out of the loop
- Increment the number of guesses the user has made
- Get out of the loop early, no point in testing the others
- Assume it's a 'miss,' unless told otherwise
- Tell the game object to start the main game play loop (keeps asking for user input and checking the guess)
- Print a message telling the user how they did in the game
- Ask the Startup to check the user guess, looking for a hit (or kill)
- Create the game object

— Print the result for the user

— Tell the game object to set up the game

you are here ▶ 147

## the StartupBust code (the game)

prep code test code real code

```
import java.util.ArrayList;

public class StartupBust {
    private GameHelper helper = new GameHelper();
    private ArrayList<Startup> startups = new ArrayList<Startup>();
    private int numOfGuesses = 0;

    private void setUpGame() {
        // first make some Startups and give them locations
        Startup one = new Startup();
        one.setName("poniez");
        Startup two = new Startup();
        two.setName("hacqi");
        Startup three = new Startup();
        three.setName("cabista");
        startups.add(one);
        startups.add(two);
        startups.add(three);

        System.out.println("Your goal is to sink three Startups.");
        System.out.println("poniez, hacqi, cabista");
        System.out.println("Try to sink them all in the fewest number of guesses");
    }

    for (Startup startup : startups) {
        ArrayList<String> newLocation = helper.placeStartup(3);
        startup.setLocationCells(newLocation);
    } // close for loop
} // close setUpGame method

private void startPlaying() {
    while (!startups.isEmpty()) { // As long as the Startup list is NOT empty (the ! means NOT, it's
        String userGuess = helper.getUserInput("Enter a guess"); // the same as (startups.isEmpty() == false).
        checkUserGuess(userGuess); // Get user input.
    } // close while
    finishGame(); // Call our own checkUserGuess method.
} // close startPlaying method
```

Declare and initialize the variables we'll need.

Make three Startup objects, give 'em names, and stick 'em in the ArrayList.

Print brief instructions for user.

Repeat with each Startup in the list.

Ask the helper for a Startup location (an ArrayList of Strings).

Call the setter method on this Startup to give it the location you just got from the helper.

As long as the Startup list is NOT empty (the ! means NOT, it's the same as (startups.isEmpty() == false).

Get user input.

Call our own checkUserGuess method.

Call our own finishGame method.

**prep code**   **test code**   **real code**

```

private void checkUserGuess(String userGuess) {
    numOfGuesses++;           ← Increment the number of guesses the user has made
    String result = "miss";   ← Assume it's a 'miss', unless told otherwise

    for (Startup startupToTest : startups) { ← Repeat with all Startups in the list
        result = startupToTest.checkYourself(userGuess); ← Ask the Startup to check the user
                                                        guess, looking for a hit (or kill)

        if (result.equals("hit")) { Get out of the loop early, no point
            break;               ← in testing the others
        }
        if (result.equals("kill")) {
            startups.remove(startupToTest);
            break;
        }
    } // close for

    System.out.println(result); ← Print the result for the user
} // close method

```

Print a message telling the user how they did in the game

```

private void finishGame() {
    System.out.println("All Startups are dead! Your stock is now worthless");
    if (numOfGuesses <= 18) {
        System.out.println("It only took you " + numOfGuesses + " guesses.");
        System.out.println("You got out before your options sank.");
    } else {
        System.out.println("Took you long enough. " + numOfGuesses + " guesses.");
        System.out.println("Fish are dancing with your options");
    }
} // close method

```

```

public static void main(String[] args) {
    StartupBust game = new StartupBust(); ← Create the game object
    game.setUpGame();                    ← Tell the game object to set up the game
    game.startPlaying();                ← Tell the game object to start the main
                                         game play loop (keeps asking for user
                                         input and checking the guess)
} // close method
}

```

# The final version of the Startup class

```

import java.util.ArrayList;

public class Startup {
    private ArrayList<String> locationCells;
    private String name;
}

public void setLocationCells(ArrayList<String> loc) { ←
    locationCells = loc;
}

public void setName(String n) { ← Your basic setter method
    name = n;
}

public String checkYourself(String userInput) {
    String result = "miss";
    int index = locationCells.indexOf(userInput); ←
    if (index >= 0) {
        locationCells.remove(index); ← Using ArrayList's remove() method to delete an entry.

        if (locationCells.isEmpty()) { ← Using the isEmpty() method to see if all
            result = "kill";           of the locations have been guessed
            System.out.println("Ouch! You sunk " + name + " : ( ");
        } else {
            result = "hit";           ← Tell the user when a Startup has been sunk.
            } // end if
        } // end outer if
    return result;
} // end method

} // close class

```

Startup's instance variables:  
 - an ArrayList of cell locations  
 - the Startup's name

A setter method that updates the Startup's location. (Random location provided by the GameHelper.placeStartup() method.)

The ArrayList indexOf() method in action! If the user guess is one of the entries in the ArrayList, indexOf() will return its ArrayList location. If not, indexOf() will return -1.

Using ArrayList's remove() method to delete an entry.

Using the isEmpty() method to see if all of the locations have been guessed

Tell the user when a Startup has been sunk.

Return: 'miss' or 'hit' or 'kill'.

# Super powerful Boolean expressions

So far, when we've used Boolean expressions for our loops or `if` tests, they've been pretty simple. We will be using more powerful boolean expressions in some of the Ready-Bake Code you're about to see, and even though we know you wouldn't peek, we thought this would be a good time to discuss how to energize your expressions.

## “And” and “Or” Operators (`&&`, `||`)

Let's say you're writing a `chooseCamera()` method, with lots of rules about which camera to select. Maybe you can choose cameras ranging from \$50 to \$1000, but in some cases you want to limit the price range more precisely. You want to say something like:

“If the price *range* is between \$300 **and** \$400, then choose X.”

```
if (price >= 300 && price < 400) {
    camera = "X";
}
```

Let's say that of the ten camera brands available, you have some logic that applies to only a few of the list:

```
if (brand.equals("A") || brand.equals("B")) {
    // do stuff for only brand A or brand B
}
```

Boolean expressions can get really big and complicated:

```
if ((zoomType.equals("optical") &&
    (zoomDegree >= 3 && zoomDegree <= 8)) ||
    (zoomType.equals("digital") &&
    (zoomDegree >= 5 && zoomDegree <= 12))) {
    // do appropriate zoom stuff
}
```

If you want to get *really* technical, you might wonder about the precedence of these operators. Instead of becoming an expert in the arcane world of precedence, we recommend that you **use parentheses** to make your code clear.

## Not equals (`!=` and `!`)

Let's say that you have a logic like “of the ten available camera models, a certain thing is *true for all but one*.”

```
if (model != 2000) {
    // do non-model 2000 stuff
}
```

or for comparing objects like strings...

```
if (!brand.equals("X")) {
    // do non-brand X stuff
}
```

## Short-Circuit Operators (`&&`, `||`)

The operators we've looked at so far, `&&` and `||`, are known as **short-circuit** operators. In the case of `&&`, the expression will be true only if *both* sides of the `&&` are true. So if the JVM sees that the left side of a `&&` expression is false, it stops right there! Doesn't even bother to look at the right side.

Similarly, with `||`, the expression will be true if *either* side is true, so if the JVM sees that the left side is true, it declares the entire statement to be true and doesn't bother to check the right side.

Why is this great? Let's say that you have a reference variable and you're not sure whether it's been assigned to an object. If you try to call a method using this null reference variable (i.e., no object has been assigned), you'll get a `NullPointerException`. So, try this:

```
if (refVar != null &&
    refVar.isValidType()) {
    // do 'got a valid type' stuff
}
```

## Non-Short-Circuit Operators (`&`, `|`)

When used in boolean expressions, the `&` and `|` operators act like their `&&` and `||` counterparts, except that they force the JVM to *always* check *both* sides of the expression. Typically, `&` and `|` are used in another context, for manipulating bits.

## Ready-Bake: GameHelper



# Ready-Bake Code

This is the helper class for the game. Besides the user input method (that prompts the user and reads input from the command line), the helper's Big Service is to create the cell locations for the Startups. We tried to keep it fairly small so you wouldn't have to type so much. And remember, you won't be able to compile the StartupBust game class until you have *this* class.

```
import java.util.*;  
  
public class GameHelper {  
    private static final String ALPHABET = "abcdefg";  
    private static final int GRID_LENGTH = 7;  
    private static final int GRID_SIZE = 49;  
    private static final int MAX_ATTEMPTS = 200;  
    static final int HORIZONTAL_INCREMENT = 1;           // A better way to represent these two  
    static final int VERTICAL_INCREMENT = GRID_LENGTH;   // things is an enum (see Appendix B)  
  
    private final int[] grid = new int[GRID_SIZE];  
    private final Random random = new Random();  
    private int startupCount = 0;  
  
    public String getUserInput(String prompt) {  
        System.out.print(prompt + ": ");  
        Scanner scanner = new Scanner(System.in);  
        return scanner.nextLine().toLowerCase();  
    } //end getUserInput  
  
    public ArrayList<String> placeStartup(int startupSize) {  
        // holds index to grid (0 - 48)  
        int[] startupCoords = new int[startupSize];  
        int attempts = 0;  
        boolean success = false;  
  
        startupCount++;  
        int increment = getIncrement();  
  
        while (!success & attempts++ < MAX_ATTEMPTS) {  
            int location = random.nextInt(GRID_SIZE);  
  
            for (int i = 0; i < startupCoords.length; i++) {  
                startupCoords[i] = location;  
                location += increment;  
            }  
            // System.out.println("Trying: " + Arrays.toString(startupCoords));  
  
            if (startupFits(startupCoords, increment)) {  
                success = coordsAvailable(startupCoords);  
            }  
        }  
        savePositionToGrid(startupCoords);  
        ArrayList<String> alphaCells = convertCoordsToAlphaFormat(startupCoords);  
        // System.out.println("Placed at: " + alphaCells);  
        return alphaCells;  
    } //end placeStartup
```

Note: For extra credit, you might try "un-commenting" the System.out.println's, just to watch it work! These print statements will let you "cheat" by giving you the location of the Startups, but it will help you test it.

This is the statement that tells you exactly where the Startup is located.



# Ready-Bake Code

## GameHelper class code continued...

```

private boolean startupFits(int[] startupCoords, int increment) {
    int finalLocation = startupCoords[startupCoords.length - 1];
    if (increment == HORIZONTAL_INCREMENT) {
        // check end is on same row as start
        return calcRowFromIndex(startupCoords[0]) == calcRowFromIndex(finalLocation);
    } else {
        return finalLocation < GRID_SIZE;                                // check end isn't off the bottom
    }
} //end startupFits
private boolean coordsAvailable(int[] startupCoords) {
    for (int coord : startupCoords) {                                // check all potential positions
        if (grid[coord] != 0) {                                         // this position already taken
            System.out.println("position: " + coord + " already taken.");
            return false;                                              // NO success
        }
    }
    return true;                                                       // there were no clashes, yay!
} //end coordsAvailable
private void savePositionToGrid(int[] startupCoords) {
    for (int index : startupCoords) {                                // mark grid position as 'used'
        grid[index] = 1;
    }
} //end savePositionToGrid
private ArrayList<String> convertCoordsToAlphaFormat(int[] startupCoords) {
    ArrayList<String> alphaCells = new ArrayList<String>();
    for (int index : startupCoords) {                                // for each grid coordinate
        String alphaCoords = getAlphaCoordsFromIndex(index); // turn it into an "a0" style
        alphaCells.add(alphaCoords);                            // add to a list
    }
    return alphaCells;                                              // return the "a0"-style coords
} // end convertCoordsToAlphaFormat
private String getAlphaCoordsFromIndex(int index) {
    int row = calcRowFromIndex(index);                                // get row value
    int column = index % GRID_LENGTH;                                 // get numeric column value
    String letter = ALPHABET.substring(column, column + 1); // convert to letter
    return letter + row;
} // end getAlphaCoordsFromIndex
private int calcRowFromIndex(int index) {
    return index / GRID_LENGTH;
} // end calcRowFromIndex
private int getIncrement() {
    if (startupCount % 2 == 0) {                                     // if EVEN Startup
        return HORIZONTAL_INCREMENT;                                // place horizontally
    } else {                                                        // else ODD
        return VERTICAL_INCREMENT;                                // place vertically
    }
} //end getIncrement
} //end class

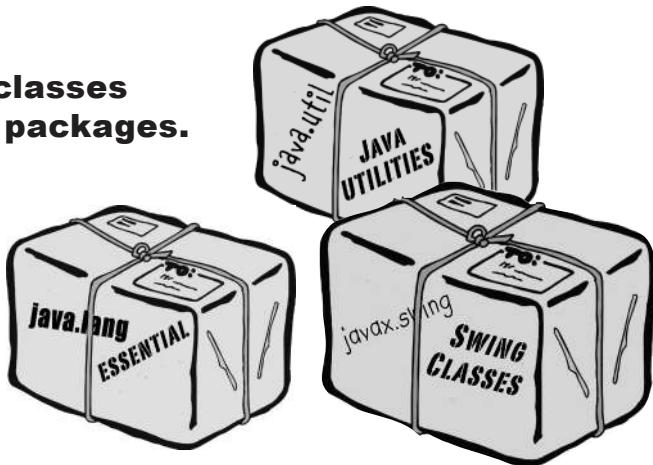
```

This code, and a basic test, is available in the GitHub repo, [https://oreil.ly/hfJava\\_3e\\_examples](https://oreil.ly/hfJava_3e_examples)

## Using the Library (the Java API)

You made it all the way through the StartupBust game, thanks to the help of ArrayList. And now, as promised, it's time to learn how to fool around in the Java library.

**In the Java API, classes are grouped into packages.**



**To use a class in the API, you have to know which package the class is in.**

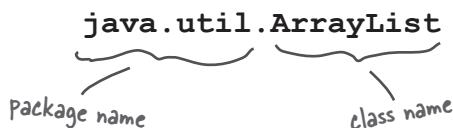
Every class in the Java library belongs to a package. The package has a name, like **javax.swing** (a package that holds some of the Swing GUI classes you'll learn about soon). ArrayList is in the package called **java.util**, which surprise surprise, holds a pile of *utility* classes. You'll learn a lot more about packages in Appendix B, including how to put your *own* classes into your *own* packages. For now, though, we're just looking to *use* some of the classes that come with Java.

Using a class from the API, in your own code, is simple. You just treat the class as though you wrote it yourself... as though you compiled it, and there it sits, waiting for you to use it. With one big difference: somewhere in your code you have to indicate the *full* name of the library class you want to use, and that means package name + class name.

Even if you didn't know it, ***you've already been using classes from a package.*** System (System.out.println), String, and Math (Math.random()) all belong to the **java.lang** package.

## You have to know the full name\* of the class you want to use in your code.

ArrayList is not the *full* name of ArrayList, just as Kathy isn't a full name (unless it's like Madonna or Cher, but we won't go there). The full name of ArrayList is actually:



**You have to tell Java which ArrayList you want to use. You have two options:**

### IMPORT

**A**

Put an import statement at the top of your source code file:

```
import java.util.ArrayList;
public class MyClass { ... }
```

**OR**

### TYPE

**B**

Type the full name everywhere in your code. Each time you use it.  
*Everywhere* you use it.

When you declare and/or instantiate it:

```
java.util.ArrayList<Dog> list = new java.util.ArrayList<Dog>();
```

When you use it as an argument type:

```
public void go(java.util.ArrayList<Dog> list) { }
```

When you use it as a return type:

```
public java.util.ArrayList<Dog> foo() { ... }
```

\*Unless the class is in the `java.lang` package.

## there are no Dumb Questions

**Q:** Why does there have to be a full name? Is that the only purpose of a package?

**A:** Packages are important for three main reasons. First, they help the overall organization of a project or library. Rather than just having one horrendously large pile of classes, they're all grouped into packages for specific kinds of functionality (like GUI or data structures or database stuff, etc.).

Second, packages give you a name-scoping, to help prevent collisions if you and 12 other programmers in your company all decide to make a class with the same name. If you have a class named Set and someone else (including the Java API) has a class named Set, you need some way to tell the JVM which Set class you're trying to use.

Third, packages provide a level of security, because you can restrict the code you write so that only other classes in the same package can access it. The details are in Appendix B.

**Q:** OK, back to the name collision thing. How does a full name really help? What's to prevent two people from giving a class the same package name?

**A:** Java has a naming convention that usually prevents this from happening, as long as developers adhere to it.

### BULLET POINTS

- **ArrayList** is a class in the Java API.
- To put something into an ArrayList, use **add()**.
- To remove something from an ArrayList use **remove()**.
- To find out where something is (and if it is) in an ArrayList, use **indexOf()**.
- To find out if an ArrayList is empty, use **isEmpty()**.
- To get the size (number of elements) in an ArrayList, use the **size() method**.
- To get the **length** (number of elements) in a regular old array, remember, you use the **length variable**.
- An ArrayList **resizes dynamically** to whatever size is needed. It grows when objects are added, and it **shrinks** when objects are removed.
- You declare the type of the array using a **type parameter**, which is a type name in angle brackets. Example: `ArrayList<Button>` means the ArrayList will be able to hold only objects of type Button (or subclasses of Button as you'll learn in the next couple of chapters).
- Although an ArrayList holds objects and not primitives, the compiler will automatically “wrap” (and “unwrap” when you take it out) a primitive into an Object and place that object in the ArrayList instead of the primitive. (More on this feature later in the book.)
- Classes are grouped into packages.
- A class has a full name, which is a combination of the package name and the class name. Class ArrayList is really `java.util.ArrayList`.
- To use a class in a package other than `java.lang`, you must tell Java the full name of the class.
- You can either use an import statement at the top of your source code, or you can type the full name every place you use the class in your code.

## there are no Dumb Questions

**Q:** Does `import` make my class bigger? Does it actually compile the imported class or package into my code?

**A:** Perhaps you're a C programmer? An `import` is not the same as an `include`. So the answer is no and no. Repeat after me: "an `import` statement saves you from typing." That's really it. You don't have to worry about your code becoming bloated, or slower, from too many imports. An `import` is simply the way you give Java the *full name of a class*.

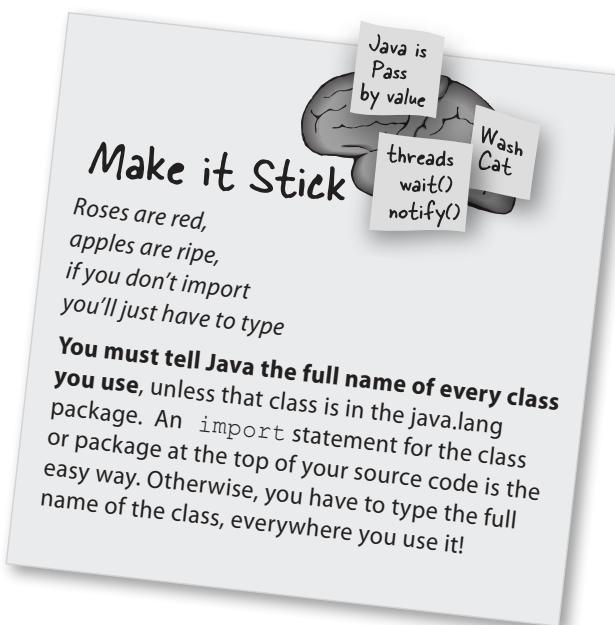
**Q:** OK, how come I never had to import the `String` class? Or `System`?

**A:** Remember, you get the `java.lang` package sort of "pre-imported" for free. Because the classes in `java.lang` are so fundamental, you don't have to use the full name. There is only one `java.lang.String` class and one `java.lang.System` class, and Java darn well knows where to find them.

**Q:** Do I have to put my own classes into packages? How do I do that? Can I do that?

**A:** In the real world (which you should try to avoid), yes, you *will* want to put your classes into packages. We'll get into that in detail in Appendix B. For now, we won't put our code examples in a package.\*

\*But when you look at the code in the repo ([https://oreil.ly/hfJava\\_3e\\_examples](https://oreil.ly/hfJava_3e_examples)), you'll see we put the classes into packages.



**One more time, in the unlikely event that you don't already have this down:**



*"Good to know there's an ArrayList in the java.util package. But by myself, how would I have figured that out?"*

- Julia, 31, hand model

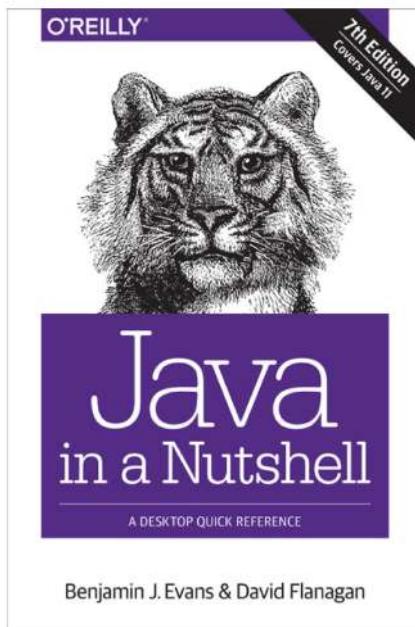


# How to discover the API

Two things you want to know:

- 1 **What features are available in the library? (Which classes?)**
- 2 **How do you use these features? (Once you find a class, how do you know what it can do?)**

## 1 Browse a book



## 2 Use the HTML API docs

### Java® Platform, Standard Edition & Java Development Kit Version 17 API Specification

This document is divided into two sections:

#### Java SE

The Java Platform, Standard Edition (Java SE) APIs define the core Java platform

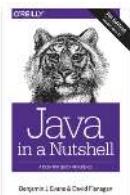
#### JDK

The Java Development Kit (JDK) APIs are specific to the JDK and will not necessarily

All Modules	Java SE	JDK	Other Modules
Module	Description		
java.base	Defines the foundational APIs of the Java SE Platform.		
java.compiler	Defines the Language Model, Annotation Processing, a		
java.datatransfer	Defines the API for transferring data between and with		

<https://docs.oracle.com/en/java/javase/17/docs/api/index.html>

# 1 Browse a book



Flipping through a reference book is a good way to find out what's in the Java library. You can easily stumble on to a package or class that looks useful just by browsing pages.

**O'REILLY**

8. Working with Java Collections

12h 3m remaining

## The List Interface

A `List` is an ordered collection of objects. Each element of a list has a position in the list, and the `List` interface defines methods to query or set the element at a particular position, or *index*. In this respect, a `List` is like an array whose size changes as needed to accommodate the number of elements it contains. Unlike sets, lists allow duplicate elements.

In addition to its index-based `get()` and `set()` methods, the `List` interface defines methods to add or remove an element at a particular index and also defines methods to return the index of the first or last occurrence of a particular value in the list. The `add()` and `remove()` methods inherited from `Collection` are defined to append to the list and to remove the first occurrence of the specified value from the list. The inherited `addAll()` appends all elements in the specified collection to the end of the list, and another version inserts the elements at a specified index. The `retainAll()` and `removeAll()` methods behave as they do for any `Collection`, retaining or removing multiple occurrences of the same value, if needed.

The `List` interface does not define methods that operate on a range of list indexes. Instead, it defines a single `subList()` method that returns a `List` object that represents just the specified range of the original list. The sublist is backed by the parent list, and any changes made to the sublist are immediately visible in the parent list. Examples of `subList()` and the other basic `List` manipulation methods are shown here:

```
// Create lists to work with
List<String> l = new ArrayList<String>(Arrays.asList(args));
List<String> words = Arrays.asList("hello", "world");
List<String> words2 = List.of("hello", "world");

// Querying and setting elements by index
```

## 2 Use the HTML API docs

Java comes with a fabulous set of online docs called, strangely, the Java API. You (or your IDE) can also download the docs to have on your hard drive just in case your internet connection fails at the Worst Possible Moment.

The API docs are the best reference for getting more details about what's in a package, and what the classes and interfaces in the package provide (e.g., in terms of methods and functionality).

**The docs look different depending upon the version of Java you're using  
Make sure you're looking at the docs for your version of Java!**

### Java 8 and earlier

<https://docs.oracle.com/javase/8/docs/api/index.html>

Java version. This is Java 8 SE

Scroll through the packages and select one (click it) to restrict the list in the lower frame to only classes from that package.

Scroll through the classes and select one (click it) to choose the class that will fill the main browser frame.

Package	Description
java.applet	Provides the classes necessary to create an applet and the classes an applet uses to communicate with its applet context.
java.awt	Contains all of the classes for creating user interfaces and for painting graphics and images.
java.awt.color	Provides classes for color spaces.
java.awt.datatransfer	Provides interfaces and classes for transferring data between and within applications.
java.awt.dnd	Drag and Drop is a direct manipulation gesture found in many Graphical User

You can navigate these docs:

- **Top down:** find a package you're interested in from the list in the top left and drill down.
- **Class-first:** find the class you want to know more about in the list in the bottom left, and click it.

The main panel will show you the details of whatever you're looking at. If you select a package, it will give summary information about that package and a list of the classes and interfaces.

If you select a class, it will show you a description of the class, and details of all the methods in the class, what they do, and how to use them.

## Java 9 and later

Java 9 introduced the Java Module System, which we're not going to cover in this book. What you do need to know to understand the docs is that the JDK is now split into *modules*. These modules group together related packages. This can make it easier to find the classes that interest you, because they're grouped by function. All of the classes we've covered in this book so far are in the **java.base** module; this contains core Java packages like `java.lang` and `java.util`.

Slightly different URL to the older docs  
<https://docs.oracle.com/en/java/javase/17/docs/api/index.html>

Java 17 is the current Long Term Support (LTS) version at the time of writing.

Search for a specific method/ class/ package/ module by typing it here. You'll see a drop-down of suggestions.

This document is divided into two sections:

- Java SE**: The Java Platform, Standard Edition (Java SE) APIs define the core Java platform for general-purpose computing. These APIs are in modules whose names start with `java`.
- JDK**: The Java Development Kit (JDK) APIs are specific to the JDK and will not necessarily be available in all implementations of the Java SE Platform. These APIs are in modules whose names start with `jdk`.

All Modules	Java SE	JDK	Other Modules
<b>Module</b>	<b>Description</b>		
<code>java.base</code>	Defines the foundational APIs of the Java SE Platform.		
<code>java.compiler</code>	Defines the Language Model, Annotation Processing, and Java Compiler APIs.		
<code>java.datatransfer</code>	Defines the API for transferring data between and within applications.		
<code>java.desktop</code>	Defines the AWT and Swing user interface toolkits, plus APIs for accessibility, audio, imaging, printing, and JavaBeans.		

The Java platform is now broken into a number of modules, which are listed on the home page of the docs.

We're mostly only interested in `java.base`. When we get to the Swing GUI, we'll care about `java.desktop` as well.

You can navigate these docs:

- Top down:** find a module that looks like it covers the functionality you want, see its packages, and drill down from a package into its classes.
- Search:** Use the search in the top right to go directly to the method, class, package, or module you want to read about.

When you've selected a module, you can see a list of all its packages and a description of what each package is for.

**Module java.base**

Defines the foundational APIs of the Java SE Platform.

**Providers:**  
 The JDK implementation of this module provides an implementation of the jrt file system provider to enable resource files in a run-time image. The jrt file system can be created by calling `FileSystems.newFileSystem(URI.create("jrt:/"))`.

**Module Graph:**

- `java.base`

**Tool Guides:**  
`java launcher`, `keytool`

**Since:**  
 9

Packages	
<b>Exports</b>	
<b>Package</b>	<b>Description</b>
<code>java.io</code>	Provides for system input and output through data streams, serializati
<code>java.lang</code>	Provides classes that are fundamental to the design of the Java progra
<code>java.lang.annotation</code>	Provides library support for the Java programming language annotati

when arrays aren't enough

## Using the class documentation

Whichever version of the Java docs you're using, they all have a similar layout for showing information about a specific class. This is where the juicy details are.

Let's say you were browsing through the reference book and found a class called ArrayList, in java.util. The book tells you a little about it, enough to know that this is indeed what you want to use, but you still need to know more about the methods. In the reference book, you'll find the method indexOf(). But if all you knew is that there is a method called indexOf() that takes an object and returns the index (an int) of that object, you still need to know one crucial thing: what happens if the object is not in the ArrayList? Looking at the method signature alone won't tell you how that works. But the API docs will (most of the time, anyway). The API docs tell you that the indexOf() method returns a -1 if the object parameter is not in the ArrayList. So now we know we can use it both as a way to check if an object is even in the ArrayList, and to get its index at the same time, if the object was there. But without the API docs, we might have thought that the indexOf() method would blow up if the object wasn't in the ArrayList.

The screenshot shows the Java SE 17 & JDK 17 API documentation for the ArrayList class. The top navigation bar includes links for OVERVIEW, MODULE, PACKAGE, CLASS (which is highlighted), USE, TREE, PREVIEW, NEW, DEPRECATED, INDEX, and HELP. A search bar is also present. The main content area has tabs for SUMMARY, NESTED, FIELD, CONSTR, and METHOD. Below these tabs, there are links for DETAIL, FIELD, CONSTR, and METHOD. A search input field is located at the bottom of the main content area. The page title is "Constructor Summary". The "Constructors" section lists three constructors:

Constructor	Description
ArrayList()	Constructs an empty list with an initial capacity of ten.
ArrayList(int initialCapacity)	Constructs an empty list with the specified initial capacity.
ArrayList(Collection<? extends E> c)	Constructs a list containing the elements of the specified collection, in the order they are returned by the collection's iterator.

A callout bubble on the left side of the constructor table says: "See the details of the current package (java.util in this case) by selecting 'Package'."

The page title is then changed to "Method Summary". The "All Methods" tab is selected. The table lists several methods:

Modifier and Type	Method	Description
void	add(int index, E element)	Inserts the specified element at the specified position in this list.
boolean	add(E e)	Appends the specified element to the end of this list.
boolean	addAll(int index, Collection<? extends E> c)	Inserts all of the elements in the specified collection into this list, starting at the specified position.
boolean	addAll(Collection<? extends E> c)	Appends all of the elements in the specified collection to the end of this list. They are returned by the collection's iterator.
void	clear()	Removes all of the elements from this list.
Object	clone()	Returns a shallow copy of this ArrayList instance.
boolean	contains(Object o)	Returns true if this list contains the specified element.

A callout bubble on the right side of the method table says: "This is where all the good stuff is. You can scroll through the methods for a brief summary or click on a method to get full details."

**In Chapters 11 and 12, you'll see how we use the API docs to figure out how to use the Java Libraries.**



## Code Magnets

Can you reconstruct the code snippets to make a working Java program that produces the output listed below? **NOTE:** To do this exercise, you need one NEW piece of info—if you look in the API for ArrayList, you'll find a *second* add method that takes two arguments:

`add(int index, Object o)`

It lets you specify to the ArrayList where to put the object you're adding.

```
public static void printList(ArrayList<String> list) {
```

`a.remove(2);`

`printList(a);`

```
a.add(0, "zero");
a.add(1, "one");
```

`printList(a);`

```
if (a.contains("two")) {
    a.add("2.2");
}
```

`a.add(2, "two");`

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

`System.out.print(element + " " );`

```
}
```

`System.out.println();`

```
if (a.contains("three")) {
    a.add("four");
}
```

```
public class ArrayListMagnet {
```

```
if (a.indexOf("four") != 4) {
    a.add(4, "4.2");
}
```

`import java.util.ArrayList;`

`printList(a);`

```
ArrayList<String> a = new ArrayList<String>();
```

File Edit Window Help Dance

```
% java ArrayListMagnet
zero one two three
zero one three four
zero one three four 4.2
zero one three four 4.2
```

```
for (String element : list) {
```

}

```
a.add(3, "three");
printList(a);
```

## puzzle: crossword



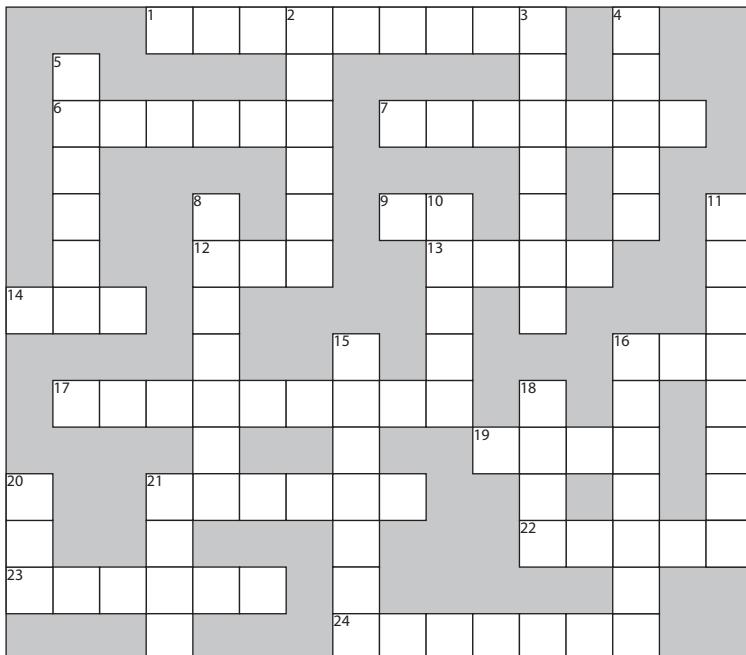
# JavaCross

How does this crossword puzzle help you learn Java? Well, all of the words **are** Java related (except one red herring).

**Hint:** When in doubt, remember ArrayList.

### Across

1. I can't behave
6. Or, in the courtroom
7. Where it's at baby
9. A fork's origin
12. Grow an ArrayList
13. Wholly massive
14. Value copy
16. Not an object
17. An array on steroids
19. Extent
21. 19's counterpart
22. Spanish geek snacks (Note: This has nothing to do with Java.)
23. For lazy fingers
24. Where packages roam



### Down

2. Where the Java action is
3. Addressable unit
4. 2nd smallest
5. Fractional default
8. Library's grandest
10. Must be low density
11. He's in there somewhere
15. As if
16. dearth method
18. What shopping and arrays have in common
20. Library acronym
21. What goes around

### More Hints:

- |      |                |                        |                    |                    |                     |                    |                    |              |                      |                                       |                    |            |                                       |                   |                      |                     |                     |                   |            |                     |           |                              |                      |                  |
|------|----------------|------------------------|--------------------|--------------------|---------------------|--------------------|--------------------|--------------|----------------------|---------------------------------------|--------------------|------------|---------------------------------------|-------------------|----------------------|---------------------|---------------------|-------------------|------------|---------------------|-----------|------------------------------|----------------------|------------------|
| Down | 1. 8 varieties | 2. What's overridable? | 3. Think ArrayList | 4. & 10. Primitive | 5. Common primitive | 6. Think ArrayList | 7. Think ArrayList | 8. Varieties | 9. 18. He's making a | 10. Not about Java—Spanish appetizers | 11. Array's extent | 12. Arrays | 13. Not about Java—Spanish appetizers | 14. He's making a | 15. Common primitive | 16. Think ArrayList | 17. Think ArrayList | 18. He's making a | 19. Extent | 20. Library acronym | 21. As if | 22. Where the Java action is | 23. Addressable unit | 24. 2nd smallest |
|------|----------------|------------------------|--------------------|--------------------|---------------------|--------------------|--------------------|--------------|----------------------|---------------------------------------|--------------------|------------|---------------------------------------|-------------------|----------------------|---------------------|---------------------|-------------------|------------|---------------------|-----------|------------------------------|----------------------|------------------|

→ Answers on page 166.



## Exercise Solutions

File Edit Window Help Dance

```
% java ArrayListMagnet
zero one two three
zero one three four
zero one three four 4.2
zero one three four 4.2
```

## Code Magnets

(from page 163)

```
import java.util.ArrayList;

public class ArrayListMagnet {
    public static void main(String[] args) {
        ArrayList<String> a = new ArrayList<String>();
        a.add(0, "zero");
        a.add(1, "one");
        a.add(2, "two");
        a.add(3, "three");
        printList(a);

        if (a.contains("three")) {
            a.add("four");
        }
        a.remove(2);
        printList(a);

        if (a.indexOf("four") != 4) {
            a.add(4, "4.2");
        }
        printList(a);

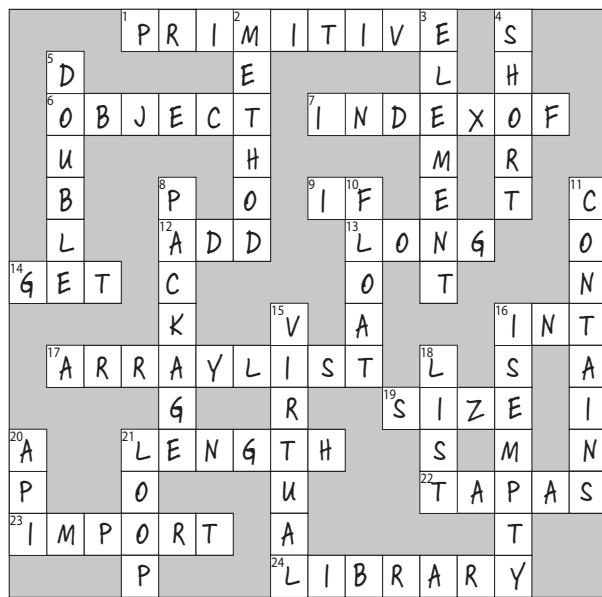
        if (a.contains("two")) {
            a.add("2.2");
        }
        printList(a);
    }

    public static void printList(ArrayList<String> list) {
        for (String element : list) {
            System.out.print(element + " ");
        }
        System.out.println();
    }
}
```



# JavaCross

(from page 164)



Write your OWN set of clues! Look at each word, and try to write your own clues. Try making them easier, or harder, or more technical than the ones we have.

### Across

1. \_\_\_\_\_
6. \_\_\_\_\_
7. \_\_\_\_\_
9. \_\_\_\_\_
12. \_\_\_\_\_
13. \_\_\_\_\_
14. \_\_\_\_\_
16. \_\_\_\_\_
17. \_\_\_\_\_
19. \_\_\_\_\_
21. \_\_\_\_\_
22. \_\_\_\_\_
23. \_\_\_\_\_
24. \_\_\_\_\_

### Down

2. \_\_\_\_\_
3. \_\_\_\_\_
4. \_\_\_\_\_
5. \_\_\_\_\_
8. \_\_\_\_\_
10. \_\_\_\_\_
11. \_\_\_\_\_
15. \_\_\_\_\_
16. \_\_\_\_\_
18. \_\_\_\_\_
20. \_\_\_\_\_
21. \_\_\_\_\_

# Better Living in Objectville



We were underpaid,  
overworked coders 'till we  
tried the Polymorphism Plan. But  
thanks to the Plan, our future is  
bright. Yours can be too!

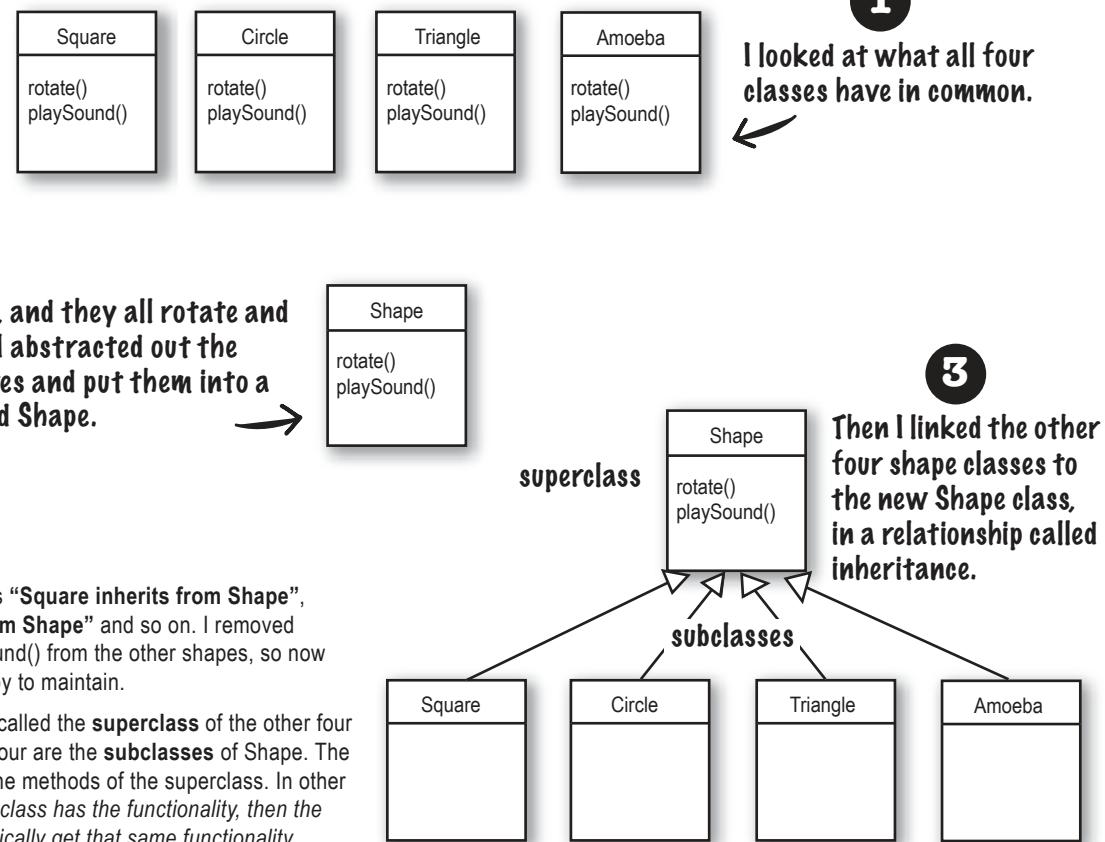
**Plan your programs with the future in mind.** If there were a way to write Java code such that you could take more vacations, how much would it be worth to you? What if you could write code that someone *else* could extend, **easily**? And if you could write code that was flexible, for those pesky last-minute spec changes, would that be something you'd be interested in? Then this is your lucky day. For just three easy payments of 60 minutes time, you can have all this. When you get on the Polymorphism Plan, you'll learn the 5 steps to better class design, the 3 tricks to polymorphism, the 8 ways to make flexible code, and if you act now—a bonus lesson on the 4 tips for exploiting inheritance. Don't delay, an offer this good will give you the design freedom and programming flexibility you deserve. It's quick, it's easy, and it's available now. Start today, and we'll throw in an extra level of abstraction!

## Chair Wars Revisited...

Remember way back in Chapter 2, when Laura (procedural programmer) and Brad (OO developer) were vying for the Aeron chair? Let's look at a few pieces of that story to review the basics of inheritance.

**LAURA:** You've got duplicated code! The rotate procedure is in all four Shape things. It's a stupid design. You have to maintain four different rotate "methods." How can that ever be good?

**BRAD:** Oh, I guess you didn't see the final design. Let me show you how OO **inheritance** works, Laura.

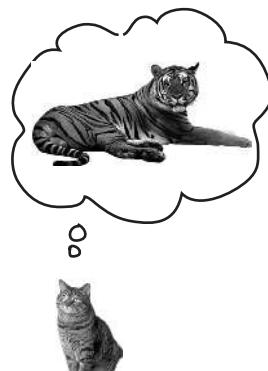
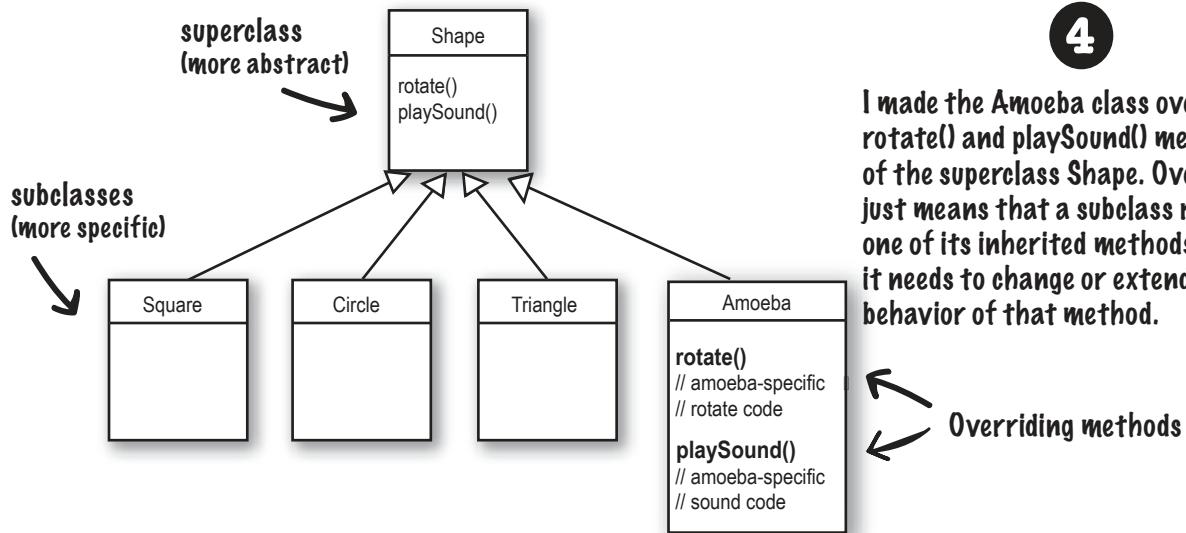


## What about the Amoeba rotate()?

**LAURA:** Wasn't that the whole problem here—that the Amoeba shape had a completely different rotate and playSound procedure?

How can Amoeba do something different if it *inherits* its functionality from the Shape class?

**BRAD:** That's the last step. The Amoeba class *overrides* any methods of the Shape class that need specific amoeba behavior. Then at runtime, the JVM knows exactly which rotate() method to run when someone tells the Amoeba to rotate.



How would you represent a house cat and a tiger, in an inheritance structure? Is a domestic cat a specialized version of a tiger? Which would be the subclass, and which would be the superclass? Or are they both subclasses to some *other* class?

How would you design an inheritance structure? What methods would be overridden?

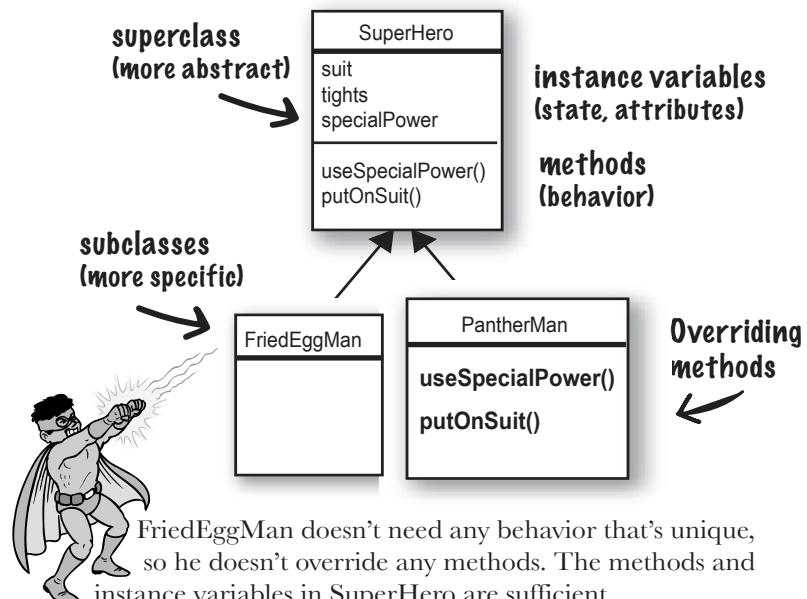
Think about it. Before you turn the page.

# Understanding Inheritance

When you design with inheritance, you put common code in a class and then tell other more specific classes that the common (more abstract) class is their superclass. When one class inherits from another, **the subclass inherits from the superclass**.

In Java, we say that the **subclass extends the superclass**.

An inheritance relationship means that the subclass inherits the **members** of the superclass. When we say “members of a class,” we mean the instance variables and methods. For example if PantherMan is a subclass of SuperHero, the PantherMan class automatically inherits the instance variables and methods common to all superheroes including suit, tights, specialPower, useSpecialPower(), and so on. But the PantherMan **subclass can add new methods and instance variables of its own, and it can override the methods it inherits from the superclass** SuperHero.



FriedEggMan doesn't need any behavior that's unique, so he doesn't override any methods. The methods and instance variables in SuperHero are sufficient.

PantherMan, though, has specific requirements for his suit and special powers, so `useSpecialPower()` and `putOnSuit()` are both overridden in the PantherMan class.

**Instance variables are not overridden** because they don't need to be. They don't define any special behavior, so a subclass can give an inherited instance variable any value it chooses. PantherMan can set his inherited `tights` to purple, while FriedEggMan sets his to white.

## An inheritance example:

```
public class Doctor {
    boolean worksAtHospital;

    void treatPatient() {
        // perform a checkup
    }
}

public class FamilyDoctor extends Doctor {
    boolean makesHouseCalls;

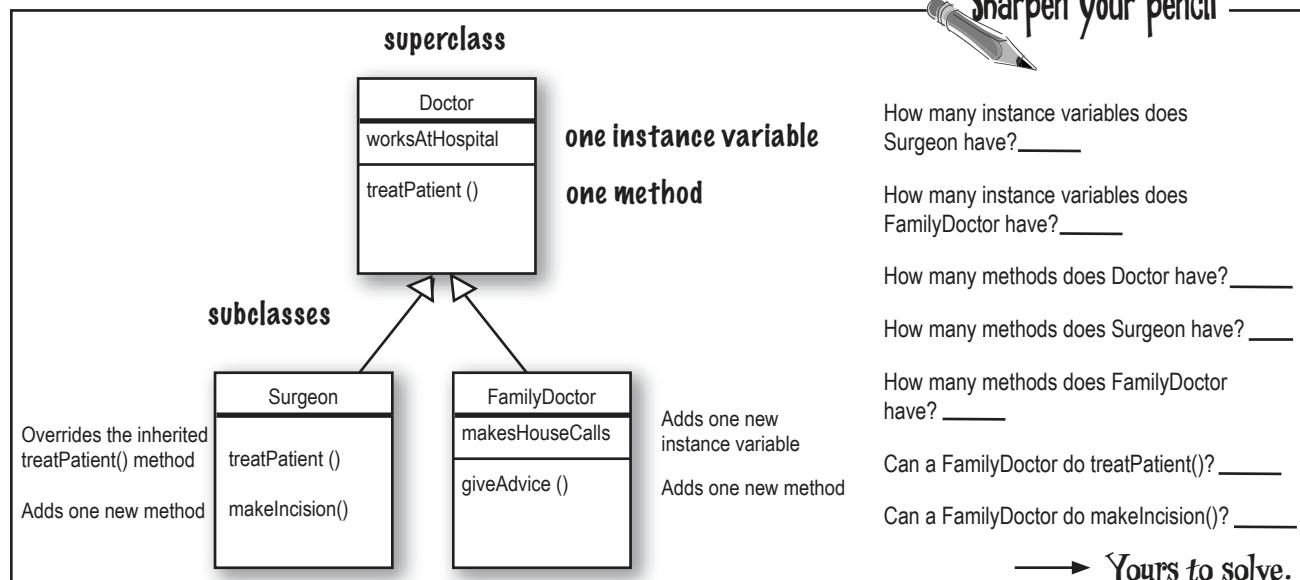
    void giveAdvice() {
        // give homespun advice
    }
}

public class Surgeon extends Doctor {
    void treatPatient() {
        // perform surgery
    }

    void makeIncision() {
        // make incision (yikes!)
    }
}
```



I inherited my procedures so I didn't bother with medical school. Relax, this won't hurt a bit. (now where did I put that power saw...)



# Let's design the inheritance tree for an Animal simulation program

Imagine you're asked to design a simulation program that lets the user throw a bunch of different animals into an environment to see what happens. We don't have to code the thing now; we're mostly interested in the design.

We've been given a list of *some* of the animals that will be in the program, but not all. We know that each animal will be represented by an object and that the objects will move around in the environment, doing whatever it is that each particular type is programmed to do.

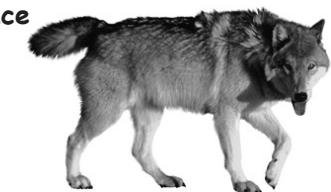
***And we want other programmers to be able to add new kinds of animals to the program at any time.***

First we have to figure out the common, abstract characteristics that all animals have, and build those characteristics into a class that all animal classes can extend.

- 1 Look for objects that have common attributes and behaviors.

**What do these six types have in common? This helps you to abstract out behaviors. (step 2)**

**How are these types related? This helps you to define the inheritance tree relationships (steps 4-5)**



# Using inheritance to avoid duplicating code in subclasses

We have five ***instance variables***:

***picture*** – the filename representing the JPEG of this animal.

***food*** – the type of food this animal eats. Right now, there can be only two values: *meat* or *grass*.

***hunger*** – an int representing the hunger level of the animal. It changes depending on when (and how much) the animal eats.

***boundaries*** – values representing the height and width of the “space” (for example, 640 x 480) that the animals will roam around in.

***location*** – the X and Y coordinates for where the animal is in the space.

We have four ***methods***:

***makeNoise()*** – behavior for when the animal is supposed to make noise.

***eat()*** – behavior for when the animal encounters its preferred food source, *meat* or *grass*.

***sleep()*** – behavior for when the animal is considered asleep.

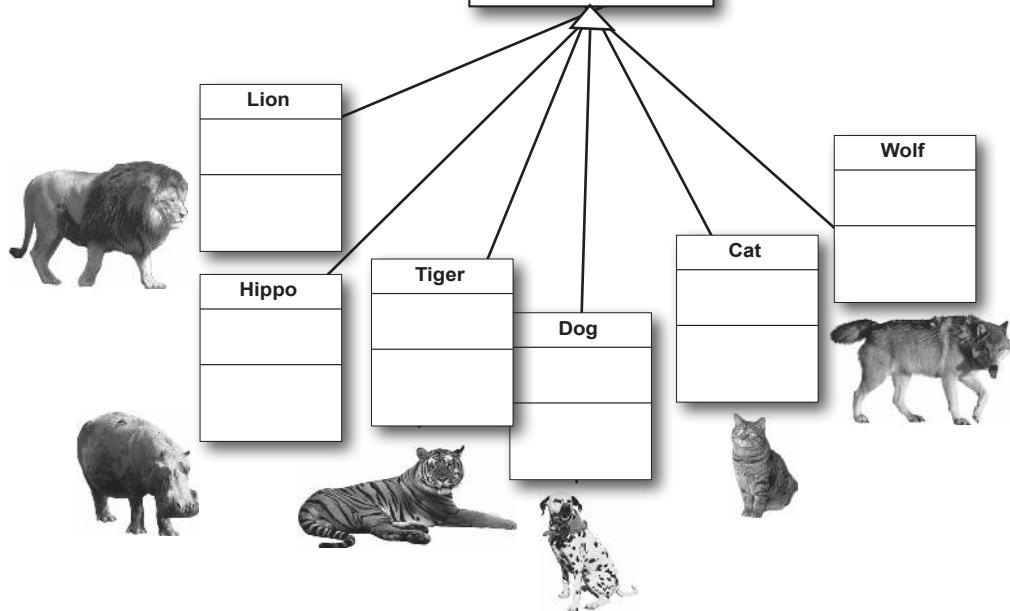
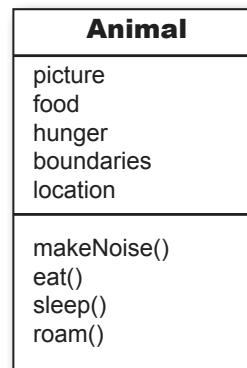
***roam()*** – behavior for when the animal is not eating or sleeping (probably just wandering around waiting to bump into a food source or a boundary).

2

Design a class that represents the common state and behavior.

These objects are all animals, so we'll make a common superclass called ***Animal***.

We'll put in methods and instance variables that all animals might need.



# Do all animals eat the same way?

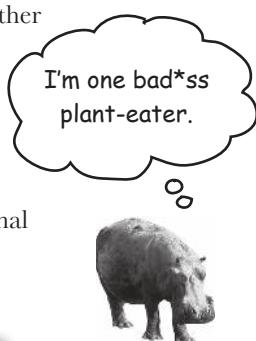
Assume that we all agree on one thing: the instance variables will work for *all* Animal types. A lion will have his own value for picture, food (we're thinking *meat*), hunger, boundaries, and location. A hippo will have different *values* for his instance variables, but he'll still have the same variables that the other Animal types have. Same with dog, tiger, and so on. But what about *behavior*?

## Which methods should we override?

Does a lion make the same **noise** as a dog? Does a cat **eat** like a hippo? Maybe in *your* version, but in ours, eating and making noise are Animal-type-specific. We can't figure out how to code those methods in such a way that they'd work for any animal. OK, that's not true. We could write the `makeNoise()` method, for example, so that all it does is play a sound file defined in an instance variable for that type, but that's not very specialized. Some animals might make different noises for different situations (like one for eating, and another when bumping into an enemy, etc.)

So just as with the Amoeba overriding the Shape class `rotate()` method, to get more amoeba-specific (in other words, *unique*) behavior, we'll have to do the same for our Animal subclasses.

Animal
picture
food
hunger
boundaries
location
makeNoise() eat() sleep() roam()



3

Decide if a subclass needs behaviors (method implementations) that are specific to that particular subclass type.

Looking at the Animal class, we decide that `eat()` and `makeNoise()` should be overridden by the individual subclasses.



We better override these two methods, `eat()` and `makeNoise()`, so that each animal type can define its own specific behavior for eating and making noise. For now, it looks like `sleep()` and `roam()` can stay generic.

# Looking for more inheritance opportunities

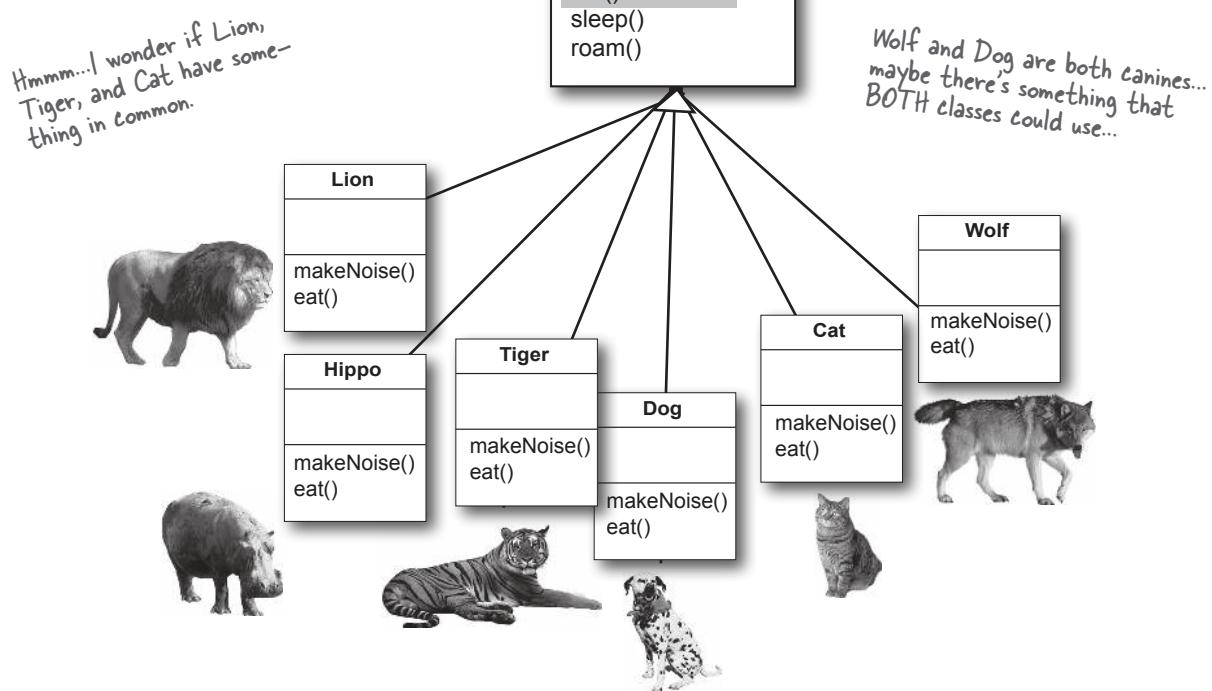
The class hierarchy is starting to shape up. We have each subclass override the `makeNoise()` and `eat()` methods so that there's no mistaking a Dog bark from a Cat meow (quite insulting to both parties). And a Hippo won't eat like a Lion.

But perhaps there's more we can do. We have to look at the subclasses of `Animal` and see if two or more can be grouped together in some way, and given code that's common to only *that* new group. Wolf and Dog have similarities. So do Lion, Tiger, and Cat.

4

Look for more opportunities to use abstraction, by finding two or more *subclasses* that might need common behavior.

We look at our classes and see that Wolf and Dog might have some behavior in common, and the same goes for Lion, Tiger, and Cat.



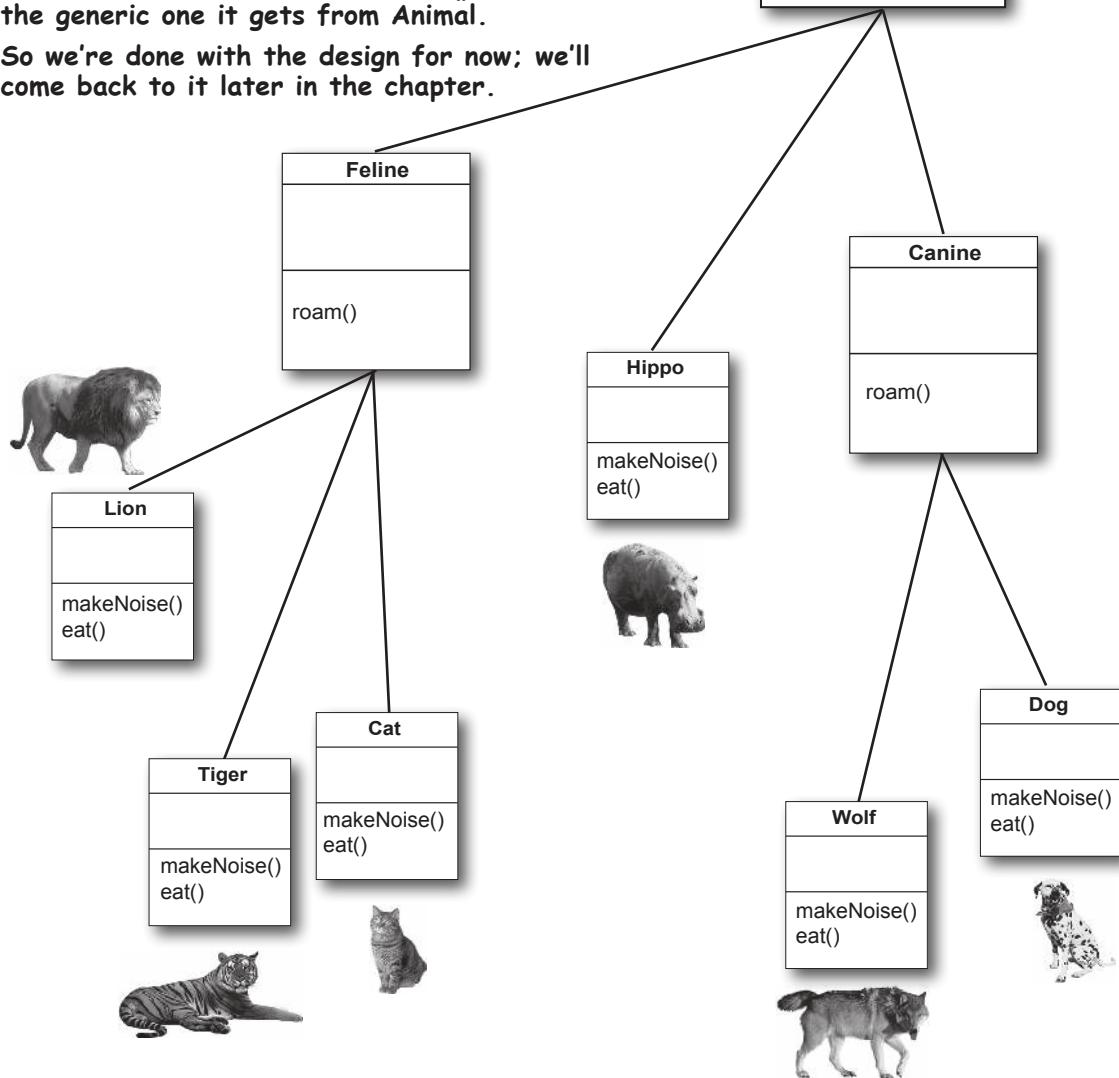
## 5 Finish the class hierarchy

Since animals already have an organizational hierarchy (the whole kingdom, genus, phylum thing), we can use the level that makes the most sense for class design. We'll use the biological "families" to organize the animals by making a Feline class and a Canine class.

We decide that Canines could use a common `roam()` method, because they tend to move in packs. We also see that Felines could use a common `roam()` method, because they tend to avoid others of their own kind. We'll let Hippo continue to use its inherited `roam()` method—the generic one it gets from Animal.

So we're done with the design for now; we'll come back to it later in the chapter.

Animal
picture food hunger boundaries location
makeNoise() eat() sleep() roam()



## Which method is called?

The Wolf class has four methods. One inherited from Animal, one inherited from Canine (which is actually an overridden version of a method in class Animal), and two overridden in the Wolf class. When you create a Wolf object and assign it to a variable, you can use the dot operator on that reference variable to invoke all four methods. But which *version* of those methods gets called?

Make a new Wolf object

```
Wolf w = new Wolf();
```

Calls the version in Wolf

```
w.makeNoise();
```

Calls the version in Canine

```
w.roam();
```

Calls the version in Wolf

```
w.eat();
```

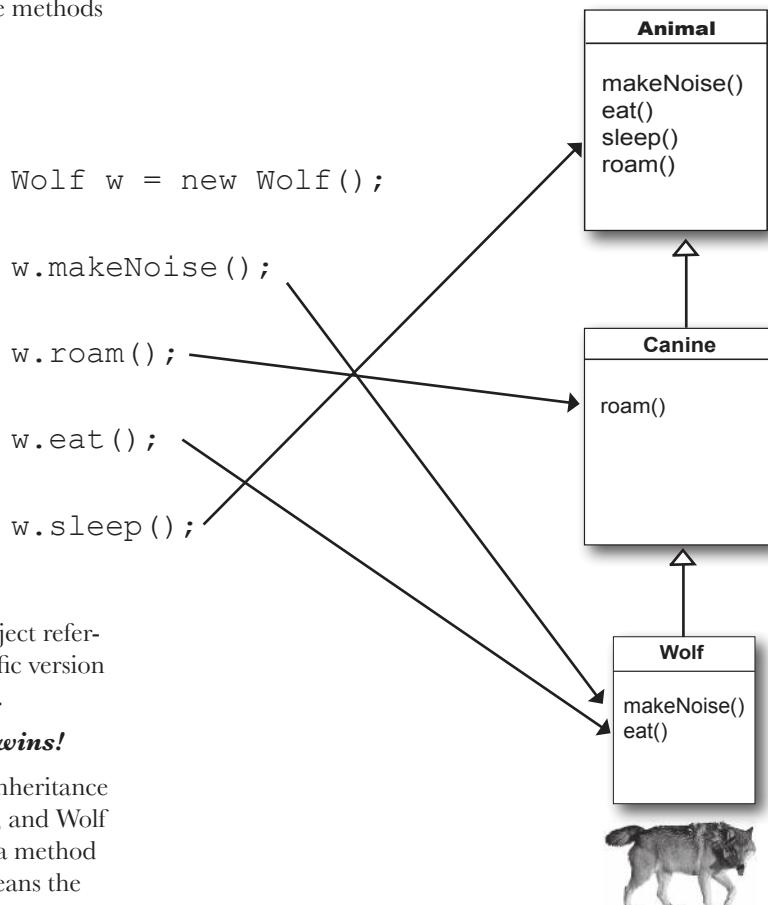
Calls the version in Animal

```
w.sleep();
```

When you call a method on an object reference, you're calling the most specific version of the method for that object type.

In other words, ***the lowest one wins!***

“Lowest” meaning lowest on the inheritance tree. Canine is lower than Animal, and Wolf is lower than Canine, so invoking a method on a reference to a Wolf object means the JVM starts looking first in the Wolf class. If the JVM doesn't find a version of the method in the Wolf class, it starts walking back up the inheritance hierarchy until it finds a match.



practice designing an inheritance tree

## Designing an Inheritance Tree

Class	Superclasses	Subclasses
Clothing	---	Boxers, Shirt
Boxers	Clothing	
Shirt	Clothing	

Inheritance Table



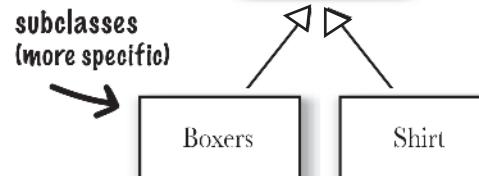
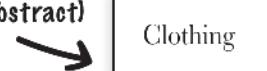
### Sharpen your pencil

Find the relationships that make sense. Fill in the last two columns

Class	Superclasses	Subclasses
Musician		
Rock Star		
Fan		
Bass Player		
Concert Pianist		

Hint: not everything can be connected to something else.

Hint: you're allowed to add to or change the classes listed.



Inheritance Class Diagram

Draw an inheritance diagram here.

→ Yours to solve.

there are no  
Dumb Questions

**Q:** You said that the JVM starts walking up the inheritance tree, starting at the class type you invoked the method on (like the Wolf example on the previous page). But what happens if the JVM doesn't ever find a match?

**A:** Good question! But you don't have to worry about that. The compiler guarantees that a particular method is callable for a specific reference type, but it doesn't say (or care) from which class that method actually comes from at runtime. With the Wolf example, the compiler checks for a sleep() method but doesn't care that sleep() is actually defined in (and inherited from) class Animal. Remember that if a class inherits a method, it has the method.

Where the inherited method is defined (in other words, in which superclass it is defined) makes no difference to the compiler. But at runtime, **the JVM will always pick the right one**. And the right one means **the most specific version for that particular object**.

# Using IS-A and HAS-A

Remember that when one class inherits from another, we say that the subclass *extends* the superclass. When you want to know if one thing should extend another, apply the IS-A test.

Triangle IS-A Shape, yeah, that works.

Cat IS-A Feline, that works too.

Surgeon IS-A Doctor, still good.

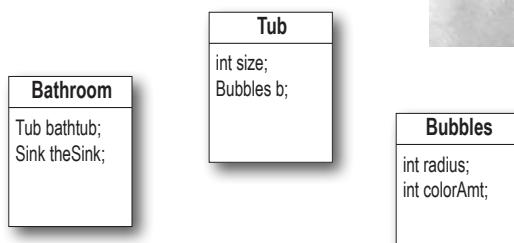
Tub extends Bathroom, sounds reasonable.

*Until you apply the IS-A test.*

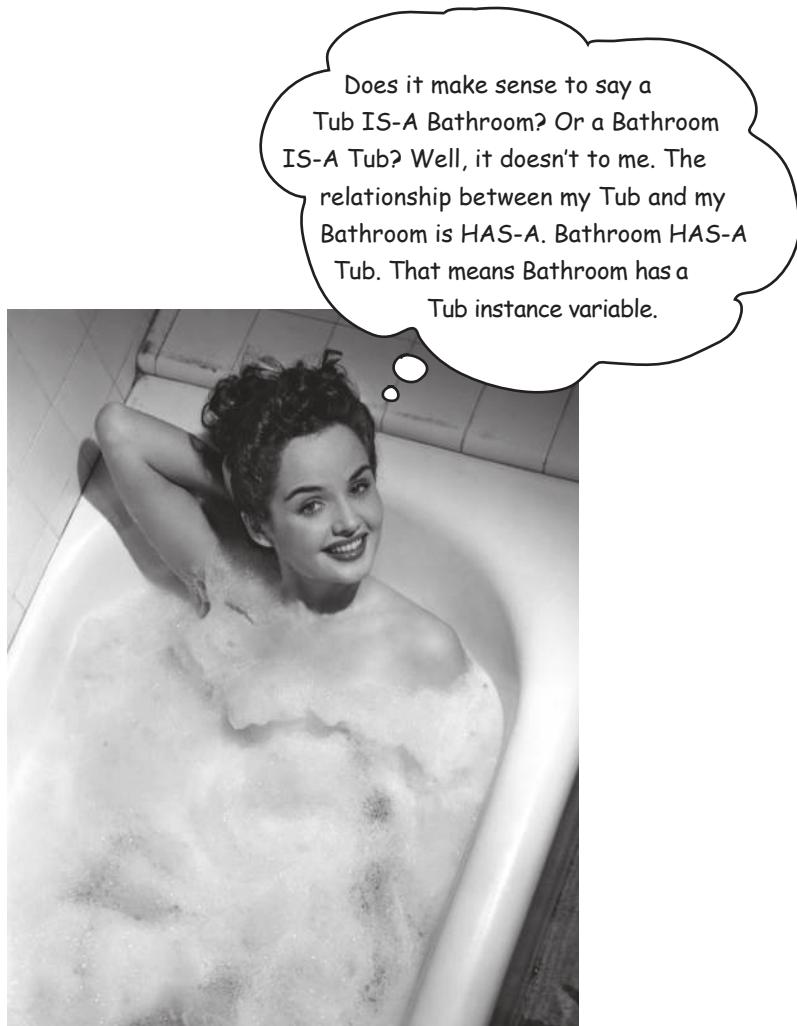
To know if you've designed your types correctly, ask, "Does it make sense to say type X IS-A type Y?" If it doesn't, you know there's something wrong with the design, so if we apply the IS-A test, Tub IS-A Bathroom is definitely false.

What if we reverse it to Bathroom extends Tub? That still doesn't work, Bathroom IS-A Tub doesn't work.

Tub and Bathroom *are* related, but not through inheritance. Tub and Bathroom are joined by a HAS-A relationship. Does it make sense to say "Bathroom HAS-A Tub"? If yes, then it means that Bathroom has a Tub instance variable. In other words, Bathroom has a *reference* to a Tub, but Bathroom does not *extend* Tub and vice versa.



Bathroom HAS-A Tub and Tub HAS-A Bubbles.  
But nobody inherits from (extends) anybody else.



## But wait! There's more!

The IS-A test works *anywhere* in the inheritance tree. If your inheritance tree is well-designed, the IS-A test should make sense when you ask *any* subclass if it IS-A *any* of its supertypes.

**If class B extends class A, class B IS-A class A.**

**This is true anywhere in the inheritance tree. If class C extends class B, class C passes the IS-A test for both B and A.**

Canine extends Animal

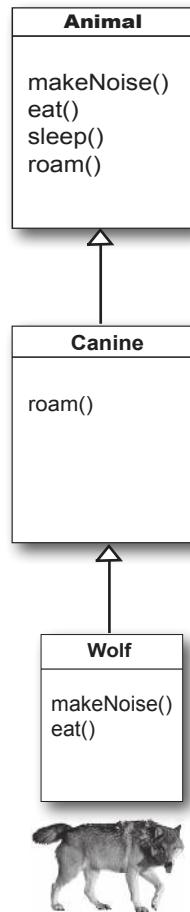
Wolf extends Canine

Wolf extends Animal

Canine IS-A Animal

Wolf IS-A Canine

Wolf IS-A Animal



With an inheritance tree like the one shown here, you're *always* allowed to say "**Wolf extends Animal**" or "**Wolf IS-A Animal.**" It makes no difference if Animal is the superclass of the superclass of Wolf. In fact, **as long as Animal is somewhere in the inheritance hierarchy above Wolf, Wolf IS-A Animal will always be true.**

The structure of the Animal inheritance tree says to the world:

"Wolf IS-A Canine, so Wolf can do anything a Canine can do. And Wolf IS-A Animal, so Wolf can do anything an Animal can do."

It makes no difference if Wolf overrides some of the methods in Animal or Canine. As far as the world (of other code) is concerned, a Wolf can do those four methods. *How* he does them, or *in which class they're overridden*, makes no difference. A Wolf can `makeNoise()`, `eat()`, `sleep()`, and `roam()` because a Wolf extends from class Animal.

## How do you know if you've got your inheritance right?

There's obviously more to it than what we've covered so far, but we'll look at a lot more OO issues in the next chapter (where we eventually refine and improve on some of the design work we did in *this* chapter).

For now, though, a good guideline is to use the IS-A test. If "X IS-A Y" makes sense, both classes (X and Y) should probably live in the same inheritance hierarchy. Chances are, they have the same or overlapping behaviors.

**Keep in mind that the inheritance IS-A relationship works in only one direction!**

Triangle IS-A Shape makes sense, so you can have Triangle extend Shape.

But the reverse—Shape IS-A Triangle—does *not* make sense, so Shape should not extend Triangle. Remember that the IS-A relationship implies that if X IS-A Y, then X can do anything a Y can do (and possibly more).



### Sharpen your pencil



Put a check next to the relationships that make sense.

- Oven extends Kitchen**
- Guitar extends Instrument**
- Person extends Employee**
- Ferrari extends Engine**
- FriedEgg extends Food**
- Beagle extends Pet**
- Container extends Jar**
- Metal extends Titanium**
- GratefulDead extends Band**
- Blonde extends Smart**
- Beverage extends Martini**

→ Yours to solve.

*Hint: apply the IS-A test*

## there are no Dumb Questions

**Q:** So we see how a subclass gets to inherit a superclass method, but what if the superclass wants to use the subclass version of the method?

**A:** A superclass won't necessarily know about any of its subclasses. You might write a class and much later someone else comes along and extends it. But even if the superclass creator does know about (and wants to use) a subclass version of a method, there's no sort of *reverse* or *backward* inheritance. Think about it, children inherit from parents, not the other way around.

**Q:** In a subclass, what if I want to use BOTH the superclass version and my overriding subclass version of a method? In other words, I don't want to completely replace the superclass version; I just want to add more stuff to it.

**A:** You can do this! And it's an important design feature. Think of the word "extends" as meaning "I want to extend the functionality of the superclass."

```
public void roam() {  
    super.roam();  
    // my own roam stuff  
}
```

You can design your superclass methods in such a way that they contain method implementations that will work for any subclass, even though the subclasses may still need to "append" more code. In your subclass overriding method, you can call the superclass version using the keyword **super**. It's like saying, "first go run the superclass version, then come back and finish with my own code..."

This calls the inherited version of `roam()`, then comes back to do your own subclass-specific code

## Who gets the Porsche, who gets the porcelain? (how to know what a subclass can inherit from its superclass)



A subclass inherits members of the superclass. Members include instance variables and methods, although later in this book we'll look at other inherited members. A superclass can choose whether or not it wants a subclass to inherit a particular member by the level of access the particular member is given.

There are four access levels that we'll cover in this book. Moving from most restrictive to least, the four access levels are:

**private    default    protected    public**



Access levels control *who sees what*, and are crucial to having well-designed, robust Java code. For now we'll focus just on public and private. The rules are simple for those two:

**public members are inherited**  
**private members are not inherited**

When a subclass inherits a member, it is *as if the subclass defined the member itself*. In the Shape example, Square inherited the `rotate()` and `playSound()` methods and to the outside world (other code) the Square class simply *has* a `rotate()` and `playSound()` method.

The members of a class include the variables and methods defined in the class plus anything inherited from a superclass.

Note: get more details about default and protected in Appendix B.

# When designing with inheritance, are you using or abusing?

Although some of the reasons behind these rules won't be revealed until later in this book, for now, simply *knowing* a few rules will help you build a better inheritance design.

**DO** use inheritance when one class is a more specific type of a superclass. Example: Willow *is a* more specific type of Tree, so Willow *extends* Tree makes sense.

**DO** consider inheritance when you have behavior (implemented code) that should be shared among multiple classes of the same general type. Example: Square, Circle, and Triangle all need to rotate and play sound, so putting that functionality in a superclass Shape might make sense and makes for easier maintenance and extensibility. Be aware, however, that while inheritance is one of the key features of object-oriented programming, it's not necessarily the best way to achieve behavior reuse. It'll get you started, and often it's the right design choice, but design patterns will help you see other more subtle and flexible options. If you don't know about design patterns, a good follow-on to this book would be *Head First Design Patterns*.

**DO NOT** use inheritance just so that you can reuse code from another class, if the relationship between the superclass and subclass violate either of the above two rules. For example, imagine you wrote special printing code in the Animal class and now you need printing code in the Potato class. You might think about making Potato extend Animal so that Potato inherits the printing code. That makes no sense! A Potato is *not* an Animal! (So the printing code should be in a Printer class that all printable objects can take advantage of via a HAS-A relationship.)

**DO NOT** use inheritance if the subclass and superclass do not pass the IS-A test. Always ask yourself if the subclass IS-A more specific type of the superclass. Example: Tea IS-A Beverage makes sense. Beverage IS-A Tea does not.

## BULLET POINTS

- A subclass *extends* a superclass.
- A subclass inherits all *public* instance variables and methods of the superclass, but does not inherit the *private* instance variables and methods of the superclass.
- Inherited methods *can* be overridden; instance variables *cannot* be overridden (although they can be *redefined* in the subclass, but that's not the same thing, and there's almost never a need to do it.)
- Use the IS-A test to verify that your inheritance hierarchy is valid. If X *extends* Y, then X IS-A Y must make sense.
- The IS-A relationship works in only one direction. A Hippo is an Animal, but not all Animals are Hippos.
- When a method is overridden in a subclass, and that method is invoked on an instance of the subclass, the overridden version of the method is called. (*The lowest one wins.*)
- If class B extends A, and C extends B, class B IS-A class A, and class C IS-A class B, and class C also IS-A class A.

## So what does all this inheritance really buy you?

You get a lot of OO mileage by designing with inheritance. You can get rid of duplicate code by abstracting out the behavior common to a group of classes, and sticking that code in a superclass. That way, when you need to modify it, you have only one place to update, and *the change is magically reflected in all the classes that inherit that behavior.*

Well, there's no magic involved, but it *is* pretty simple: make the change and compile the class again. That's it. **You don't have to touch the subclasses!**

**Just deliver the newly changed superclass, and all classes that extend it will automatically use the new version.**

A Java program is nothing but a pile of classes, so the subclasses don't have to be recompiled in order to use the new version of the superclass. As long as the superclass doesn't *break* anything for the subclass, everything's fine. (We'll discuss what the word "break" means in this context later in the book. For now, think of it as modifying something in the superclass that the subclass is depending on, like a particular method's arguments, return type, method name, etc.)

1

### **You avoid duplicate code.**

Put common code in one place, and let the subclasses inherit that code from a superclass. When you want to change that behavior, you have to modify it in only one place, and everybody else (i.e., all the subclasses) sees the change.

2

### **You define a common protocol for a group of classes.**



# Inheritance lets you guarantee that all classes grouped under a certain supertype have all the methods that the supertype has\*

In other words, you define a common protocol for a set of classes related through inheritance.

When you define methods in a superclass that can be inherited by subclasses, you're announcing a kind of protocol to other code that says, "All my subtypes (i.e., subclasses) can do these things, with these methods that look like this..."

In other words, you establish a *contract*.

Class Animal establishes a common protocol for all Animal subtypes:

Animal
makeNoise() eat() sleep() roam()

You're telling the world that any Animal can do these four things. That includes the method arguments and return types.

And remember, when we say *any Animal*, we mean Animal and *any class that extends from Animal*. That again means, *any class that has Animal somewhere above it in the inheritance hierarchy*.

But we're not even at the really cool part yet, because we saved the best—*polymorphism*—for last.

When you define a supertype for a group of classes, *any subclass of that supertype can be substituted where the supertype is expected*.

Say, what?

Don't worry, we're nowhere near done explaining it. Two pages from now, you'll be an expert.

\*When we say "all the methods," we mean "all the *inheritable* methods," which for now actually means "all the *public* methods," although later we'll refine that definition a bit more.

## And I care because...

You get to take advantage of polymorphism.

## Which matters to me because...

You get to refer to a subclass object using a reference declared as the supertype.

## And that means to me...

You get to write really flexible code. Code that's cleaner (more efficient, simpler). Code that's not just easier to *develop*, but also much, much easier to *extend*, in ways you never imagined at the time you originally wrote your code.

That means you can take that tropical vacation while your co-workers update the program, and your co-workers might not even need your source code.

You'll see how it works on the next page.

We don't know about you, but personally, we find the whole tropical vacation thing particularly motivating.



**To see how polymorphism works, we have to step back and look at the way we normally declare a reference and create an object...**

## The 3 steps of object declaration and assignment

**1** **Dog myDog**    **3** **= new Dog();**    **2**

**1** Declare a reference variable

`Dog myDog = new Dog();`

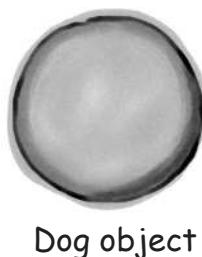
Tells the JVM to allocate space for a reference variable. The reference variable is, forever, of type Dog. In other words, a remote control that has buttons to control a Dog, but not a Cat or a Button or a Socket.



**2** Create an object

`Dog myDog = new Dog();`

Tells the JVM to allocate space for a new Dog object on the garbage collectible heap.

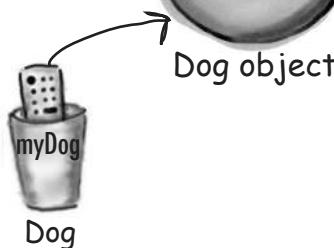


Dog object

**3** Link the object and the reference

`Dog myDog = new Dog();`

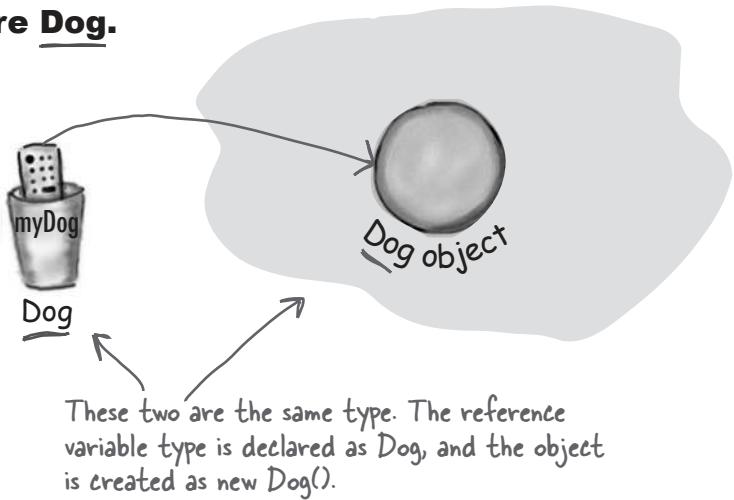
Assigns the new Dog to the reference variable myDog. In other words, **program the remote control**.



Dog object

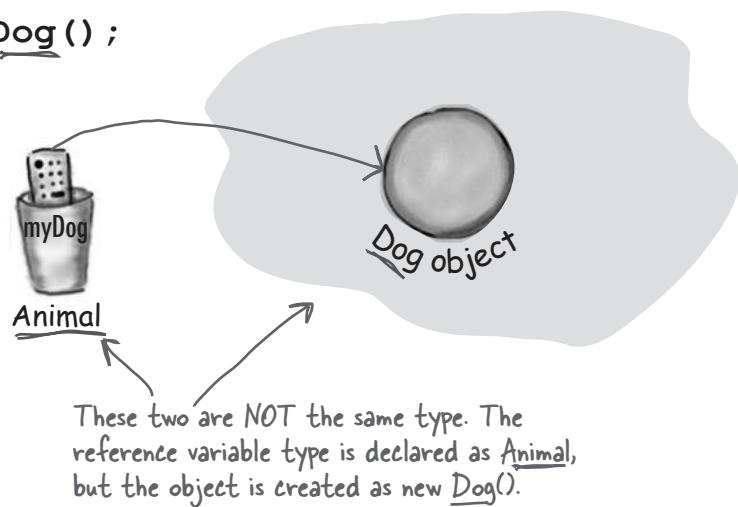
**The important point is that the reference type AND the object type are the same.**

**In this example, both are Dog.**



**But with polymorphism, the reference type and the object type can be different.**

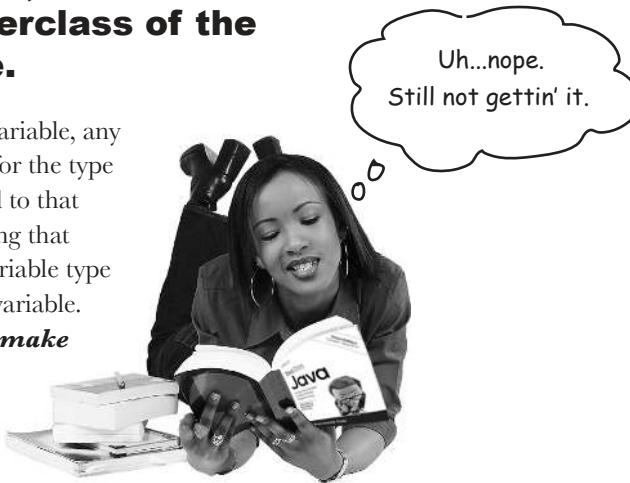
Animal myDog = new Dog();



## With polymorphism, the reference type can be a superclass of the actual object type.

When you declare a reference variable, any object that passes the IS-A test for the type of the reference can be assigned to that variable. In other words, anything that *extends* the declared reference variable type can be *assigned* to the reference variable.

**This lets you do things like make polymorphic arrays.**



## OK, OK maybe an example will help.

```
Animal[] animals = new Animal[5];  
  
animals[0] = new Dog();  
animals[1] = new Cat();  
animals[2] = new Wolf();  
animals[3] = new Hippo();  
animals[4] = new Lion();  
  
for (Animal animal : animals) {  
    animal.eat();  
    animal.roam();  
}
```

Declare an array of type Animal. In other words, an array that will hold objects of type Animal.

But look what you get to do...you can put ANY subclass of Animal in the Animal array!

And here's the best polymorphic part (the raison d'être for the whole example): you get to loop through the array and call one of the Animal-class methods, and every object does the right thing!

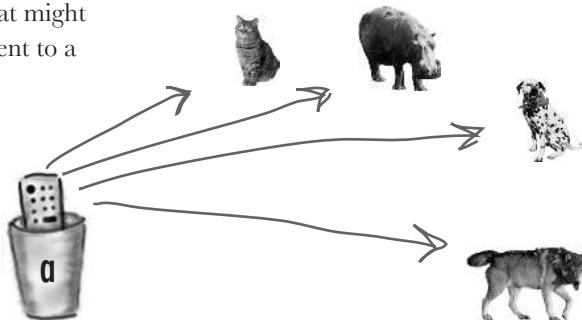
On the first pass through the loop, 'animal' is a Dog, so you get the Dog's eat() method. On the next pass, 'animal' is a Cat, so you get the Cat's eat() method.

Same with roam().

## But wait! There's more!

### You can have polymorphic arguments and return types.

If you can declare a reference variable of a supertype, say, Animal, and assign a subclass object to it, say, Dog, think of how that might work when the reference is an argument to a method...

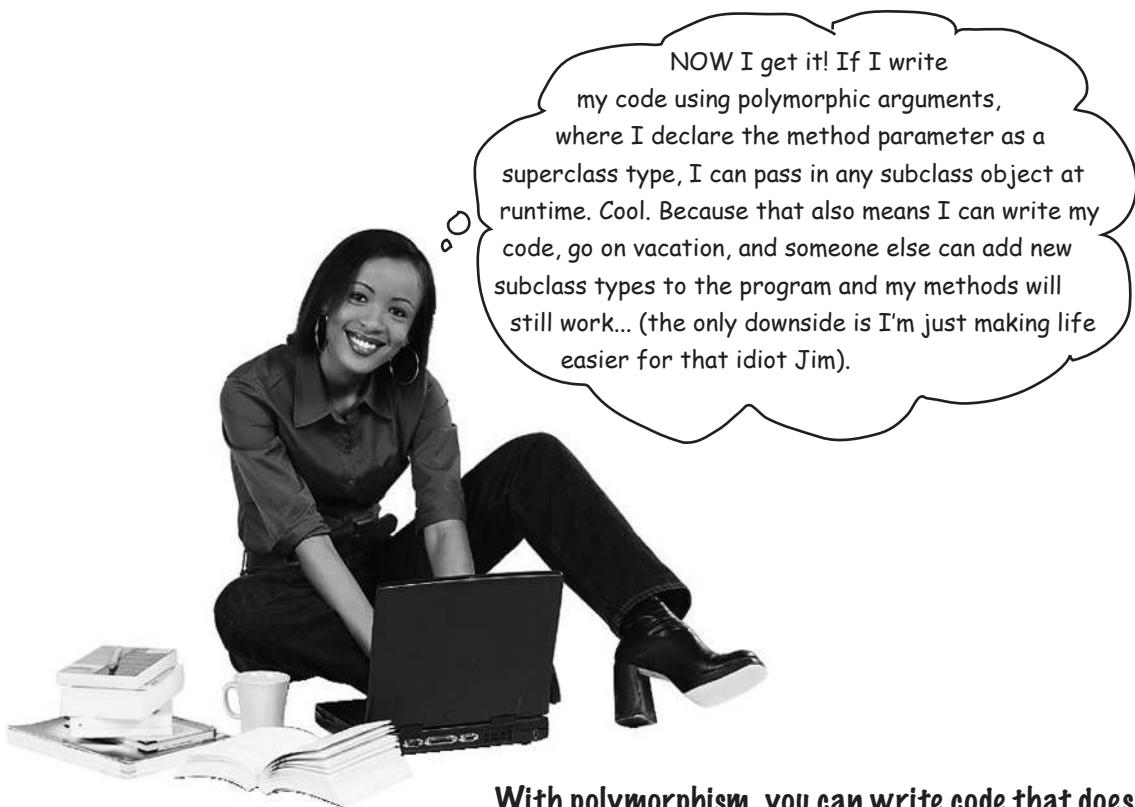


```
class Vet {
    public void giveShot(Animal a) {
        // do horrible things to the Animal at
        // the other end of the 'a' parameter
        a.makeNoise();
    }
}
```

The 'a' parameter can take ANY Animal type as the argument. And when the Vet is done giving the shot, it tells the Animal to makeNoise(), and whatever Animal is really out there on the heap, that's whose makeNoise() method will run.

```
class PetOwner {
    public void start() {
        Vet vet = new Vet();
        Dog dog = new Dog();
        Hippo hippo = new Hippo();
        vet.giveShot(dog);           ← Dog's makeNoise() runs
        vet.giveShot(hippo);         ← Hippo's makeNoise() runs
    }
}
```

The Vet's giveShot() method can take any Animal you give it. As long as the object you pass in as the argument is a subclass of Animal, it will work.



**With polymorphism, you can write code that doesn't have to change when you introduce new subclass types into the program.**

Remember that Vet class? If you write that Vet class using arguments declared as type *Animal*, your code can handle any *Animal subclass*. That means if others want to take advantage of your Vet class, all they have to do is make sure *their* new Animal types extend class Animal. The Vet methods will still work, even though the Vet class was written without any knowledge of the new Animal subtypes the Vet will be working on.



Why is polymorphism guaranteed to work this way? Why is it always safe to assume that any *subclass* type will have the methods you think you're calling on the *superclass* type (the superclass reference type you're using the dot operator on)?

there are no  
Dumb Questions

**Q:** Are there any practical limits on the levels of subclassing? How deep can you go?

**A:** If you look in the Java API, you'll see that most inheritance hierarchies are wide but not deep. Most are no more than one or two levels deep, although there are exceptions (especially in the GUI classes). You'll come to realize that it usually makes more sense to keep your inheritance trees shallow, but there isn't a hard limit (well, not one that you'd ever run into).

**Q:** Hey, I just thought of something...if you don't have access to the source code for a class but you want to change the way a method of that class works, could you use subclassing to do that? To extend the "bad" class and override the method with your own better code?

**A:** Yep. That's one cool feature of OO, and sometimes it saves you from having to rewrite the class from scratch or track down the programmer who hid the source code.

**Q:** Can you extend *any* class? Or is it like class members where if the class is private you can't inherit it...

**A:** There's no such thing as a private class, except in a very special case called an *inner* class, which we haven't looked at yet. But there are three things that can prevent a class from being subclassed.

The first is access control. Even though a class *can't* be marked `private`, a class *can* be non-public (what you get if you don't declare the class as `public`). A non-public class can be subclassed only by classes in the same package as the class. Classes in a different package won't be able to subclass (or even *use*, for that matter) the non-public class.

The second thing that stops a class from being subclassed is the keyword modifier `final`. A final class means that it's the end of the inheritance line. Nobody, ever, can extend a final class.

The third issue is that if a class has only *private* constructors (we'll look at constructors in Chapter 9), it can't be subclassed.

**Q:** Why would you ever want to make a final class? What advantage would there be in preventing a class from being subclassed?

**A:** Typically, you won't make your classes final. But if you need security—the security of knowing that the methods will always work the way that you wrote them (because they can't be overridden), a final class will give you that. A lot of classes in the Java API are final for that reason. The `String` class, for example, is final because, well, imagine the havoc if somebody came along and changed the way `Strings` behave!

**Q:** Can you make a *method* final, without making the whole class final?

**A:** If you want to protect a specific method from being overridden, mark the *method* with the `final` modifier. Mark the whole *class* as final if you want to guarantee that *none* of the methods in that class will ever be overridden.

# Keeping the contract: rules for overriding

When you override a method from a superclass, you're agreeing to fulfill the contract. The contract that says, for example, "I take no arguments and I return a boolean." In other words, the arguments and return types of your overriding method must look to the outside world *exactly* like the overridden method in the superclass.

## The methods are the contract.

If polymorphism is going to work, the Toaster's version of the overridden method from Appliance has to work at runtime. Remember, the compiler looks at the reference type to decide whether you can call a particular method on that reference.

```
Appliance appliance = new Toaster();
```

*Reference type*                           *Object type*

With an *Appliance* reference to a Toaster, the compiler cares only if class *Appliance* has the method you're invoking on an *Appliance* reference. But at runtime, the JVM does not look at the **reference** type (*Appliance*) but at the actual **Toaster object** on the heap.

So if the compiler has already approved the method call, the only way it can work is if the overriding method has the same arguments and return types. Otherwise, someone with an *Appliance* reference will call *turnOn()* as a no-arg method, even though there's a version in *Toaster* that takes an int. Which one is called at runtime? The one in *Appliance*. In other words, **the *turnOn(int level)* method in *Toaster* is not an override!**

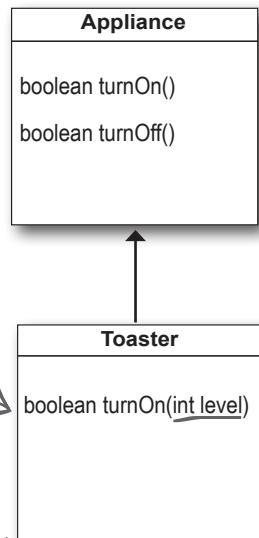
## ① Arguments must be the same, and return types must be compatible.

The contract of superclass defines how other code can use a method. Whatever the superclass takes as an argument, the subclass overriding the method must use that same argument. And whatever the superclass declares as a return type, the overriding method must declare either the same type or a subclass type. Remember, a subclass object is guaranteed to be able to do anything its superclass declares, so it's safe to return a subclass where the superclass is expected.

## ② The method can't be less accessible.

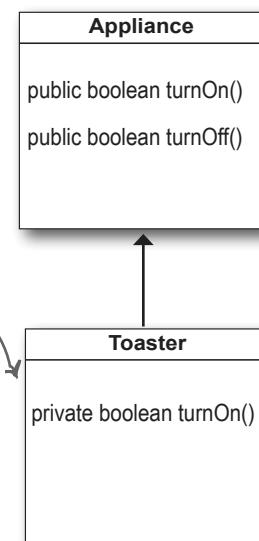
That means the access level must be the same, or friendlier. You can't, for example, override a public method and make it private. What a shock that would be to the code invoking what it *thinks* (at compile time) is a public method, if suddenly at runtime the JVM slammed the door shut because the overriding version called at runtime is private!

So far we've learned about two access levels: private and public. The other two are in appendix B. There's also another rule about overriding related to exception handling, but we'll wait until Chapter 13, *Risky Behavior*, to cover that.



This is NOT an override!  
Can't change the arguments in an overriding method!

This is actually a legal overLOAD, but not an overRIDE.



NOT LEGAL!  
It's not a legal override because you restricted the access level. Nor is it a legal overLOAD, because you didn't change arguments.

# Overloading a method

Method overloading is nothing more than having two methods with the same name but different argument lists. Period. There's no polymorphism involved with overloaded methods!

Overloading lets you make multiple versions of a method, with different argument lists, for convenience to the callers. For example, if you have a method that takes only an int, the calling code has to convert, say, a double into an int before calling your method. But if you overloaded the method with another version that takes a double, then you've made things easier for the caller. You'll see more of this when we look into constructors in Chapter 9, *Life and Death of an Object*.

Since an overloading method isn't trying to fulfill the polymorphism contract defined by its superclass, overloaded methods have much more flexibility.

An overloaded method is just a different method that happens to have the same method name. It has nothing to do with inheritance and polymorphism. An overloaded method is NOT the same as an overridden method.

## ① The return types can be different.

You're free to change the return types in overloaded methods, as long as the argument lists are different.

## ② You can't change ONLY the return type.

If only the return type is different, it's not a valid *overload*—the compiler will assume you're trying to *override* the method. And even *that* won't be legal unless the return type is a subtype of the return type declared in the superclass. To overload a method, you MUST change the argument list, although you *can* change the return type to anything.

## ③ You can vary the access levels in any direction.

You're free to overload a method with a method that's more restrictive. It doesn't matter, since the new method isn't obligated to fulfill the contract of the overloaded method.

## Legal examples of method overloading:

```
public class Overloads {
    String uniqueID;

    public int addNums(int a, int b) {
        return a + b;
    }

    public double addNums(double a, double b) {
        return a + b;
    }

    public void setUniqueID(String theID) {
        // lots of validation code, and then:
        uniqueID = theID;
    }

    public void setUniqueID(int ssNumber) {
        String numString = "" + ssNumber;
        setUniqueID(numString);
    }
}
```

## exercise: Mixed Messages



A short Java program is listed below. One block of the program is missing! Your challenge is to match the candidate block of code (on the left), with the output that you'd see if the block were inserted. Not all the lines of output will be used, and some of the lines of output might be used more than once. Draw lines connecting the candidate blocks of code with their matching command-line output.

### The program:

```
class A {  
    int ivar = 7;  
  
    void m1() {  
        System.out.print("A's m1, ");  
    }  
    void m2() {  
        System.out.print("A's m2, ");  
    }  
    void m3() {  
        System.out.print("A's m3, ");  
    }  
}  
  
class B extends A {  
    void m1() {  
        System.out.print("B's m1, ");  
    }  
}
```

```
class C extends B {  
    void m3() {  
        System.out.print("C's m3, " + (ivar + 6));  
    }  
}  
  
public class Mixed2 {  
    public static void main(String[] args) {  
        A a = new A();  
        B b = new B();  
        C c = new C();  
        A a2 = new C();  
        _____  
    }  
}
```

Candidate code  
goes here  
(three lines)

### Code candidates:

b.m1();  
c.m2();  
a.m3();

c.m1();  
c.m2();  
c.m3();

a.m1();  
b.m2();  
c.m3();

a2.m1();  
a2.m2();  
a2.m3();

### Output:

A's m1, A's m2, C's m3, 6

B's m1, A's m2, A's m3,

A's m1, B's m2, A's m3,

B's m1, A's m2, C's m3, 13

B's m1, C's m2, A's m3,

B's m1, A's m2, C's m3, 6

A's m1, A's m2, C's m3, 13



# BE the Compiler

Which of the A-B pairs of methods listed on the right, if inserted into the classes on the left, would compile and produce the output shown? (The A method inserted into class Monster, the B method inserted into class Vampire.)

```
public class MonsterTestDrive {
    public static void main(String[] args) {
        Monster[] monsters = new Monster[3];
        monsters[0] = new Vampire();
        monsters[1] = new Dragon();
        monsters[2] = new Monster();
        for (int i = 0; i < monsters.length; i++) {
            monsters[i].frighten(i);
        }
    }
}

class Monster {
    A
}

class Vampire extends Monster {
    B
}

class Dragon extends Monster {
    boolean frighten(int degree) {
        System.out.println("breathe fire");
        return true;
    }
}
```

File Edit Window Help SaveYourself

```
% java MonsterTestDrive
a bite?
breathe fire
arrrgh
```

- 1     boolean frighten(int d) {
 **A**     System.out.println("arrrgh");
 return true;
 }
 **B**     boolean frighten(int x) {
 System.out.println("a bite?");
 return false;
 }

---

- 2     boolean frighten(int x) {
 **A**     System.out.println("arrrgh");
 return true;
 }
 **B**     int frighten(int f) {
 System.out.println("a bite?");
 return 1;
 }

---

- 3     boolean frighten(int x) {
 **A**     System.out.println("arrrgh");
 return false;
 }
 boolean scare(int x) {
 **B**     System.out.println("a bite?");
 return true;
 }

---

- 4     boolean frighten(int z) {
 **A**     System.out.println("arrrgh");
 return true;
 }
 **B**     boolean frighten(byte b) {
 System.out.println("a bite?");
 return true;
 }

→ Answers on page 197.

## puzzle: Pool Puzzle



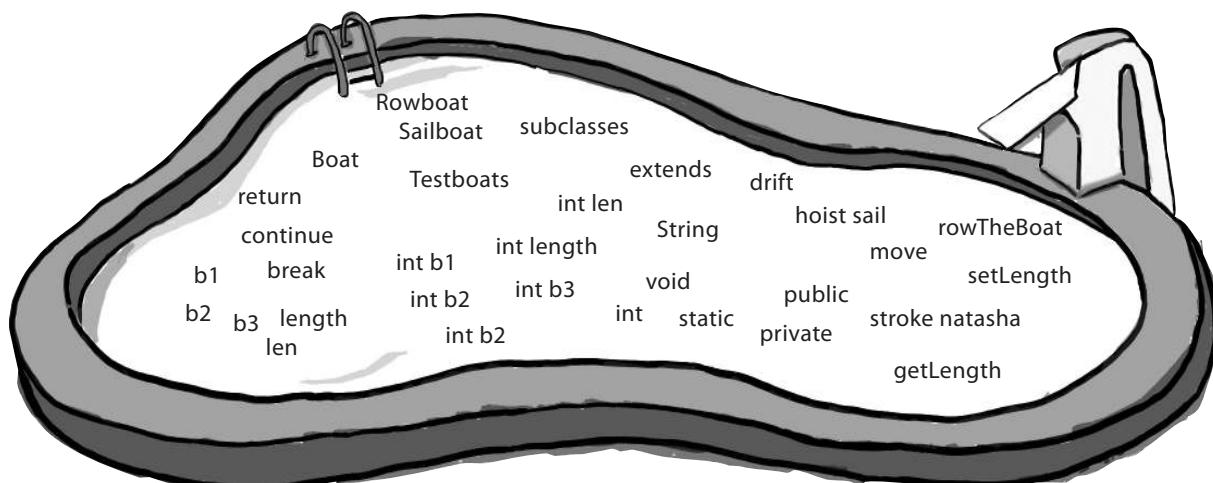
## Pool Puzzle

Your **job** is to take code snippets from the pool and place them into the blank lines in the code. You may use the same snippet more than once, and you might not need to use all the snippets. Your **goal** is to make a set of classes that will compile and run together as a program. Don't be fooled—this one's harder than it looks.

```
public class Rowboat _____ {  
    public _____ rowTheBoat() {  
        System.out.print("stroke natasha");  
    }  
  
    public class _____ {  
        private int _____;  
        void _____(_____) {  
            length = len;  
        }  
        public int getLength() {  
            _____;  
        }  
        public _____ move() {  
            System.out.print("_____");  
        }  
    }  
}
```

```
public class TestBoats {  
    _____ main(String[] args){  
        _____ b1 = new Boat();  
        Sailboat b2 = new _____();  
        Rowboat _____ = new Rowboat();  
        b2.setLength(32);  
        b1._____();  
        b3._____();  
        _____.move();  
    }  
  
    public class _____ Boat {  
        public _____ _____() {  
            System.out.print("_____");  
        }  
    }  
}
```

**OUTPUT:** drift drift hoist sail





## Exercise Solutions



## BE the Compiler (from page 195)

Set 1 **will** work.

Set 2 **will not** compile because of Vampire's return type (int).

The Vampire's `frighten()` method (B) is not a legal override OR overload of Monster's `frighten()` method. Changing ONLY the return type is not enough to make a valid overload, and since an int is not compatible with a boolean, the method is not a valid override. (Remember, if you change ONLY the return type, it must be to a return type that is compatible with the superclass version's return type, and then it's an *override*.)

Sets 3 and 4 **will** compile but produce:

**arrrgh**

**breathe fire**

**arrrgh**

Remember, class Vampire did not override class Monster's `frighten()` method. (The `frighten()` method in Vampire's set 4 takes a byte, not an int.)

## Code candidates:



```
b.m1();
c.m2();
a.m3();
```

```
c.m1();
c.m2();
c.m3();
```

```
a.m1();
b.m2();
c.m3();
```

```
a2.m1();
a2.m2();
a2.m3();
```

## Output:

A's m1, A's m2, C's m3, 6

B's m1, A's m2, A's m3,

A's m1, B's m2, A's m3,

B's m1, A's m2, C's m3, 13

B's m1, C's m2, A's m3,

B's m1, A's m2, C's m3, 6

A's m1, A's m2, C's m3, 13



## Poo] Puzz]e (from page 196)

```
public class Rowboat extends Boat {
    public void rowTheBoat() {
        System.out.print("stroke natasha");
    }
}

public class Boat {
    private int length ;
    public void setLength ( int len ) {
        length = len;
    }
    public int getLength() {
        return length ;
    }
    public void move() {
        System.out.print("drift ");
    }
}
```

```
public class TestBoats {
    public static void main(String[] args){
        Boat b1 = new Boat();
        Sailboat b2 = new Sailboat();
        Rowboat b3 = new Rowboat();
        b2.setLength(32);
        b1.move();
        b3.move();
        b2.move();
    }
}

public class Sailboat extends Boat {
    public void move() {
        System.out.print("hoist sail ");
    }
}
```

**OUTPUT:** drift drift hoist sail

# Serious Polymorphism

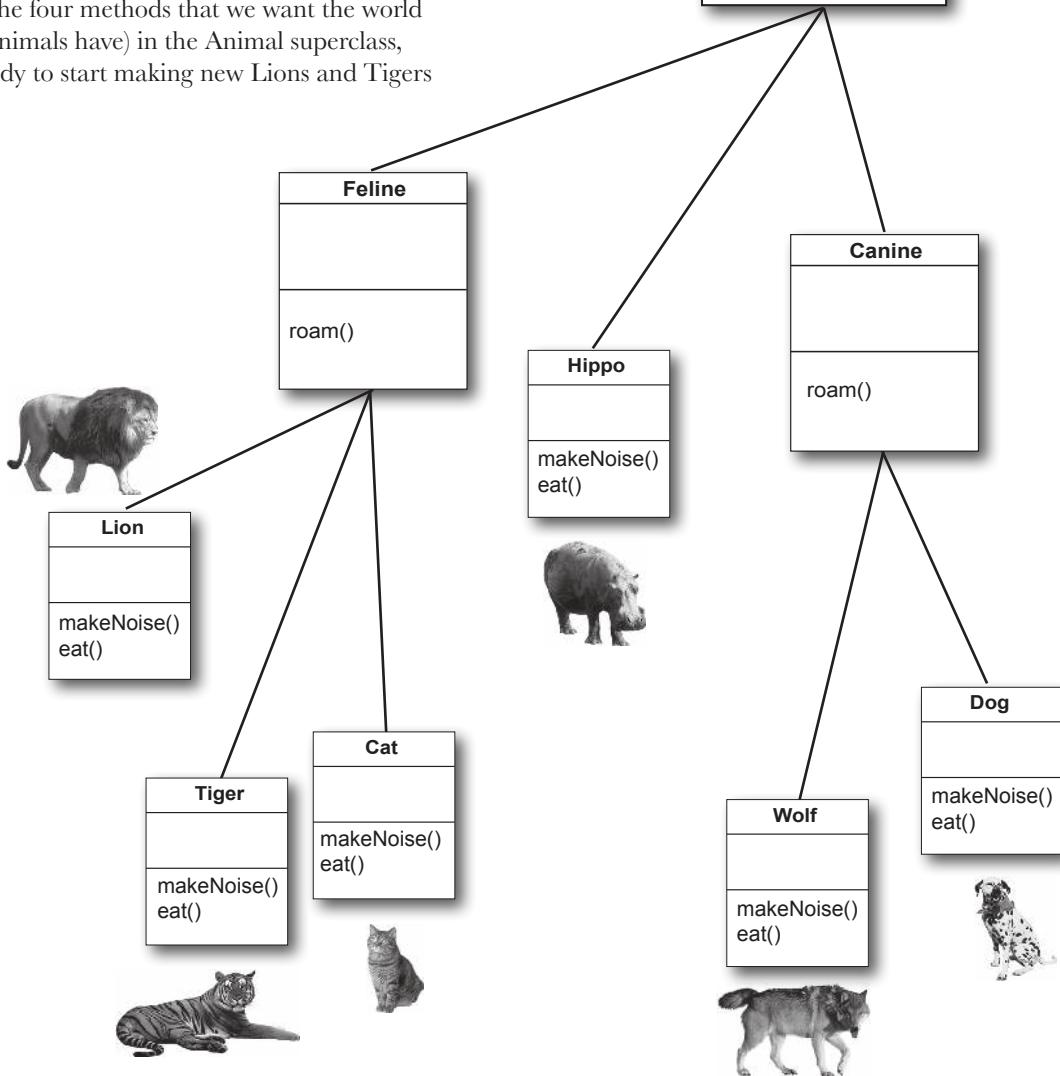


**Inheritance is just the beginning.** To exploit polymorphism, we need interfaces (and not the GUI kind). We need to go beyond simple inheritance to a level of flexibility and extensibility you can get only by designing and coding to interface specifications. Some of the coolest parts of Java wouldn't even be possible without interfaces, so even if you don't design with them yourself, you still have to use them. But you'll *want* to design with them. You'll *need* to design with them. **You'll wonder how you ever lived without them.** What's an interface? It's a 100% abstract class. What's an abstract class? It's a class that can't be instantiated. What's that good for? You'll see in just a few moments. But if you think about the end of the previous chapter, and how we used polymorphic arguments so that a single Vet method could take Animal subclasses of all types, well, that was just scratching the surface. Interfaces are the **poly** in polymorphism. The **ab** in abstract. The **caffeine** in Java.

# Did we forget about something when we designed this?

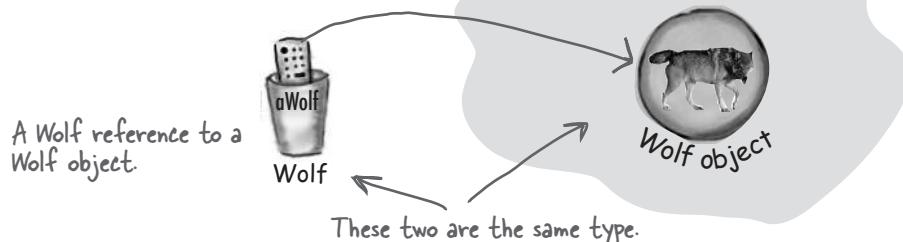
The class structure isn't too bad. We've designed it so that duplicate code is kept to a minimum, and we've overridden the methods that we think should have subclass-specific implementations. We've made it nice and flexible from a polymorphic perspective, because we can design Animal-using programs with Animal arguments (and array declarations) so that any Animal subtype—**including those we never imagined at the time we wrote our code**—can be passed in and used at runtime. We've put the common protocol for all Animals (the four methods that we want the world to know all Animals have) in the Animal superclass, and we're ready to start making new Lions and Tigers and Hippos.

Animal
picture food hunger boundaries location
makeNoise() eat() sleep() roam()

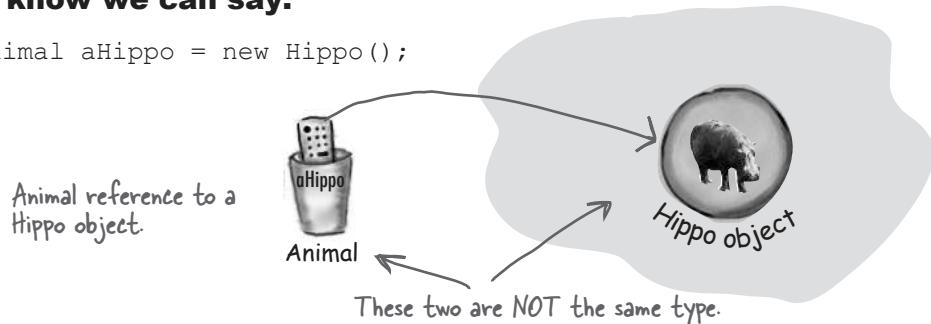


**We know we can say:**

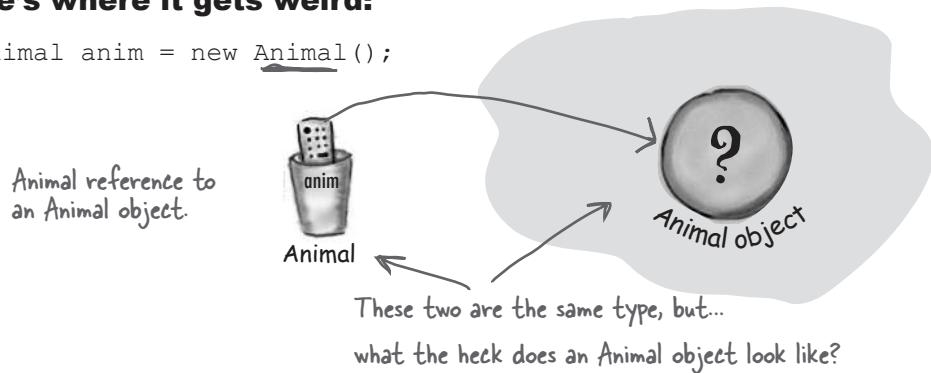
```
Wolf aWolf = new Wolf();
```

**And we know we can say:**

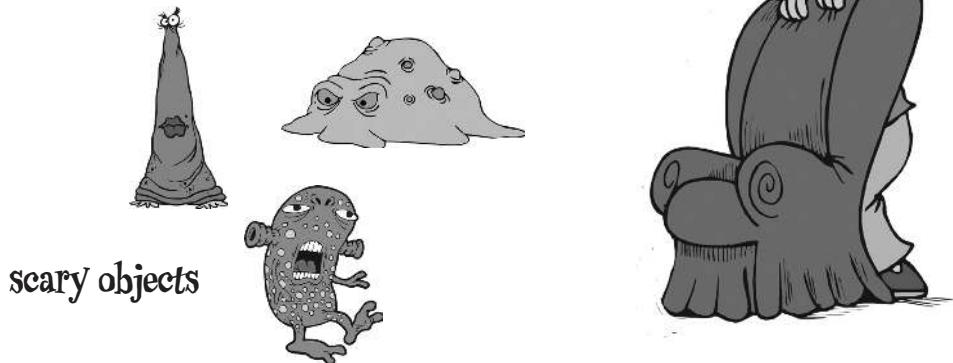
```
Animal aHippo = new Hippo();
```

**But here's where it gets weird:**

```
Animal anim = new Animal();
```



## What does a new Animal() object look like?



## What are the instance variable values?

### Some classes just should not be instantiated!

It makes sense to create a Wolf object or a Hippo object or a Tiger object, but what exactly *is* an Animal object? What shape is it? What color, size, number of legs...

Trying to create an object of type Animal is like a **nightmare Star Trek™ transporter accident**. The one where somewhere in the beam-me-up process something bad happened to the buffer.

But how do we deal with this? We *need* an Animal class, for inheritance and polymorphism. But we want programmers to instantiate only the less abstract *subclasses* of class Animal, not Animal itself. We want Tiger objects and Lion objects, **not Animal objects**.

Fortunately, there's a simple way to prevent a class from ever being instantiated. In other words, to stop anyone from saying “**new**” on that type. By marking the class as **abstract**, the compiler will stop any code, anywhere, from ever creating an instance of that type.

You can still use that abstract type as a reference type. In fact, that's a big part of why you have that abstract class

in the first place (to use it as a polymorphic argument or return type, or to make a polymorphic array).

When you're designing your class inheritance structure, you have to decide which classes are *abstract* and which are *concrete*. Concrete classes are those that are specific enough to be instantiated. A *concrete* class just means that it's OK to make objects of that type.

Making a class abstract is easy—put the keyword **abstract** before the class declaration:

```
abstract class Canine extends Animal {  
    public void roam() { }  
}
```

# The compiler won't let you instantiate an abstract class

An abstract class means that nobody can ever make a new instance of that class. You can still use that abstract class as a declared reference type, for the purpose of polymorphism, but you don't have to worry about somebody making objects of that type. The compiler *guarantees* it.

```
abstract public class Canine extends Animal
{
    public void roam() { }
}
```

```
public class MakeCanine {
    public void go() {
        Canine c;           }   ← This is OK, because you can always assign a
        c = new Dog();     subclass object to a superclass reference, even
        c = new Canine(); ← if the superclass is abstract.
        c.roam();
    }
}
```

```
File Edit Window Help BeamMeUp

% javac MakeCanine.java
MakeCanine.java:5: Canine is abstract;
cannot be instantiated
    c = new Canine();
               ^
1 error
```

An **abstract class** has virtually\* no use, no value, no purpose in life, unless it is **extended**.

With an abstract class, it's the **instances of a subclass** of your abstract class that's doing the work at runtime

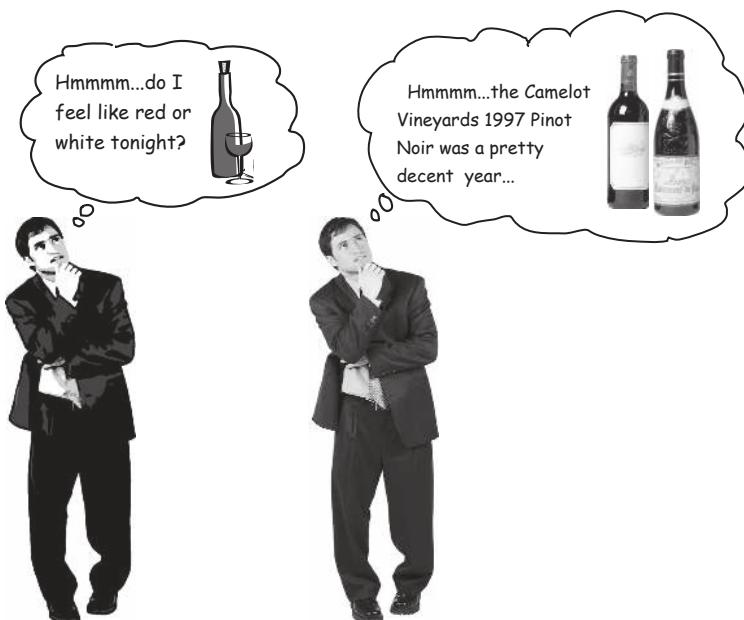
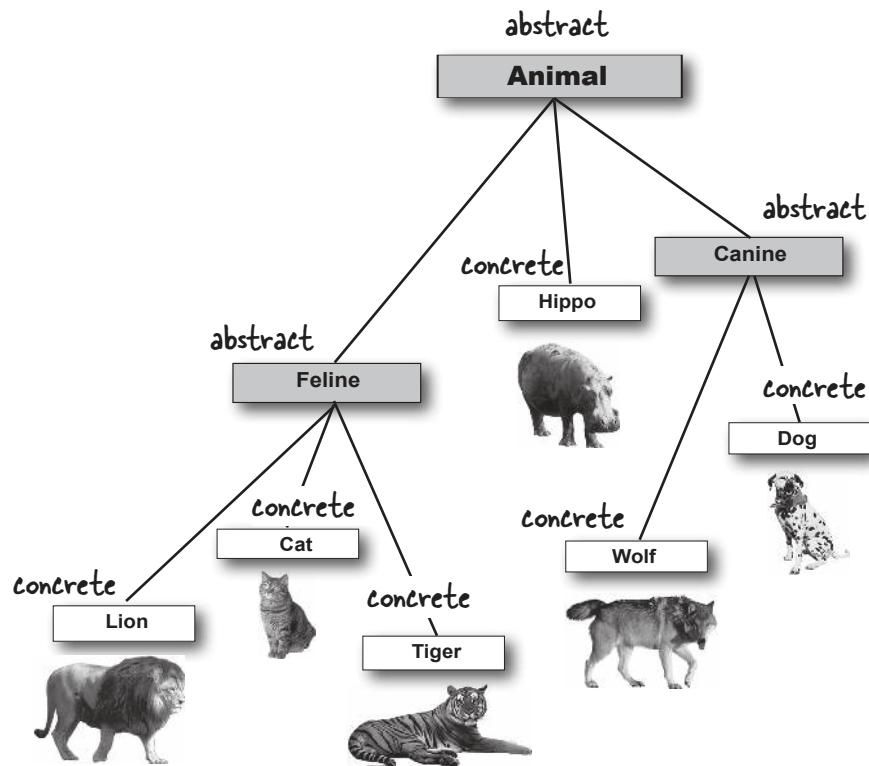
\*There is an exception to this—an abstract class can have static members (see Chapter 10).

## abstract and concrete classes

### Abstract vs. Concrete

A class that's not abstract is called a *concrete* class. In the Animal inheritance tree, if we make Animal, Canine, and Feline abstract, that leaves Hippo, Wolf, Dog, Tiger, Lion, and Cat as the concrete subclasses.

Flip through the Java API and you'll find a lot of abstract classes, especially in the GUI library. What does a GUI Component look like? The Component class is the superclass of GUI-related classes for things like buttons, text areas, scrollbars, dialog boxes, you name it. You don't make an instance of a generic *Component* and put it on the screen; you make a JButton. In other words, you instantiate only a *concrete subclass* of Component, but never Component itself.



### Abstract or concrete?

How do you know when a class should be abstract? **Wine** is probably abstract. But what about **Red** and **White**? Again probably abstract (for some of us, anyway). But at what point in the hierarchy do things become concrete?

Do you make **PinotNoir** concrete, or is it abstract too? It looks like the Camelot Vineyards 1997 Pinot Noir is probably concrete no matter what. But how do you know for sure?

Look at the Animal inheritance tree above. Do the choices we've made for which classes are abstract and which are concrete seem appropriate? Would you change anything about the Animal inheritance tree (other than adding more Animals, of course)?

## Abstract methods

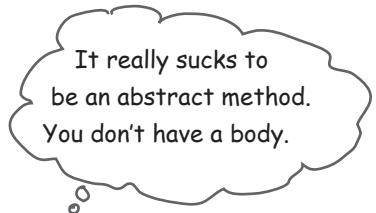
Besides classes, you can mark *methods* abstract, too. An abstract class means the class must be *extended*; an abstract method means the method must be *overridden*. You might decide that some (or all) behaviors in an abstract class don't make any sense unless they're implemented by a more specific subclass. In other words, you can't think of any generic method implementation that could possibly be useful for subclasses. What would a generic eat() method look like?

### An abstract method has no body!

Because you've already decided there isn't any code that would make sense in the abstract method, you won't put in a method body. So no curly braces—just end the declaration with a semicolon.

```
public abstract void eat();
```

No method body!  
End it with a semicolon.




**If you declare an abstract method, you MUST mark the class abstract as well. You can't have an abstract method in a non-abstract class.**

If you put even a single abstract method in a class, you have to make the class abstract. But you *can* mix both abstract and non-abstract methods in the abstract class.

---

there are no  
**Dumb Questions**

---

**Q:** What is the *point* of an abstract method? I thought the whole point of an abstract class was to have common code that could be inherited by subclasses.

**A:** Inheritable method implementations (in other words, methods with actual *bodies*) are A Good Thing to put in a superclass. *When it makes sense*. And in an abstract class, it often *doesn't* make sense, because you can't come up with any generic code that subclasses would find useful. The point of an abstract method is that even though you haven't put in any actual method code, you've still defined part of the *protocol* for a group of subtypes (subclasses).

**Q:** Which is good because...

**A:** Polymorphism! Remember, what you want is the ability to use a superclass type (often abstract) as a method argument, return type, or array type. That way, you get to add new subtypes (like a new Animal subclass) to your program without having to rewrite (or add) new methods to deal with those new types. Imagine how you'd have to change the Vet class, if it didn't use Animal as its argument type for methods. You'd have to have a separate method for every single Animal subclass! One that takes a Lion, one that takes a Wolf, one that takes a...you get the idea. So with an abstract method, you're saying, "All subtypes of this type have THIS method" for the benefit of polymorphism.

## You MUST implement all abstract methods



**Implementing an abstract method is just like overriding a method.**

Abstract methods don't have a body; they exist solely for polymorphism. That means the first concrete class in the inheritance tree must implement *all* abstract methods.

You can, however, pass the buck by being abstract yourself. If both Animal and Canine are abstract, for example, and both have abstract methods, class Canine does not have to implement the abstract methods from Animal. But as soon as we get to the first concrete subclass, like Dog, that subclass must implement *all* of the abstract methods from both Animal and Canine.

But remember that an abstract class can have both abstract and *non-abstract* methods, so Canine, for example, could implement an abstract method from Animal, so that Dog didn't have to. But if Canine says nothing about the abstract methods from Animal, Dog has to implement all of Animal's abstract methods.

When we say "you must implement the abstract method," that means you *must provide a body*. That means you must create a non-abstract method in your class with the same method signature (name and arguments) and a return type that is compatible with the declared return type of the abstract method. What you put *in* that method is up to you. All Java cares about is that the method is *there*, in your concrete subclass.



## Abstract versus Concrete classes

Let's put all this abstract rhetoric into some concrete use. In the middle column we've listed some classes. Your job is to imagine applications where the listed class might be concrete, and applications where the listed class might be abstract. We took a shot at the first few to get you going. For example, class Tree would be abstract in a tree nursery program, where differences between an Oak and an Aspen matter. But in a golf simulation program, Tree might be a concrete class (perhaps a subclass of Obstacle), because the program doesn't care about or distinguish between different types of trees. (There's no one right answer; it depends on your design.)

Concrete	Sample class	Abstract
golf course simulation	Tree	tree nursery application
_____	House	architect application
satellite photo application	Town	_____
_____	Football Player	coaching application
_____	Chair	_____
_____	Customer	_____
_____	Sales Order	_____
_____	Book	_____
_____	Store	_____
_____	Supplier	_____
_____	Golf Club	_____
_____	Carburetor	_____
_____	Oven	_____

## Polymorphism in action

Let's say that we want to write our *own* kind of list class, one that will hold Dog objects, but pretend for a moment that we don't know about the ArrayList class. For the first pass, we'll give it just an add() method. We'll use a simple Dog array (Dog[]) to keep the added Dog objects, and give it a length of 5. When we reach the limit of 5 Dog objects, you can still call the add() method, but it won't do anything. If we're *not* at the limit, the add() method puts the Dog in the array at the next available index position and then increments that next available index (nextIndex).

### Building our own Dog-specific list

(Perhaps the world's worst attempt at making our own ArrayList kind of class, from scratch.)

version 1
<b>MyDogList</b>
Dog[] dogs int nextIndex
add(Dog d)

```

public class MyDogList {
    private Dog[] dogs = new Dog[5];
    private int nextIndex = 0;

    public void add(Dog d) {
        if (nextIndex < dogs.length) {
            dogs[nextIndex] = d;
            System.out.println("Dog added at " + nextIndex);
            nextIndex++;
        }
    }
}

```

*Use a plain old Dog array behind the scenes.*

*We'll increment this each time a new Dog is added.*

*If we're not already at the limit of the dogs array, add the Dog and print a message.*

*increment, to give us the next index to use*

# Uh-oh, now we need to keep Cats, too

We have a few options here:

1. Make a separate class, MyCatList, to hold Cat objects. Pretty clunky.
2. Make a single class, DogAndCatList, that keeps two different arrays as instance variables and has two different add() methods: addCat(Cat c) and addDog(Dog d). Another clunky solution.
3. Make a heterogeneous AnimalList class that takes *any* kind of Animal subclass (since we know that if the spec changed to add Cats, sooner or later we'll have some *other* kind of animal added as well). We like this option best, so let's change our class to make it more generic, to take Animals instead of just Dogs. We've highlighted the key changes (the logic is the same, of course, but the type has changed from Dog to Animal everywhere in the code).

## Building our own Animal-specific list

```

version 2
public class MyAnimalList {
    private Animal[] animals = new Animal[5];
    private int nextIndex = 0;

    public void add(Animal a) {
        if (nextIndex < animals.length) {
            animals[nextIndex] = a;
            System.out.println("Animal added at " + nextIndex);
            nextIndex++;
        }
    }
}

```

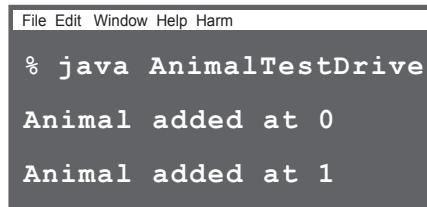
Don't panic. We're not making a new Animal object; we're making a new array object, of type Animal. (Remember, you cannot make a new instance of an abstract type, but you CAN make an array object declared to HOLD that type.)

---

```

public class AnimalTestDrive {
    public static void main(String[] args) {
        MyAnimalList list = new MyAnimalList();
        Dog dog = new Dog();
        Cat cat = new Cat();
        list.add(dog);
        list.add(cat);
    }
}

```



```

File Edit Window Help Harm
% java AnimalTestDrive
Animal added at 0
Animal added at 1

```

## What about non-Animals? Why not make a class generic enough to take anything?

You know where this is heading. We want to change the type of the array, along with the add() method argument, to something *above* Animal. Something even *more* generic, *more* abstract than Animal. But how can we do it? We don't *have* a superclass for Animal.

Then again, maybe we do...

### Every class in Java extends class Object.

Class Object is the mother of all classes; it's the superclass of *everything*.

Even if you take advantage of polymorphism, you still have to create a class with methods that take and return *your* polymorphic type. Without a common superclass for everything in Java, there'd be no way for the developers of Java to create classes with methods that could take *your* custom types...*types they never knew about when they wrote the library class*.

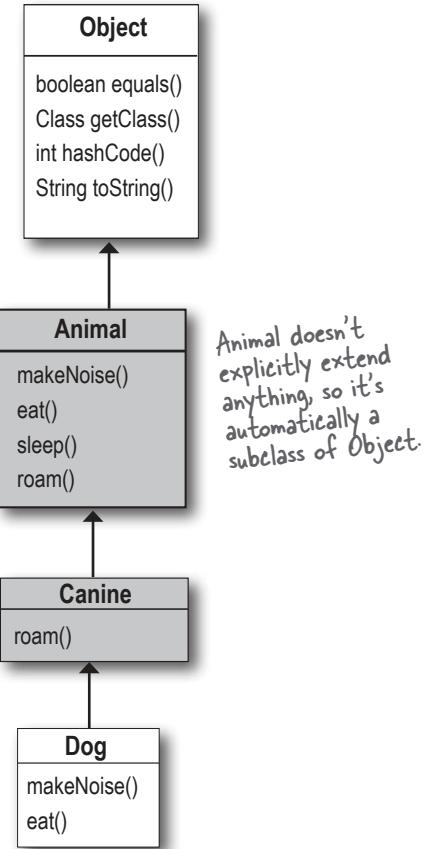
So you were making subclasses of class Object from the very beginning and you didn't even know it. **Every class you write extends Object**, without your ever having to say it. But you can think of it as though a class you write looks like this:

```
public class Dog extends Object { }
```

But wait a minute, Dog *already* extends something, Canine. That's OK. The compiler will make Canine extend Object instead. Except Canine extends Animal. No problem, then the compiler will just make Animal extend Object.

### Any class that doesn't explicitly extend another class, implicitly extends Object.

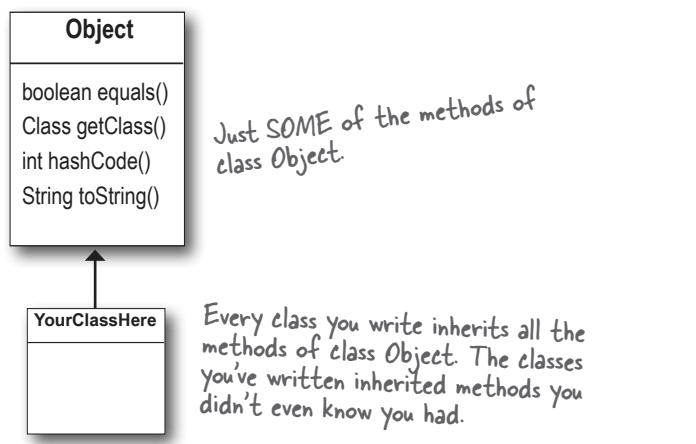
So, since Dog extends Canine, it doesn't *directly* extend Object (although it does extend it indirectly), and the same is true for Canine, but Animal *does* directly extend Object.



# So what's in this ultra-super-mega-class Object?

If you were Java, what behavior would you want *every* object to have? Hmm...let's see...how about a method that lets you find out if one object is equal to another object? What about a method that can tell you the actual class type of that object? Maybe a method that gives you a hashcode for the object, so you can use the object in hashtables (we'll talk about Java's hashtables later). Oh, here's a good one—a method that prints out a String message for that object.

And what do you know? As if by magic, class Object does indeed have methods for those four things. That's not all, though, but these are the ones we really care about.



## ① equals(Object o)

```

Dog a = new Dog();
Cat c = new Cat();

if (a.equals(c)) {
    System.out.println("true");
} else {
    System.out.println("false");
}

```

```

File Edit Window Help Stop
% java TestObject
false

```

Tells you if two objects are considered 'equal'.

## ③ hashCode()

```

Cat c = new Cat();
System.out.println(c.hashCode());

```

```

File Edit Window Help Drop
% java TestObject
8202111

```

Prints out a hashcode for the object (for now, think of it as a unique ID).

## ② getClass()

```

Cat c = new Cat();
System.out.println(c.getClass());

```

```

File Edit Window Help Paint
% java TestObject
class Cat

```

Gives you back the class that object was instantiated from.

## ④ toString()

```

Cat c = new Cat();
System.out.println(c.toString());

```

```

File Edit Window Help LapseIntoComa
% java TestObject
Cat@7d277f

```

Prints out a String message with the name of the class and some other number we rarely care about.

## Object and abstract classes

there are no  
**Dumb Questions**

**Q:** Is class Object abstract?

**A:** No. Well, not in the formal Java sense anyway. Object is a non-abstract class because it's got method implementation code that all classes can inherit and use out of the box, without having to override the methods.

**Q:** Then *can* you override the methods in Object?

**A:** Some of them. But some of them are marked *final*, which means you can't override them. You're encouraged (strongly) to override hashCode(), equals(), and toString() in your own classes, and you'll learn how to do that a little later in the book. But some of the methods, like getClass(), do things that must work in a specific, guaranteed way.

**Q:** HOW can you let somebody make an Object object? Isn't that just as weird as making an Animal object?

**A:** Good question! Why is it acceptable to make a new Object instance? Because sometimes you just want a generic object to use as, well, as an object. A *lightweight* object. For now, just stick that on the back burner and assume that you will rarely make objects of type Object, even though you *can*.

**Q:** So is it fair to say that the main purpose for type Object is so that you can use it for a polymorphic argument and return type?

**A:** The Object class serves two main purposes: to act as a polymorphic type for methods that need to work on any class that you or anyone else makes, and to provide *real* method code that all objects in Java need at runtime (and putting them in class Object means all other classes inherit them). Some of the most important methods in Object are related to threads, and we'll see those later in the book.

**Q:** If it's so good to use polymorphic types, why don't you just make ALL your methods take and return type Object?

**A:** Ahhhh...think about what would happen. For one thing, you would defeat the whole point of "type-safety," one of Java's greatest protection mechanisms for your code. With type-safety, Java guarantees that you won't ask the wrong object to do something you *meant* to ask of another object type. Like, ask a *Ferrari* (which you think is a *Toaster*) to *cook itself*. But the truth is, you *don't* have to worry about that fiery Ferrari scenario, even if you *do* use Object references for everything. Because when objects are referred to by an Object reference type, Java *thinks* it's referring to an instance of type Object. And that means the only methods you're allowed to call on that object are the ones declared in class Object! So if you were to say:

```
Object o = new Ferrari();  
o.goFast(); //Not legal!
```

You wouldn't even make it past the compiler.

Because Java is a strongly typed language, the compiler checks to make sure that you're calling a method on an object that's actually capable of *responding*. In other words, you can call a method on an object reference *only* if the class of the reference type actually *has* the method. We'll cover this in much greater detail a little later, so don't worry if the picture isn't crystal clear.

## Using polymorphic references of type Object has a price...

Before you run off and start using type Object for all your ultra-flexible argument and return types, you need to consider a little issue of using type Object as a reference. And keep in mind that we're not talking about making instances of type Object; we're talking about making instances of some other type, but using a reference of type Object.

When you put an object into an `ArrayList<Dog>`, it goes in as a Dog and comes out as a Dog:

```
ArrayList<Dog> myDogArrayList = new ArrayList<Dog>(); ← Make an ArrayList de-
← clared to hold Dog objects.  

Dog aDog = new Dog(); ← Make a Dog.  

myDogArrayList.add(aDog); ← Add the Dog to the list.  

Dog d = myDogArrayList.get(0); ← Assign the Dog from the list to a new Dog reference vari-
← able. (Think of it as though the get() method declares a Dog
return type because you used ArrayList<Dog>.)
```

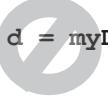
But what happens when you declare it as `ArrayList<Object>`? If you want to make an ArrayList that will literally take *any* kind of Object, you declare it like this:

```
ArrayList<Object> myDogArrayList = new ArrayList<Object>(); ← Make an ArrayList declared
← to hold any type of Object.  

Dog aDog = new Dog(); ← Make a Dog.  

myDogArrayList.add(aDog); ← Add the Dog to the list. (These two steps are the same as the
last example.)
```

But what happens when you try to get the Dog object and assign it to a Dog reference?

  
`Dog d = myDogArrayList.get(0);` NO!! Won't compile!! When you use `ArrayList<Object>`, the `get()` method returns type `Object`. The Compiler knows only that the object inherits from `Object` (somewhere in its inheritance tree) but it doesn't know it's a Dog!!

***Everything comes out of an `ArrayList<Object>` as a reference of type `Object`, regardless of what the actual object is or what the reference type was when you added the object to the list.***

The objects go IN as SoccerBall, Fish, Guitar, and Car.



**ArrayList<Object>**

But they come OUT as though they were of type Object.



Objects come out of an `ArrayList<Object>` acting like they're generic instances of class `Object`. The Compiler cannot assume the object that comes out is of any type other than `Object`.

## When a Dog loses its Dogness

# When a Dog won't act like a Dog

The problem with having everything treated polymorphically as an Object is that the objects *appear* to lose (but not permanently) their true essence. *The Dog appears to lose its dogness.* Let's see what happens when we pass a Dog to a method that returns a reference to the same Dog object, but declares the return type as type Object rather than Dog.



**BAD** ☹

```
public void go() {  
    Dog aDog = new Dog();  
    Dog sameDog = getObject(aDog);  
}  
  
public Object getObject(Object o) {  
    return o;  
}
```

We're returning a reference to the same Dog, but as a return type of Object. This part is perfectly legal. Note: this is similar to how the get() method works when you have an ArrayList<Object> rather than an ArrayList<Dog>.

This line won't work! Even though the method returned a reference to the very same Dog the argument referred to, the return type Object means the compiler won't let you assign the returned reference to anything but Object.

```
File Edit Window Help Remember  
  
DogPolyTest.java:10: incompatible types  
found   : java.lang.Object  
required: Dog  
        Dog sameDog = getObject(aDog);  
1 error
```

The compiler doesn't know that the thing returned from the method is actually a Dog, so it won't let you assign it to a Dog reference. (You'll see why on the next page.)

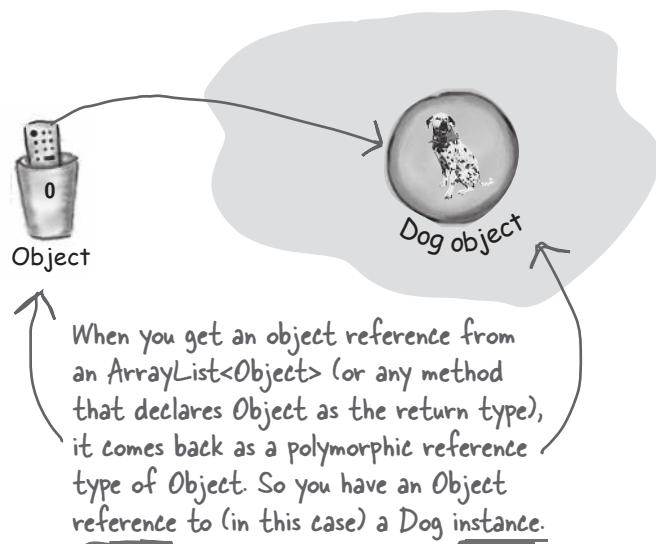
**GOOD** ☺

```
public void go() {  
    Dog aDog = new Dog();  
    Object sameDog = getObject(aDog);  
}  
  
public Object getObject(Object o) {  
    return o;  
}
```

This works (although it may not be very useful, as you'll see in a moment) because you can assign ANYTHING to a reference of type Object, since every class passes the IS-A test for Object. Every object in Java is an instance of type Object, because every class in Java has Object at the top of its inheritance tree.

## Objects don't bark

So now we know that when an object is referenced by a variable declared as type Object, it can't be assigned to a variable declared with the actual object's type. And we know that this can happen when a return type or argument is declared as type Object, as would be the case, for example, when the object is put into an ArrayList of type Object using `ArrayList<Object>`. But what are the implications of this? Is it a problem to have to use an Object reference variable to refer to a Dog object? Let's try to call Dog methods on our Dog-That-Compiler-Thinks-Is-An-Object:



```
Object o = al.get(index);
int i = o.hashCode();
```

Won't compile! → `o.bark();`

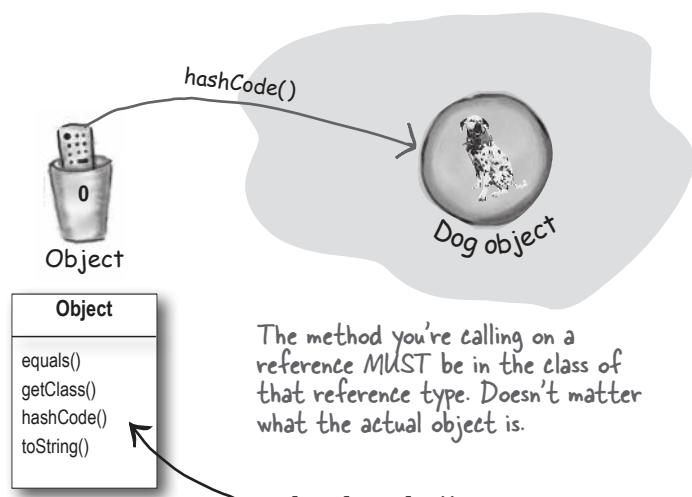
This is fine. Class `Object` has a `hashCode()` method, so you can call that method on ANY object in Java.

Can't do this!! The `Object` class has no idea what it means to `bark()`. Even though YOU know it's really a Dog at that index, the compiler doesn't.

**The compiler decides whether you can call a method based on the reference type, not the actual object type.**

Even if you *know* the object is capable ("...but it really **is** a Dog, honest..."), the compiler sees it only as a generic Object. For all the compiler knows, you put a Button object out there. Or a Microwave object. Or some other thing that really doesn't know how to bark.

The compiler checks the class of the *reference* type—not the *object* type—to see if you can call a method using that reference.



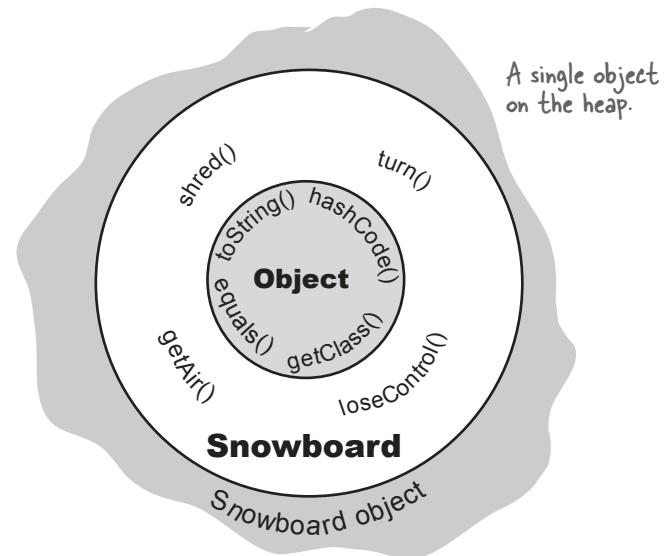
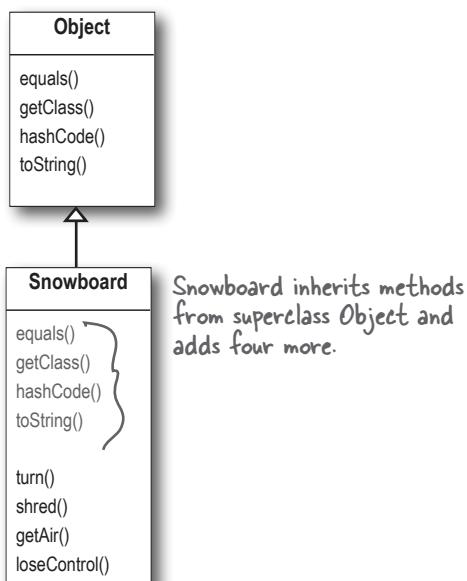
`o.hashCode();`

The "o" reference was declared as type `Object`, so you can call methods only if those methods are in class `Object`.



## Get in touch with your inner Object

An object contains *everything* it inherits from each of its superclasses. That means *every* object—regardless of its actual class type—is *also* an instance of class `Object`. That means any object in Java can be treated not just as a Dog, Button, or Snowboard, but also as an Object. When you say `new Snowboard()`, you get a single object on the heap—a Snowboard object—but that Snowboard wraps itself around an inner core representing the Object (capital “O”) portion of itself.



There is only ONE object on the heap here. A Snowboard object. But it contains both the Snowboard class parts of itself and the Object class parts of itself.

## **Polymorphism means “many forms.”**

### **You can treat a Snowboard as a Snowboard or as an Object.**

If a reference is like a remote control, the remote control takes on more and more buttons as you move down the inheritance tree. A remote control (reference) of type Object has only a few buttons—the buttons for the exposed methods of class Object. But a remote control of type Snowboard includes all the buttons from class Object, plus any new buttons (for new methods) of class Snowboard. The more specific the class, the more buttons it may have.

Of course that's not always true; a subclass might not add any new methods, but simply override the methods of its superclass. The key point is that even if the *object* is of type Snowboard, an Object *reference* to the Snowboard object can't see the Snowboard-specific methods.

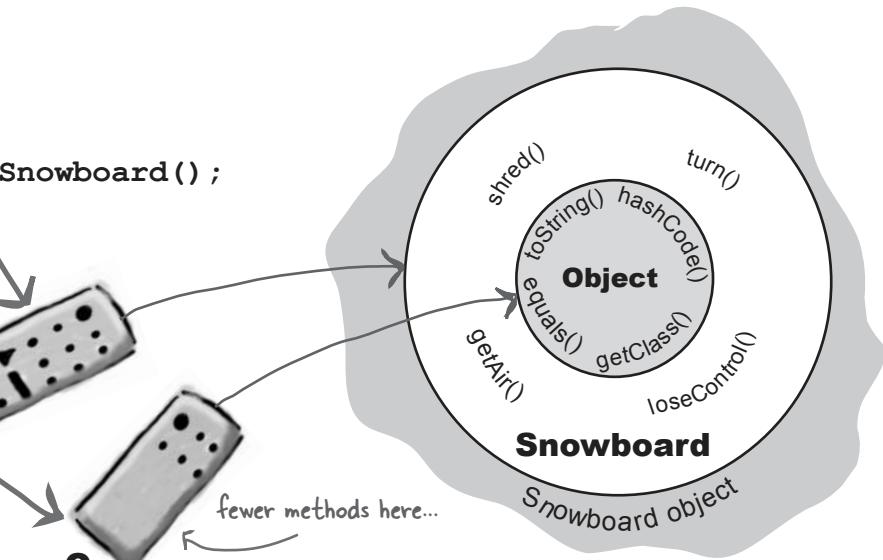
**When you put an object in an ArrayList<Object>, you can treat it only as an Object, regardless of the type it was when you put it in.**

**When you get a reference from an ArrayList<Object>, the reference is always of type Object.**

**That means you get an Object remote control.**

```
Snowboard s = new Snowboard();
Object o = s;
```

The Snowboard remote control (reference) has more buttons than an Object remote control. The Snowboard remote can see the full Snowboardness of the Snowboard object. It can access all the methods in Snowboard, including both the inherited Object methods and the methods from class Snowboard.

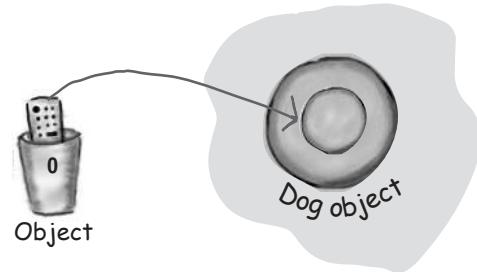


The Object reference can see only the Object parts of the Snowboard object. It can access only the methods of class Object. It has fewer buttons than the Snowboard remote control.

## casting objects

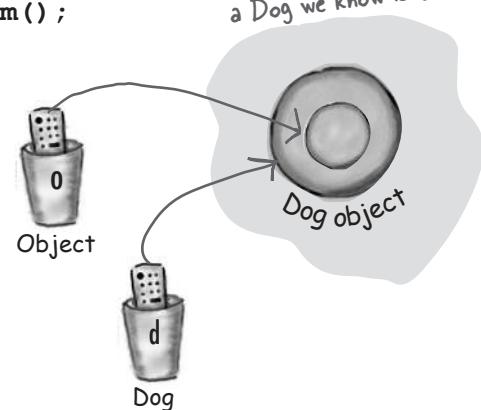


## Casting an object reference back to its *real* type.



It's really still a Dog *object*, but if you want to call Dog-specific methods, you need a *reference* declared as type Dog. If you're *sure*\* the object is really a Dog, you can make a new Dog reference to it by copying the Object reference, and forcing that copy to go into a Dog reference variable, using a cast (Dog). You can use the new Dog reference to call Dog methods.

```
Object o = al.get(index);  
Dog d = (Dog) o; ← cast the Object back to  
d.roam(); a Dog we know is there.
```



\*If you're *not* sure it's a Dog, you can use the `instanceof` operator to check. Because if you're wrong when you do the cast, you'll get a `ClassCastException` at runtime and come to a grinding halt.

```
if (o instanceof Dog) {  
    Dog d = (Dog) o;  
}
```

**So now you've seen how much Java cares about the methods in the class of the reference variable.**

**You can call a method on an object only if the class of the reference variable has that method.**

**Think of the public methods in your class as your contract, your promise to the outside world about the things you can do.**



When you write a class, you almost always *expose* some of the methods to code outside the class. To *expose* a method means you make a method *accessible*, usually by marking it public.

Imagine this scenario: you're writing code for a small business accounting program. A custom application for Simon's Surf Shop. The good re-user that you are, you found an Account class that appears to meet your needs perfectly, according to its documentation, anyway. Each account instance represents an individual customer's account with the store. So there you are minding your own business invoking the *credit()* and *debit()* methods on an Account object when you realize you need to get a balance on an account. No problem—there's a *getBalance()* method that should do nicely.

Account
debit(double amt)
credit(double amt)
double getBalance()

Except...when you invoke the *getBalance()* method, the whole thing blows up at runtime. Forget the documentation, the class does not have that method. Yikes!

But that won't happen to you, because every time you use the dot operator on a reference (*a.doStuff()*), the compiler looks at the *reference type* (the type "a" was declared to be) and checks that class to guarantee the class has the method, and that the method does indeed take the argument you're passing and return the kind of value you're expecting to get back.

**Just remember that the compiler checks the class of the reference variable, not the class of the actual object at the other end of the reference.**

## What if you need to change the contract?

OK, pretend you're a Dog. Your Dog class isn't the *only* contract that defines who you are. Remember, you inherit accessible (which usually means *public*) methods from all of your superclasses.

True, your Dog class defines a contract.

But not *all* of your contract.

**Everything in class *Canine* is part of your contract.**

**Everything in class *Animal* is part of your contract.**

**Everything in class *Object* is part of your contract.**

According to the IS-A test, you *are* each of those things—Canine, Animal, and Object.

But what if the person who designed your class had in mind the Animal simulation program, and now he wants to use you (class Dog) for a Science Fair Tutorial on Animal objects.

That's OK, you're probably reusable for that.

But what if later he wants to use you for a Pet-Shop program? *You don't have any Pet behaviors.* A Pet needs methods like *beFriendly()* and *play()*.

OK, now pretend you're the Dog class programmer. No problem, right? Just add some more methods to the Dog class. You won't be breaking anyone else's code by *adding* methods, since you aren't touching the *existing* methods that someone else's code might be calling on Dog objects.

Can you see any drawbacks to that approach (adding Pet methods to the Dog class)?



Think about what **YOU** would do if **YOU** were the Dog class programmer and needed to modify the Dog so that it could do Pet things, too. We know that simply adding new Pet behaviors (methods) to the Dog class will work, and won't break anyone else's code.

But...this is a PetShop program. It has more than just Dogs! And what if someone wants to use your Dog class for a program that has *wild* Dogs? What do you think your options might be, and without worrying about how Java handles things, just try to imagine how you'd *like* to solve the problem of modifying some of your Animal classes to include Pet behaviors.

Stop right now and think about it, **before you look at the next page** where we begin to reveal everything.\*

\*(Thus rendering the whole exercise completely useless, robbing you of your One Big Chance to burn some brain calories.)

# Let's explore some design options for reusing some of our existing classes in a PetShop program

On the next few pages, we're going to walk through some possibilities. We're not yet worried about whether Java can actually *do* what we come up with. We'll cross that bridge once we have a good idea of some of the trade-offs.

## ① Option one

We take the easy path and put pet methods in class Animal.

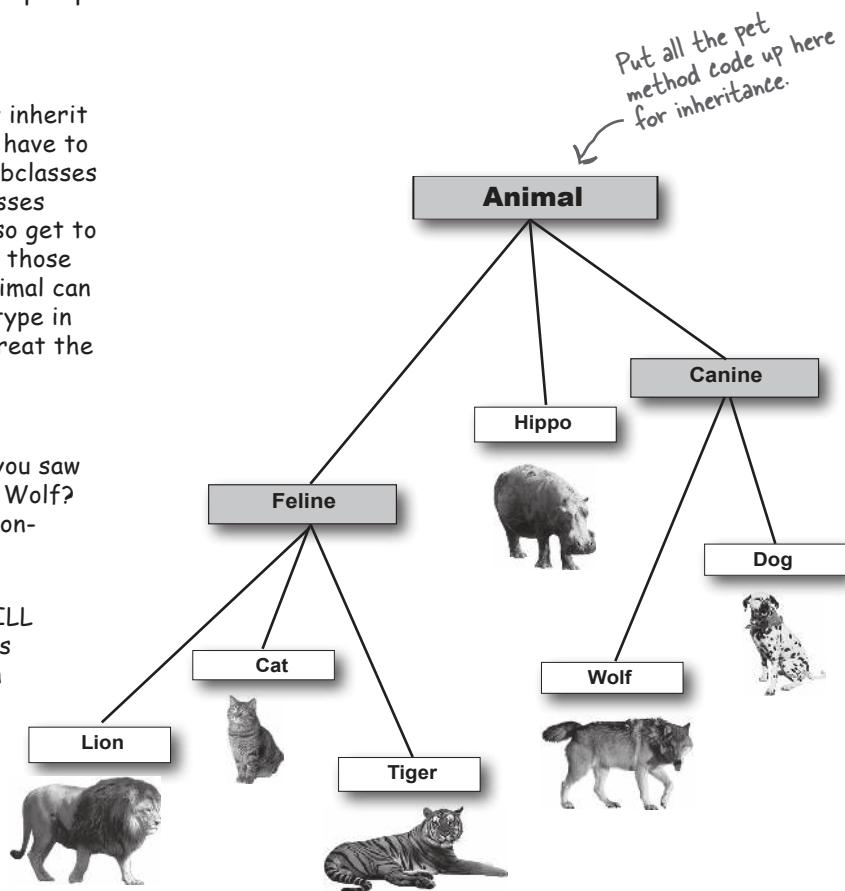
### Pros:

All the Animals will instantly inherit the pet behaviors. We won't have to touch the existing Animal subclasses at all, and any Animal subclasses created in the future will also get to take advantage of inheriting those methods. That way, class Animal can be used as the polymorphic type in any program that wants to treat the Animals as pets.

### Cons:

So...when was the last time you saw a Hippo at a pet shop? Lion? Wolf? Could be dangerous to give non-pets pet methods.

Also, we almost certainly WILL have to touch the pet classes like Dog and Cat, because (in our house, anyway) Dogs and Cats tend to implement pet behaviors VERY differently.



② Option two

We start with Option One, putting the pet methods in class Animal, but we make the methods abstract, forcing the Animal subclasses to override them.

Pros:

That would give us all the benefits of option one, but without the drawback of having non-pet Animals running around with pet methods (like `beFriendly()`). All Animal classes would have the method (because it's in class Animal), but because it's abstract, the non-pet Animal classes won't inherit any functionality. All classes MUST override the methods, but they can make the methods "do-nothings."

Put all the pet methods up here, but with no implementations. Make all pet methods abstract.

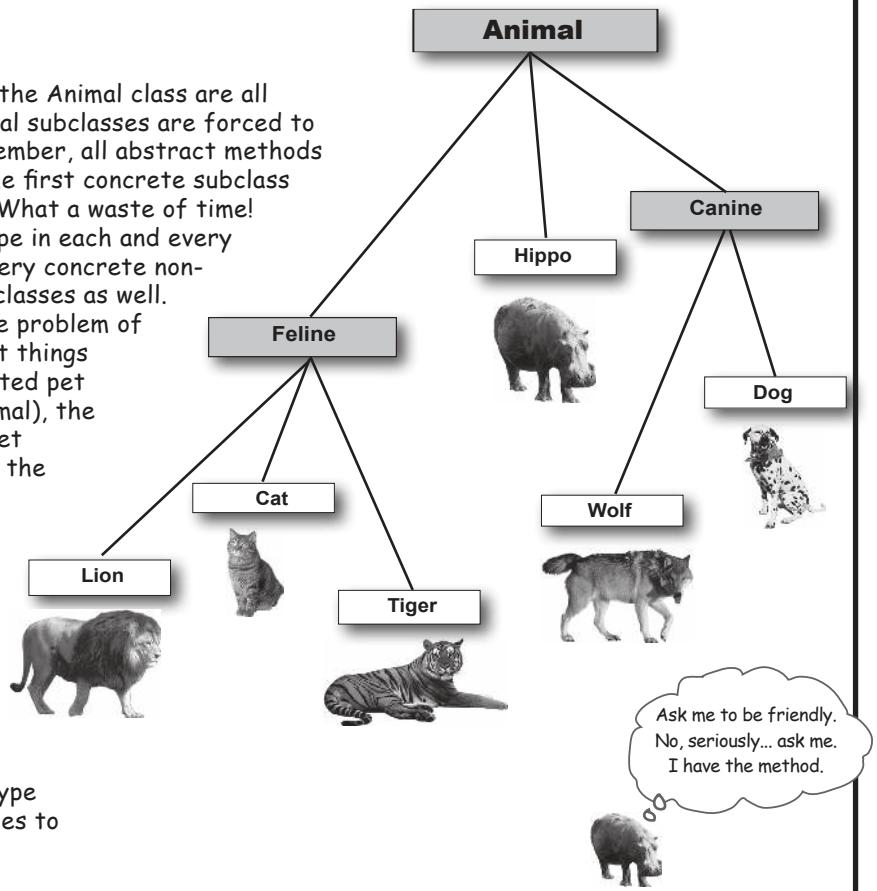
Cons:

Because the pet methods in the Animal class are all abstract, the concrete Animal subclasses are forced to implement all of them. (Remember, all abstract methods MUST be implemented by the first concrete subclass down the inheritance tree.) What a waste of time!

You have to sit there and type in each and every pet method into each and every concrete non-pet class, and all future subclasses as well.

And while this does solve the problem of non-pets actually DOING pet things (as they would if they inherited pet functionality from class Animal), the contract is bad. Every non-pet class would be announcing to the world that it, too, has those pet methods, even though the methods wouldn't actually DO anything when called.

This approach doesn't look good at all. It just seems wrong to stuff everything into class Animal that more than one Animal type might need, UNLESS it applies to ALL Animal subclasses.



### ③ Option three

Put the pet methods ONLY in the classes where they belong.

#### Pros:

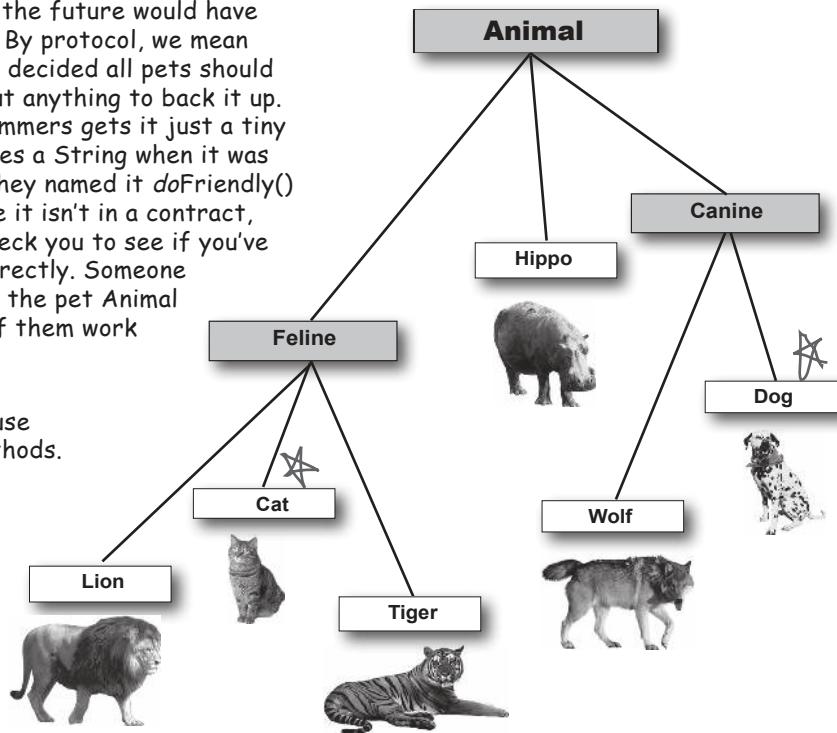
No more worries about Hippos greeting you at the door or licking your face. The methods are where they belong, and ONLY where they belong. Dogs can implement the methods and Cats can implement the methods, but nobody else has to know about them.

#### Cons:

Two Big Problems with this approach. First off, you'd have to agree to a protocol, and all programmers of pet Animal classes now and in the future would have to KNOW about the protocol. By protocol, we mean the exact methods that we've decided all pets should have. The pet contract without anything to back it up. But what if one of the programmers gets it just a tiny bit wrong? Like, a method takes a String when it was supposed to take an int? Or they named it *doFriendly()* instead of *beFriendly()*? Since it isn't in a contract, the compiler has no way to check you to see if you've implemented the methods correctly. Someone could easily come along to use the pet Animal classes and find that not all of them work quite right.

And second, you don't get to use polymorphism for the pet methods. Every class that needs to use pet behaviors would have to know about each and every class! In other words, you can't use *Animal* as the polymorphic type now, because the compiler won't let you call a Pet method on an *Animal* reference (even if it's really a *Dog* object) because class *Animal* doesn't have the method.

~~Put the pet methods ONLY in the Animal classes that can be pets, instead of in Animal.~~

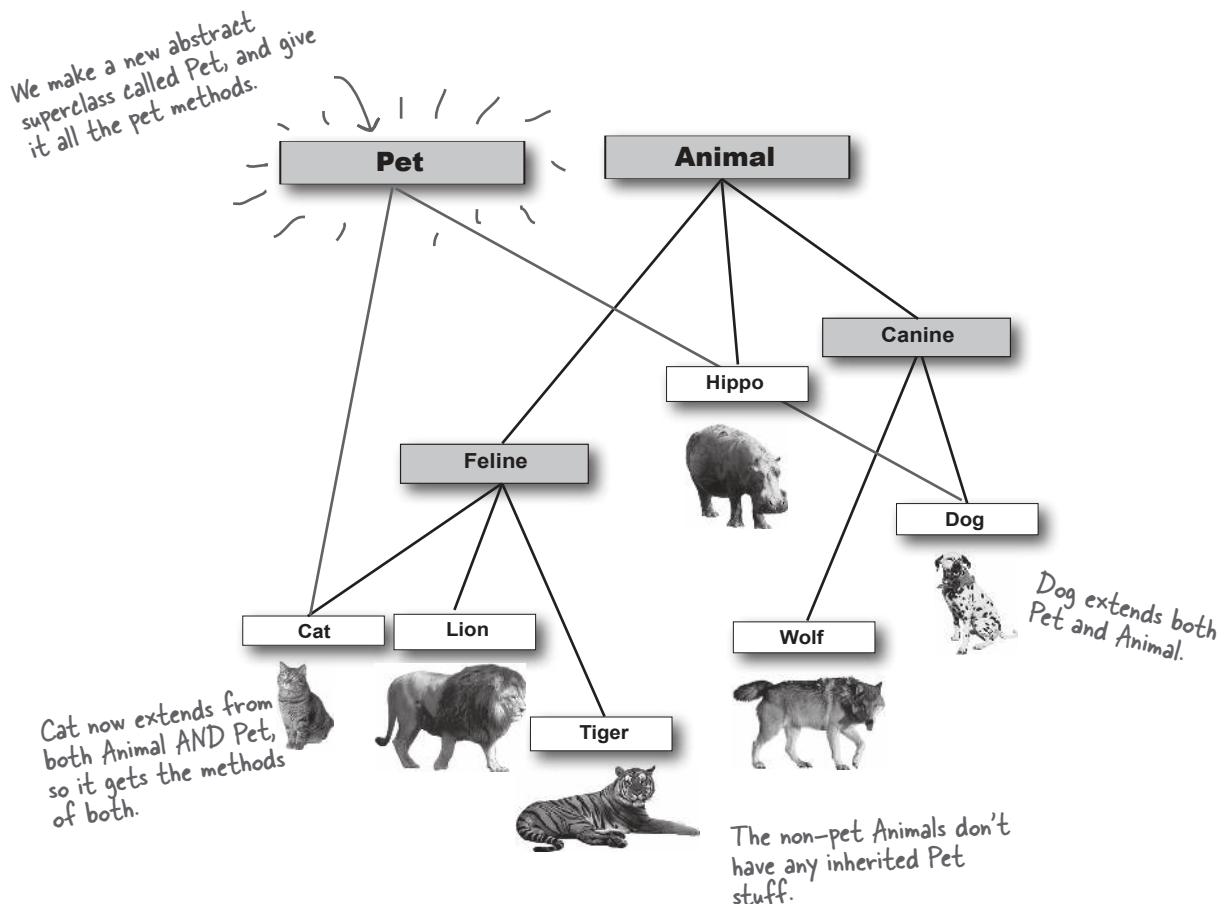


multiple inheritance?

## So what we **REALLY** need is:

- ↗ A way to have pet behavior in **just** the pet classes
- ↗ A way to guarantee that all pet classes have all of the same methods defined (same name, same arguments, same return types, no missing methods, etc.), without having to cross your fingers and hope all the programmers get it right
- ↗ A way to take advantage of polymorphism so that all pets can have their pet methods called, without having to use arguments, return types, and arrays for each and every pet class

**It looks like we need TWO superclasses at the top.**



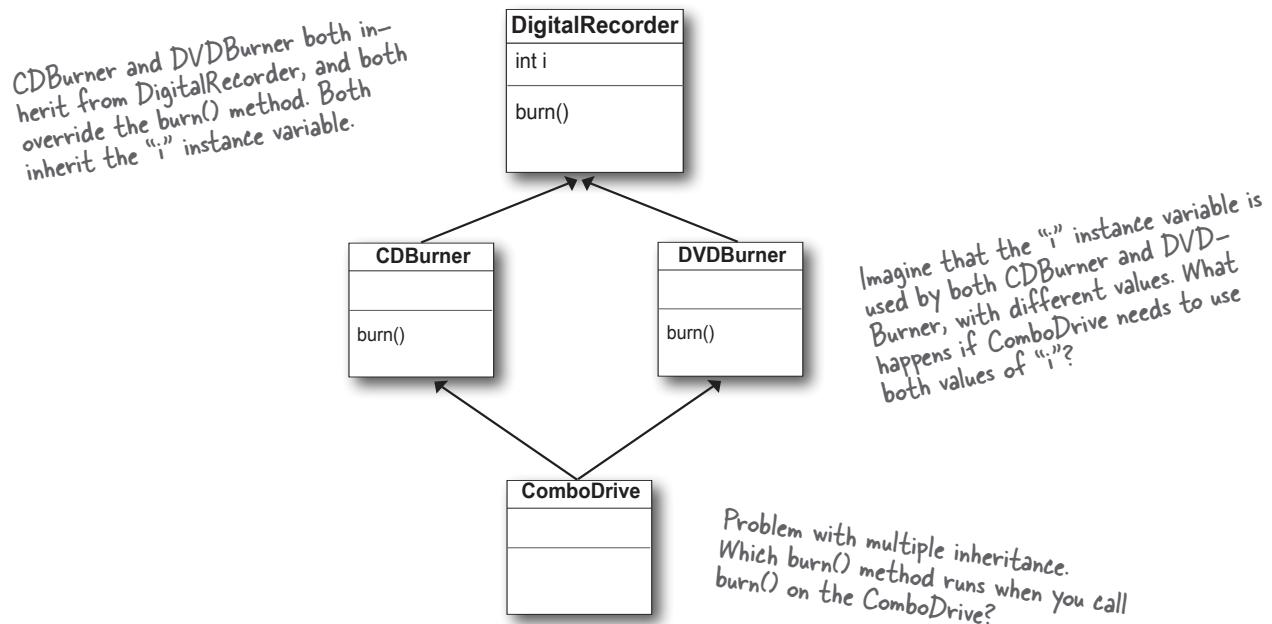
There's just one problem with the "two superclasses" approach...

## It's called "multiple inheritance," and it can be a Really Bad Thing.

That is, if it were possible to do in Java.

But it isn't, because multiple inheritance has a problem known as The Deadly Diamond of Death.

### Deadly Diamond of Death



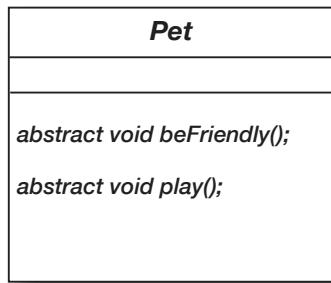
A language that allows the Deadly Diamond of Death can lead to some ugly complexities, because you have to have special rules to deal with the potential ambiguities. And extra rules means extra work for you both in *learning* those rules and watching out for those "special cases." Java is supposed to be *simple*, with consistent rules that don't blow up under some scenarios. So Java (unlike C++) protects you from having to think about the Deadly Diamond of Death. But that brings us back to the original problem! *How do we handle the Animal/Pet thing?*

## Interface to the rescue!

Java gives you a solution. An *interface*. Not a *GUI* interface, not the generic use of the *word* interface as in, “That’s the public interface for the Button class API,” but the Java *keyword* **interface**.

A Java interface solves your multiple inheritance problem by giving you much of the polymorphic *benefits* of multiple inheritance without the pain and suffering from the Deadly Diamond of Death (DDD).

The way in which interfaces side-step the DDD is surprisingly simple: **make all the methods abstract!** That way, the subclass **must** implement the methods (remember, abstract methods *must* be implemented by the first concrete subclass), so at runtime the JVM isn’t confused about *which* of the two inherited versions it’s supposed to call.



**A Java interface is like a 100% pure abstract class.**

All methods in an interface are abstract, so any class that IS-A Pet MUST implement (i.e., override) the methods of Pet.

### To DEFINE an interface:

```
public interface Pet { ... }
```

↑  
Use the keyword “interface” instead of “class.”

### To IMPLEMENT an interface:

```
public class Dog extends Canine implements Pet { ... }
```

↑  
Use the keyword “implements” followed by the interface name. Note that when you implement an interface, you still get to extend a class.

# Making and implementing the Pet interface

You say "interface"  
instead of "class" here.

```
public interface Pet {
    public abstract void beFriendly();
    public abstract void play();
}
```

Interface methods are implicitly public and is optional (in fact, it's not considered "good style" to type the words in, but we did here just to reinforce it).

All interface methods are abstract,  
so they MUST end in semicolons.  
Remember, they have no body!

Dog IS-A Animal  
and Dog IS-A Pet

```
public class Dog extends Canine implements Pet {
    public void beFriendly() {...}
    public void play() {...}
    public void roam() {...}
    public void eat() {...}
}
```

You say "implements"  
followed by the name  
of the interface.

You SAID you are a Pet, so you MUST implement the Pet methods. It's your instead of semicolons.

These are just normal  
overriding methods.

## there are no Dumb Questions

**Q:** Wait a minute, interfaces don't really give you multiple inheritance, because you can't put any implementation code in them. If all the methods are abstract, what does an interface really buy you?

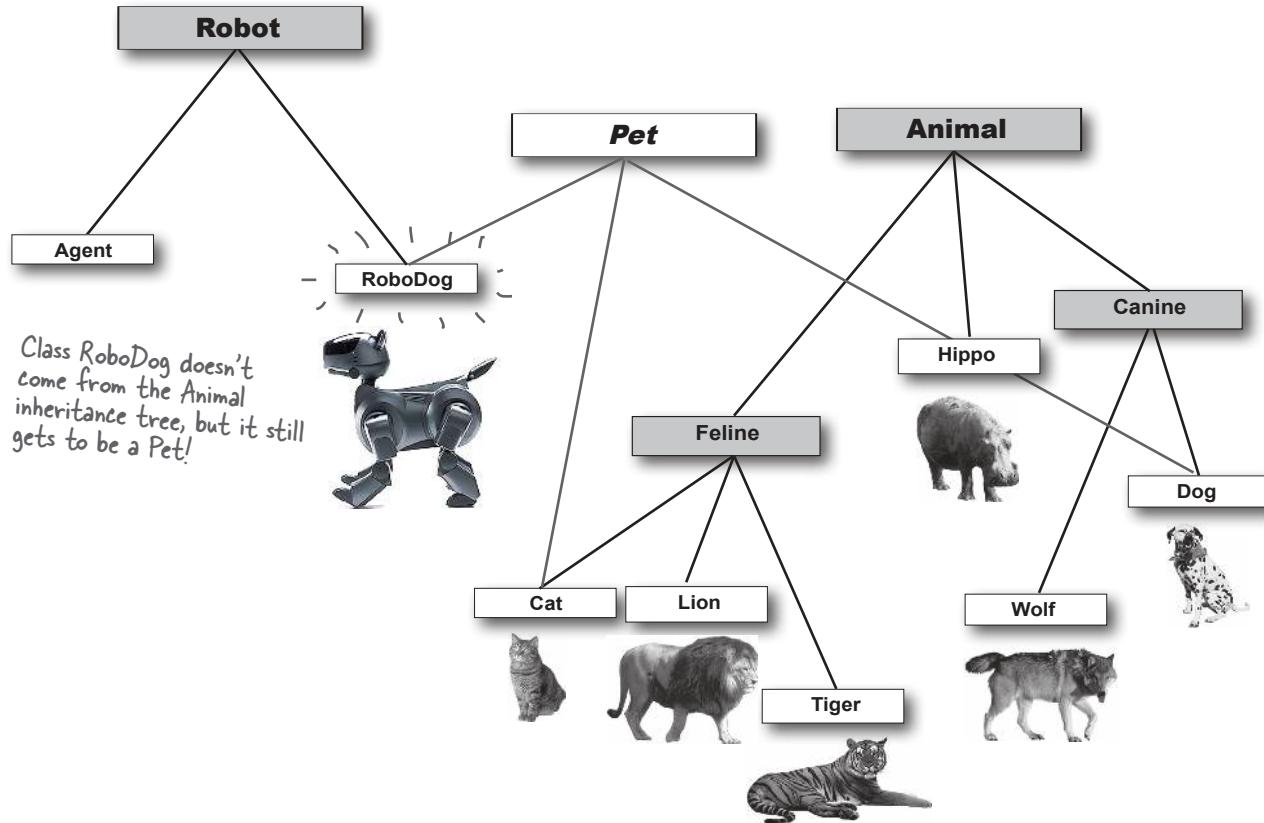
**A:** Well, actually...there are cases where interfaces can have implementation code (static and default methods, for example), but we're not going to go into them here.

The main purpose of interfaces is polymorphism, polymorphism, polymorphism. Interfaces are the ultimate in flexibility, because if you use interfaces instead of concrete classes (or even abstract classes) as arguments and return types, you can pass anything that implements that interface. And with an interface, a class doesn't have to come from just one inheritance tree. A class can extend one class, and implement an interface. But another class might implement the same interface, yet come from a completely different inheritance tree!

So you get to treat an object by the role it plays, rather than by the class type from which it was instantiated.

In fact, if you write your code using interfaces, you don't even have to give anyone a superclass to extend. You can just give them the interface and say, "Here, I don't care what kind of class inheritance structure you come from, just implement this interface and you'll be good to go."

## Classes from different inheritance trees can implement the same interface.



When you use a *class* as a polymorphic type (like an array of type Animal or a method that takes a Canine argument), the objects you can stick in that type must be from the same inheritance tree. But not just anywhere in the inheritance tree; the objects must be from a class that is a subclass of the polymorphic type. An argument of type Canine can accept a Wolf and a Dog, but not a Cat or a Hippo.

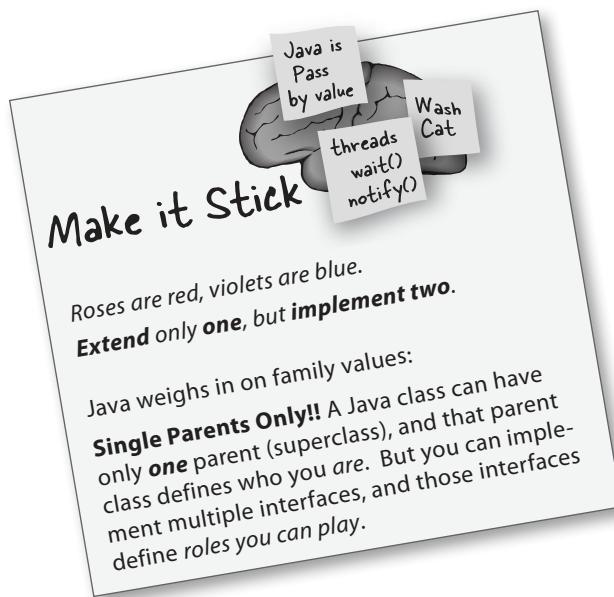
But when you use an **interface** as a polymorphic type (like an array of Pets), the objects can be from *anywhere* in the inheritance tree. The only requirement is that the objects are from a class that *implements* the interface. Allowing classes in different inheritance trees to implement a common interface is crucial in the Java API. Do you want an object to be able to save its state to a file? Implement the Serializable interface. Do you need objects to run their methods in a separate thread of execution?

Implement Runnable. You get the idea. You'll learn more about Serializable and Runnable in later chapters, but for now, remember that classes from *any* place in the inheritance tree might need to implement those interfaces. Nearly *any* class might want to be saveable or runnable.

### Better still, a class can implement multiple interfaces!

A Dog object IS-A Canine, and IS-A Animal, and IS-A Object, all through inheritance. But a Dog IS-A Pet through interface implementation, and the Dog might implement other interfaces as well. You could say:

```
public class Dog extends Animal implements Pet, Saveable, Paintable { ... }
```



### How do you know whether to make a class, a subclass, an abstract class, or an interface?

- Make a class that doesn't extend anything (other than `Object`) when your new class doesn't pass the IS-A test for any other type.
- Make a subclass (in other words, extend a class) only when you need to make a **more specific** version of a class and need to override or add new behaviors.
- Use an abstract class when you want to define a **template** for a group of subclasses, and you have at least *some* implementation code that all subclasses could use. Make the class abstract when you want to guarantee that nobody can make objects of that type.
- Use an interface when you want to define a **role** that other classes can play, regardless of where those classes are in the inheritance tree.

# Invoking the superclass version of a method

*there are no  
Dumb Questions*

**Q:** What if you make a concrete subclass and you need to override a method, but you want the behavior in the superclass version of the method? In other words, what if you don't need to *replace* the method with an override, but you just want to *add* to it with some additional specific code.

**A:** Ahhh...think about the meaning of the word *extends*. One area of good OO design looks at how to design concrete code that's *meant* to be overridden. In other words, you write method code in, say, an abstract class, that does work that's generic enough to support typical concrete implementations. But, the concrete code isn't enough to handle *all* of the subclass-specific work. So the subclass overrides the method and *extends* it by adding the rest of the code. The keyword *super* lets you invoke a superclass version of an overridden method, from within the subclass.

If method code inside a BuzzwordReport subclass says:  
**super.runReport();**

the runReport() method inside the superclass Report will run

## super.runReport();

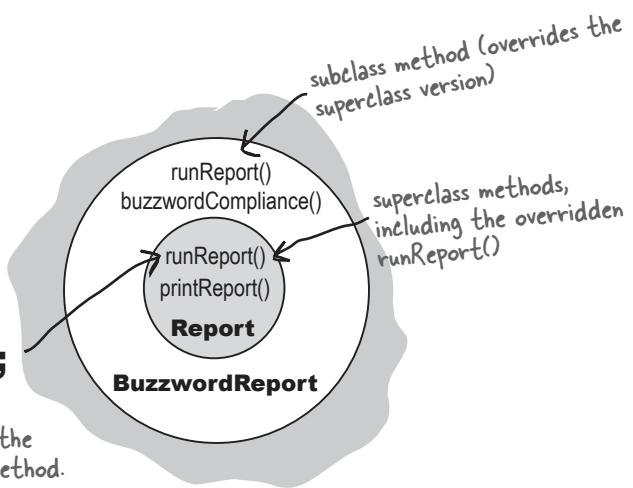
A reference to the subclass object (BuzzwordReport) will always call the subclass version of an overridden method. That's polymorphism. But the subclass code can call *super.runReport()* to invoke the superclass version.

```
abstract class Report {
    void runReport() {
        // set up report
    }
    void printReport() {
        // generic printing
    }
}
```

```
class BuzzwordsReport extends Report {
    void runReport() {
        super.runReport();
        buzzwordCompliance();
        printReport();
    }
    void buzzwordCompliance() { ... }
}
```

superclass version of the method does important stuff that subclasses could use

call superclass version; then come back and do some subclass-specific stuff



The *super* keyword is really a reference to the superclass portion of an object. When subclass code uses *super*, as in *super.runReport()*, the superclass version of the method will run.

**BULLET POINTS**

- When you don't want a class to be instantiated (in other words, you don't want anyone to make a new object of that class type), mark the class with the **abstract** keyword.
- An abstract class can have both abstract and non-abstract methods.
- If a class has even *one* abstract method, the class must be marked abstract.
- An abstract method has no body, and the declaration ends with a semicolon (no curly braces).
- All abstract methods must be implemented in the first concrete subclass in the inheritance tree.
- Every class in Java is either a direct or indirect subclass of class **Object** (`java.lang.Object`).
- Methods can be declared with **Object** arguments and/or return types.
- You can call methods on an object *only* if the methods are in the class (or interface) used as the *reference* variable type, regardless of the actual *object* type. So, a reference variable of type **Object** can be used only to call methods defined in class **Object**, regardless of the type of the object to which the reference refers.
- When a method is invoked, it will use the object type's implementation of that method.
- A reference variable of type **Object** can't be assigned to any other reference type without a *cast*. A cast can be used to assign a reference variable of one type to a reference variable of a subtype, but at runtime the cast will fail if the object on the heap is NOT of a type compatible with the cast.

Example: `Dog d = (Dog) x.getObject(aDog);`

- All objects come out of an `ArrayList<Object>` as type **Object** (meaning, they can be referenced only by an **Object** reference variable, unless you use a *cast*).
  - Multiple inheritance is not allowed in Java, because of the problems associated with the Deadly Diamond of Death. That means you can extend only one class (i.e., you can have only one immediate superclass).
  - Create an interface using the **interface** keyword instead of the word **class**.
  - Implement an interface using the keyword **implements**.
- Example: `Dog implements Pet`
- Your class can implement multiple interfaces.
  - A class that implements an interface *must* implement all the methods of the interface, except default and static methods (which we'll see in Chapter 12).
  - To invoke the superclass version of a method from a subclass that's overridden the method, use the **super** keyword. Example: `super.runReport();`

there are no  
Dumb Questions

**Q:** There's still something strange here...you never explained how it is that `ArrayList<Dog>` gives back Dog references that don't need to be cast. What's the special trick going on when you say `ArrayList<Dog>`?

**A:** You're right for calling it a special trick. In fact, it is a special trick that `ArrayList<Dog>` gives back Dogs without you having to do any cast, since it looks like `ArrayList` methods don't know anything about Dogs, or any type besides **Object**.

The short answer is that *the compiler puts in the cast for you!* When you say `ArrayList<Dog>`, there is no special class that has methods to take and return Dog objects, but instead the `<Dog>` is a signal to the compiler that you want the compiler to let you put ONLY Dog objects in and to stop you if you try to add any other type to the list. And since the compiler stops you from adding anything but Dogs to the `ArrayList`, the compiler also knows that it's safe to cast anything that comes out of that `ArrayList` to a Dog reference. In other words, using `ArrayList<Dog>` saves you from having to cast the Dog you get back. But it's much more important than that... because remember, a cast can fail at runtime, and wouldn't you rather have your errors happen at compile time rather than, say, when your customer is using it for something critical?

But there's a lot more to this story, and we'll get into all the details in Chapter 11, *Data Structures*.

## exercise: What's the Picture?



Here's your chance to demonstrate your artistic abilities. On the left you'll find sets of class and interface declarations. Your job is to draw the associated class diagrams on the right. We did the first one for you. Use a dashed line for "implements" and a solid line for "extends."

Given:

1. public interface Foo { }  
public class Bar implements Foo { }

2. public interface Vinn { }  
public abstract class Vout implements Vinn { }

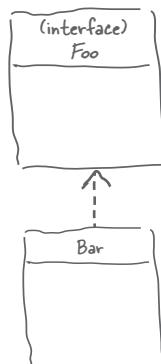
3. public abstract class Muffie implements Whuffle { }  
public class Fluffie extends Muffie { }  
public interface Whuffle { }

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 { }

What's the Picture ?

1.

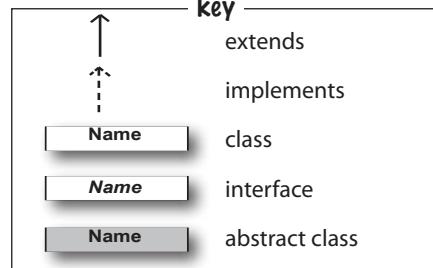


2.

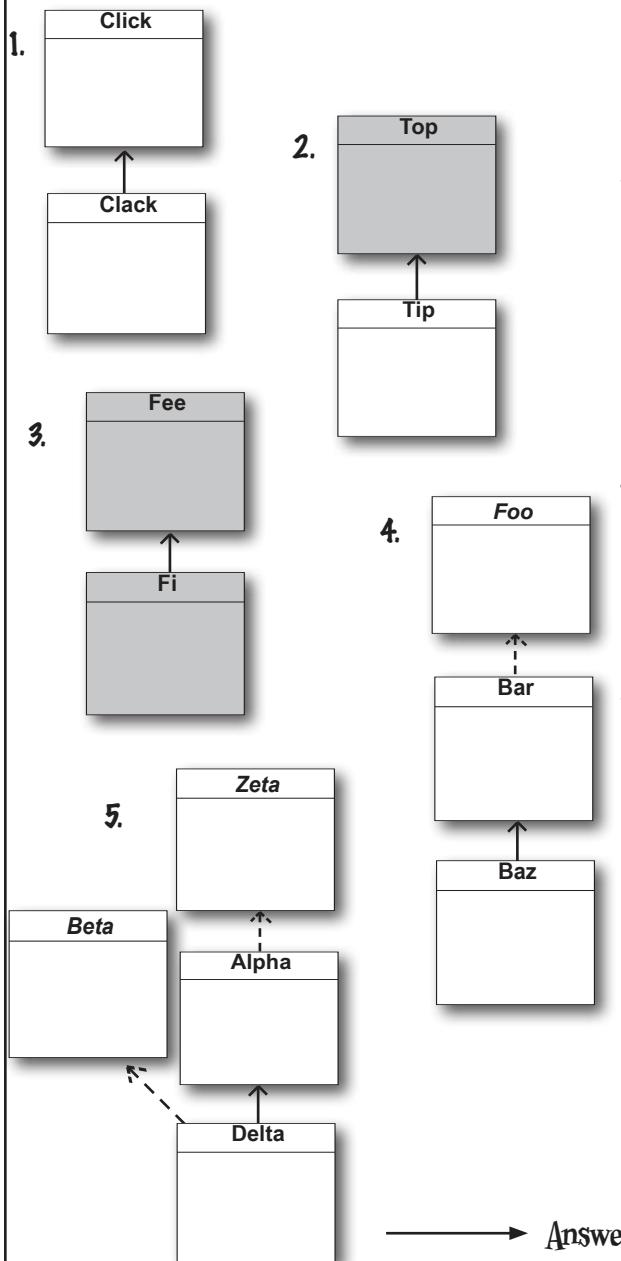
3.

4.

5.



→ Answers on page 235.

**Given:**

KEY	
	extends
	implements
	class
	interface
	abstract class

## puzzle: Pool Puzzle



# Pool Puzzle



Your **job** is to take code snippets from the pool and place them into the blank lines in the code and output. You **may** use the same snippet more than once, and you won't need to use all the snippets. Your **goal** is to make a set of classes that will compile and run and produce the output listed.

```
_____ Nose {  
_____ }  
  
abstract class Picasso implements _____ {  
_____ return 7;  
}  
}  
  
class _____ { }  
  
class _____ {  
_____ return 5;  
}  
}
```

**Note:** Each snippet from the pool can be used more than once!

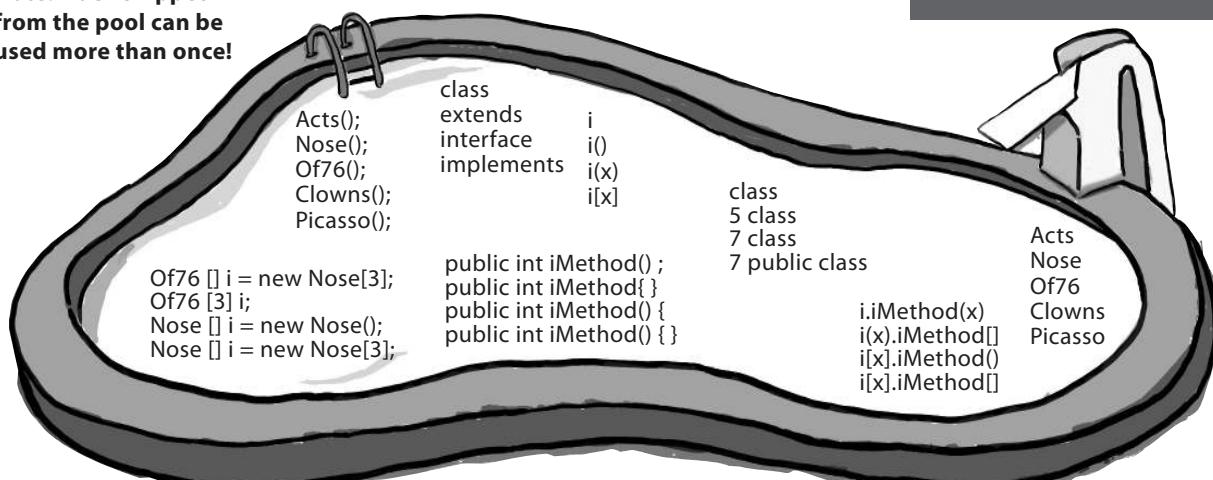
Acts();  
Nose();  
Of76();  
Clowns();  
Picasso();  
  
Of76 [] i = new Nose[3];  
Of76 [3] i;  
Nose [] i = new Nose();  
Nose [] i = new Nose[3];

class  
extends  
interface  
implements  
i  
i()  
i(x)  
i[x]  
  
public int iMethod();  
public int iMethod{}  
public int iMethod()  
public int iMethod() {}

```
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());  
}  
}  
}
```

### Output

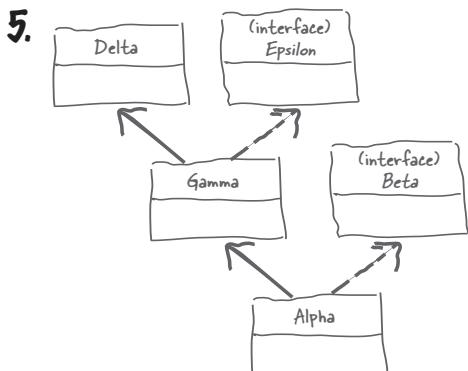
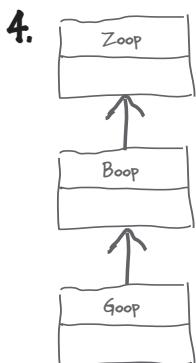
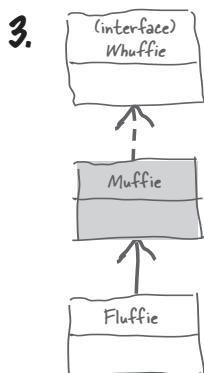
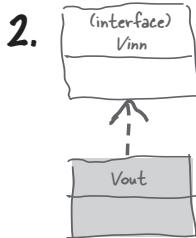
```
File Edit Window Help BeAfraid  
%java _____  
5 class Acts  
7 class Clowns  
_____ Of76
```





## Exercise Solutions

### What's the Picture? (from page 232)



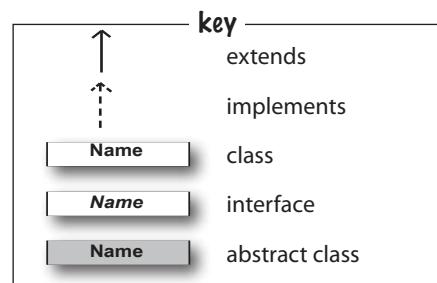
### What's the Declaration? (from page 233)

2. public abstract class Top { }  
public class Tip extends Top { }

3. public abstract class Fee { }  
public abstract class Fi 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 { }



puzzle solution



## Pool Puzzle

(from page 234)

```
interface Nose {  
    public int iMethod();  
}  
  
abstract class Picasso implements Nose {  
    public int iMethod(){  
        return 7;  
    }  
}  
  
class Clowns extends Picasso {}  
  
class Acts extends Picasso {  
    public int iMethod(){  
        return 5;  
    }  
}
```

```
public class Of76 extends Clowns {  
    public static void main(String[] args) {  
        Nose[] i = new Nose [3];  
        i[0] = new Acts();  
        i[1] = new Clowns();  
        i[2] = new Of76();  
        for (int x = 0; x < 3; x++) {  
            System.out.println(i[x].iMethod()  
                + " " + i[x].getClass());  
        }  
    }  
}
```

**Output**

```
File Edit Window Help KillTheMime  
%java Of76  
5 class Acts  
7 class Clowns  
7 class Of76
```

# Life and Death of an Object



...then he said,  
"I can't feel my legs!"  
and I said "Joe! Stay with me  
Joe!" But it was...too late. The garbage  
collector came and...he was gone. Best  
object I ever had. Gone.

### Objects are born and objects die.

You're in charge of an object's lifecycle. You decide when and how to **construct** it. You decide when to **destroy** it. Except you don't actually *destroy* the object yourself, you simply *abandon* it. But once it's abandoned, the heartless **Garbage Collector (gc)** can vaporize it, reclaiming the memory that object was using. If you're gonna write Java, you're gonna create objects. Sooner or later, you're gonna have to let some of them go, or risk running out of RAM. In this chapter we look at how objects are created, where they live while they're alive, and how to keep or abandon them efficiently. That means we'll talk about the heap, the stack, scope, constructors, superclass constructors, null references, and more. Warning: this chapter contains material about object death that some may find disturbing. Best not to get too attached.

# The Stack and the Heap: where things live

Before we can understand what really happens when you create an object, we have to step back a bit. We need to learn more about where everything lives (and for how long) in Java. That means we need to learn more about two areas of memory—the Stack and the Heap. When a JVM starts up, it gets a chunk of memory from the underlying OS and uses it to run your Java program. How *much* memory, and whether or not you can tweak it, is dependent on which version of the JVM (and on which platform) you’re running. But usually you *won’t* have any say in the matter. And with good programming, you probably won’t care (more on that a little later).

In Java, we (programmers) care about the area of memory where objects live (the heap) and the one where method invocations and local variables live (the stack).

We know that all *objects* live on the garbage-collectible heap, but we haven’t yet looked at where *variables* live. And where a variable lives depends on what *kind* of variable it is. And by “kind,” we don’t mean *type* (i.e., primitive or object reference). The two *kinds* of variables whose lives we care about now are *instance* variables and *local* variables. Local variables are also known as *stack* variables, which is a big clue for where they live.

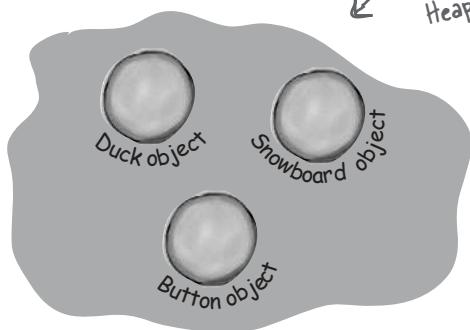
## The Stack

Where method invocations and local variables live



## The Heap

Where **ALL** objects live



Also known as “The Garbage-Collectible Heap”

## Instance Variables

**Instance variables are declared inside a class but not inside a method.** They represent the “fields” that each individual object has (which can be filled with different values for each instance of the class). Instance variables live inside the object they belong to.

```
public class Duck {
    int size; ← Every Duck has a "size"
}
```

## Local Variables

**Local variables are declared inside a method, including method parameters.** They’re temporary and live only as long as the method is on the stack (in other words, as long as the method has not reached the closing curly brace).

```
public void foo(int x) {
    int i = x + 3;
    boolean b = true;
}
```

The parameter x and the variables i and b are all local variables.

# Methods are stacked

When you call a method, the method lands on the top of a call stack. That new thing that's actually pushed onto the stack is the *stack frame*, and it holds the state of the method including which line of code is executing, and the values of all local variables.

The method at the *top* of the stack is always the currently running method for that stack (for now, assume there's only one stack, but in Chapter 14, *A Very Graphic Story*, we'll add more.) A method stays on the stack until the method hits its closing curly brace (which means the method's done). If method `foo()` calls method `bar()`, method `bar()` is stacked on top of method `foo()`.

```
public void doStuff() {
    boolean b = true;
    go(4);
}

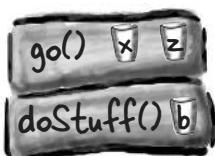
public void go(int x) {
    int z = x + 24;
    crazy();
    // imagine more code here
}

public void crazy() {
    char c = 'a';
}
```

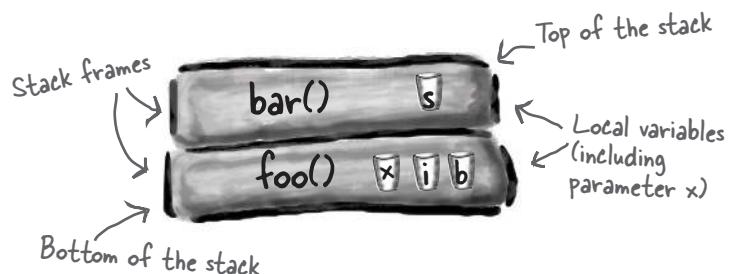
- ① Code from another class calls `doStuff()`, and `doStuff()` goes into a stack frame at the top of the stack. The boolean variable named "b" goes on the `doStuff()` stack frame.



- ② `doStuff()` calls `go()`, and `go()` is pushed on top of the stack. Variables "x" and "z" are in the `go()` stack frame.



A call stack with two methods

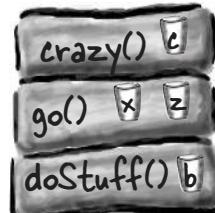


**The method on the top of the stack is always the currently executing method.**

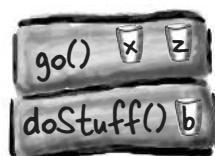
## A stack scenario

The code on the left is a snippet (we don't care what the rest of the class looks like) with three methods. The first method (`doStuff()`) calls the second method (`go()`), and the second method calls the third (`crazy()`). Each method declares one local variable within the body of the method (`b`, `z`, and `c`), and method `go()` also declares a parameter variable (which means `go()` has two local variables, `x` and `z`).

- ③ `go()` calls `crazy()`. `crazy()` is now on the top of the stack, with variable "c" in the frame.



- ④ `crazy()` completes, and its stack frame is popped off the stack. Execution goes back to the `go()` method and picks up at the line following the call to `crazy()`.

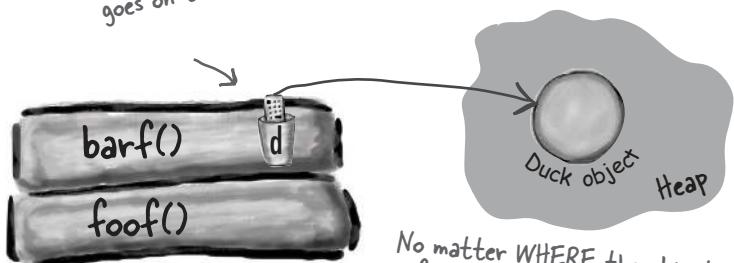


## What about local variables that are objects?

Remember, a non-primitive variable holds a *reference* to an object, not the object itself. You already know where objects live—on the heap. It doesn't matter where they're declared or created. **If the local variable is a reference to an object, only the variable (the reference/remote control) goes on the stack.**

```
public class StackRef {  
    public void foof() {  
        barf();  
    }  
  
    public void barf() {  
        Duck d = new Duck();  
    }  
}
```

barf() declares and creates a new Duck reference variable "d" (since it's declared inside the method, it's a local variable and goes on the stack).



No matter WHERE the object reference variable is declared (inside a method vs. as an instance variable of a class), the object always, always, always goes on the heap.

### there are no Dumb Questions

**Q:** One more time, WHY are we learning the whole stack/heap thing? How does this help me? Do I really need to learn about it?

**A:** Knowing the fundamentals of the Java Stack and Heap is crucial if you want to understand variable scope, object creation issues, memory management, threads, and exception handling. We cover threads and exception handling in later chapters. You do not need to know anything about how the Stack and Heap are implemented in any particular JVM and/or platform. Everything you need to know about the Stack and Heap is on this page and the previous one. If you nail these pages, all the other topics that depend on your knowing this stuff will go much, much, much easier. Once again, some day you will SO thank us for shoving Stacks and Heaps down your throat.

### BULLET POINTS

- Java has two areas of memory we care about: the Stack and the Heap.
- Instance variables are variables declared inside a class but outside any method.
- Local variables are variables declared inside a method or method parameter.
- All local variables live on the stack, in the frame corresponding to the method where the variables are declared.
- Object reference variables work just like primitive variables—if the reference is declared as a local variable, it goes on the stack.
- All objects live in the heap, regardless of whether the reference is a local or instance variable.

## If local variables live on the stack, where do instance variables live?

When you say `new CellPhone()`, Java has to make space on the Heap for that `CellPhone`. But how *much* space? Enough for the object, which means enough to house all of the object's instance variables. That's right, instance variables live on the Heap, *inside* the object they belong to.

Remember that the *values* of an object's instance variables live inside the object. If the instance variables are all primitives, Java makes space for the instance variables based on the primitive type. An `int` needs 32 bits, a `long` 64 bits, etc. Java doesn't care about the value inside primitive variables; the bit-size of an `int` variable is the same (32 bits) whether the value of the `int` is 32,000,000 or 32.

But what if the instance variables are *objects*? What if `CellPhone HAS-A Antenna`? In other words, `CellPhone` has a reference variable of type `Antenna`.

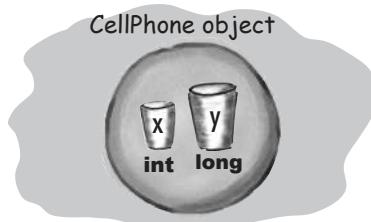
When the new object has instance variables that are object references rather than primitives, the real question is: does the object need space for all of the objects it holds references to? The answer is, *not exactly*. No matter what, Java has to make space for the instance variable *values*. But remember that a reference variable value is not the whole *object*, but merely a *remote control* to the object. So if `CellPhone` has an instance variable declared as the non-primitive type `Antenna`, Java makes space within the `CellPhone` object only for the `Antenna`'s *remote control* (i.e., reference variable) but not the `Antenna` *object*.

Well, then, when does the `Antenna object` get space on the Heap? First we have to find out *when* the `Antenna` object itself is created. That depends on the instance variable declaration. If the instance variable is declared but no object is assigned to it, then only the space for the reference variable (the remote control) is created.

```
private Antenna ant;
```

No actual `Antenna` object is made on the heap unless or until the reference variable is assigned a new `Antenna` object.

```
private Antenna ant = new Antenna();
```

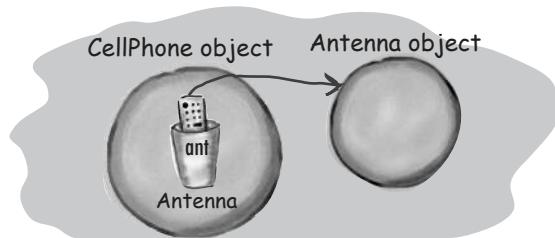


Object with two primitive instance variables.  
Space for the variables lives in the object.



Object with one non-primitive instance variable—a reference to an `Antenna` object, but no actual `Antenna` object. This is what you get if you declare the variable but don't initialize it with an actual `Antenna` object.

```
public class CellPhone {  
    private Antenna ant;  
}
```



Object with one non-primitive instance variable, and the `Antenna` variable is assigned a new `Antenna` object.

```
public class CellPhone {  
    private Antenna ant = new Antenna();  
}
```

## The miracle of object creation

Now that you know where variables and objects live, we can dive into the mysterious world of object creation. Remember the three steps of object declaration and assignment: declare a reference variable, create an object, and assign the object to the reference.

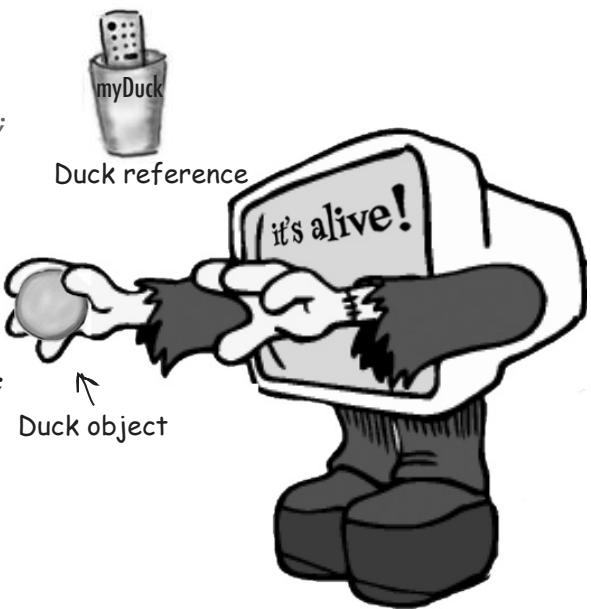
But until now, step two—where a miracle occurs and the new object is “born”—has remained a Big Mystery. Prepare to learn the facts of object life. *Hope you’re not squeamish.*

### Let's review the 3 steps of object declaration, creation and assignment:

Make a new reference variable of a class or interface type.

- 1 Declare a reference variable

Duck myDuck = new Duck();



A miracle occurs here.

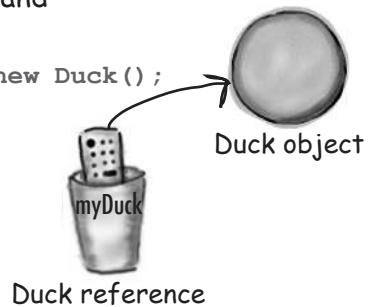
- 2 Create an object

Duck myDuck = new Duck();

Assign the new object to the reference.

- 3 Link the object and the reference

Duck myDuck = new Duck();



## Are we calling a method named Duck()?

Because it sure *looks* like it.

```
Duck myDuck = new Duck();
```

It looks like we're calling a method named Duck(), because of the parentheses.

No.

We're calling the Duck **constructor**.

A constructor *does* look and feel a lot like a method, but it's not a method. It's got the code that runs when you say **new**. In other words, *the code that runs when you instantiate an object*.

The only way to invoke a constructor is with the keyword **new** followed by the class name. The JVM finds that class and invokes the constructor in that class. (OK, technically this isn't the *only* way to invoke a constructor. But it's the only way to do it from *outside* a constructor. You *can* call a constructor from within another constructor, with restrictions, but we'll get into all that later in the chapter.)

**But where is the constructor?**

If we didn't write it, who did?

**A constructor** has the code that runs when you instantiate an object. In other words, the code that runs when you say **new** on a class type.

Every class you write has a constructor, even if you don't write it yourself.

You can write a constructor for your class (we're about to do that), but if you don't, **the compiler writes one for you!**

Here's what the compiler's default constructor looks like:

```
public Duck() {  
}
```

**Notice something missing? How is this different from a method?**

Where's the return type? If this were a method, you'd need a return type between "Public" and "Duck()".

```
public Duck() {  
    // constructor code goes here  
}
```

Its name is the same as the class name. That's mandatory.

## Construct a Duck

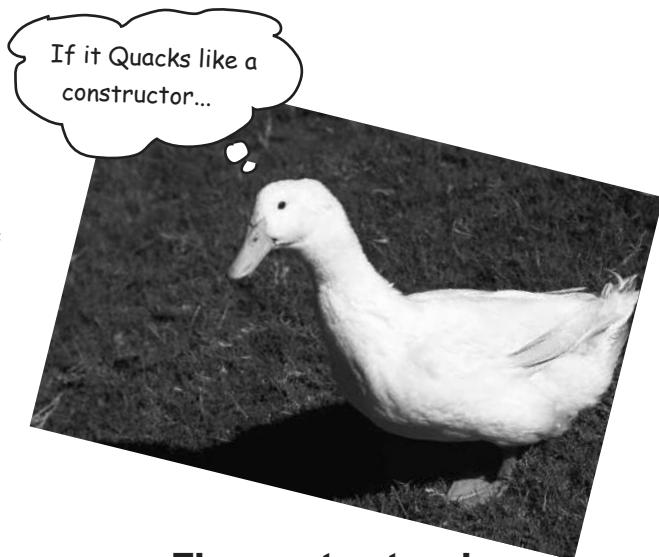
The key feature of a constructor is that it runs *before* the object can be assigned to a reference. That means you get a chance to step in and do things to get the object ready for use. In other words, before anyone can use the remote control for an object, the object has a chance to help construct itself. In our Duck constructor, we're not doing anything useful, just demonstrating the sequence of events.

```
public class Duck {  
  
    public Duck() {  
        System.out.println("Quack");  
    }  
}
```

Constructor code.

```
public class UseADuck {  
  
    public static void main (String[] args) {  
        Duck d = new Duck();  
    }  
}
```

This calls the Duck constructor.



**The constructor gives you a chance to step into the middle of `new`.**

```
File Edit Window Help Quack  
% java UseADuck  
Quack
```



### Sharpen your pencil

A constructor lets you jump into the middle of the object creation step—into the middle of `new`. Can you imagine conditions where that would be useful? Which of the actions on the right might be useful in a Car class constructor, if the Car is part of a Racing Game? Check off the ones that you came up with a scenario for.

- Increment a counter to track how many objects of this class type have been made.
- Assign runtime-specific state (data about what's happening NOW).
- Assign values to the object's important instance variables.
- Get and save a reference to the object that's *creating* the new object.
- Add the object to an `ArrayList`.
- Create HAS-A objects.
- \_\_\_\_\_ (your idea here)

→ Yours to solve.

# Initializing the state of a new Duck

Most people use constructors to initialize the state of an object. In other words, to make and assign values to the object's instance variables.

```
public Duck() {
    size = 34;
}
```

That's all well and good when the Duck class *developer* knows how big the Duck object should be. But what if we want the programmer who is *using* Duck to decide how big a particular Duck should be?

Imagine the Duck has a size instance variable, and you want the programmer using your Duck class to set the size of the new Duck. How could you do it?

Well, you could add a setSize() setter method to the class. But that leaves the Duck temporarily without a size\* and forces the Duck user to write *two* statements—one to create the Duck, and one to call the setSize() method. The code below uses a setter method to set the initial size of the new Duck.

```
public class Duck {
    int size;           ← Instance variable

    public Duck() {
        System.out.println("Quack");   ← Constructor
    }

    public void setSize(int newSize) {
        size = newSize;             ← Setter method
    }
}
```

---

```
public class UseADuck {
```

```
    public static void main(String[] args) {
        Duck d = new Duck();

        d.setSize(42);           ←
    }
}
```

*There's a bad thing here. The Duck is alive at this point in the code, but without a size!\* And then you're relying on the Duck user to KNOW that Duck creation is a two-part process: one to call the constructor and one to call the setter.*

\*Instance variables do have a default value. 0 or 0.0 for numeric primitives, false for booleans, and null for references.

there are no  
Dumb Questions

**Q:** Why do you need to write a constructor if the compiler writes one for you?

**A:** If you need code to help initialize your object and get it ready for use, you'll have to write your own constructor. You might, for example, be dependent on input from the user before you can finish making the object ready. There's another reason you might have to write a constructor, even if you don't need any constructor code yourself. It has to do with your superclass constructor, and we'll talk about that soon.

**Q:** How can you tell a constructor from a method? Can you also have a method that's the same name as the class?

**A:** Java lets you declare a method with the same name as your class. That doesn't make it a constructor, though. The thing that separates a method from a constructor is the return type. Methods *must* have a return type, but constructors *cannot* have a return type.

`public Duck() {}` Constructor

`public void Duck() {}` Method  
  ← Return type

The compiler will allow these methods but **don't do this**. It's against normal naming conventions (methods start with a lower-case letter) but more importantly it's super confusing.

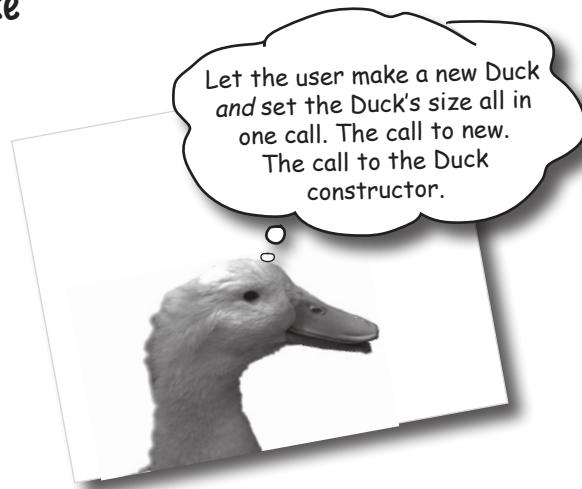
**Q:** Are constructors inherited? If you don't provide a constructor but your superclass does, do you get the superclass constructor instead of the default?

**A:** Nope. Constructors are not inherited. We'll look at that in just a few pages.

## Using the constructor to initialize important Duck state\*

If an object shouldn't be used until one or more parts of its state (instance variables) have been initialized, don't let anyone get hold of a Duck object until you're finished initializing! It's usually way too risky to let someone make—and get a reference to—a new Duck object that isn't quite ready for use until that someone turns around and calls the `setSize()` method. How will the Duck user even *know* that he's required to call the setter method after making the new Duck?

The best place to put initialization code is in the constructor. And all you need to do is make a constructor with arguments.



```
public class Duck {
    int size;

    public Duck(int duckSize) {
        System.out.println("Quack");

        size = duckSize;
        System.out.println("size is " + size);
    }
}
```

Add an int parameter to the Duck constructor.

Use the argument value to set the size instance variable. We could have called the `setSize` method instead.

```
public class UseADuck {

    public static void main (String[] args) {
        Duck d = new Duck(42);
    }
}
```

This time there's only one statement. We make the new Duck and set its size in one statement.

Pass a value to the constructor.

File Edit Window Help Honk
% java UseADuck
Quack
size is 42

\*Not to imply that not all Duck state is not unimportant.

# Make it easy to make a Duck

## Be sure you have a no-arg constructor

What happens if the Duck constructor takes an argument? Think about it. On the previous page, there's only *one* Duck constructor—and it takes an int argument for the *size* of the Duck. That might not be a big problem, but it does make it harder for a programmer to create a new Duck object, especially if the programmer doesn't *know* what the size of a Duck should be. Wouldn't it be helpful to have a default size for a Duck so that if the user doesn't know an appropriate size, they can still make a Duck that works?

**Imagine that you want Duck users to have TWO options for making a Duck—one where they supply the Duck size (as the constructor argument) and one where they don't specify a size and thus get your default Duck size.**

You can't do this cleanly with just a single constructor. Remember, if a method (or constructor—same rules) has a parameter, you *must* pass an appropriate argument when you invoke that method or constructor. You can't just say, "If someone doesn't pass anything to the constructor, then use the default size" because they won't even be able to compile without sending an int argument to the constructor call. You *could* do something clunky like this:

```
public class Duck {
    int size;

    public Duck(int newSize) {
        if (newSize == 0) { ←
            size = 27;
        } else {
            size = newSize;
        }
    }
}
```

If the parameter value is zero, give the new Duck a default size; otherwise, use the parameter value for the size. NOT a very good solution.

But that means the programmer making a new Duck object has to *know* that passing a "0" is the protocol for getting the default Duck size. Pretty ugly. What if the other programmer doesn't know that? Or what if they really *do* want a zero-sized Duck? (Assuming a zero-sized Duck is allowed. If you don't want zero-sized Duck objects, put validation code in the constructor to prevent it.) The point is, it might not always be possible to distinguish between a genuine "I want zero for the size" constructor argument and a "I'm sending zero so you'll give me the default size, whatever that is" constructor argument.

**You really want TWO ways to make a new Duck:**

```
public class Duck2 {
    int size;

    public Duck2() {
        // supply default size
        size = 27;
    }

    public Duck2(int duckSize) {
        // use duckSize parameter
        size = duckSize;
    }
}
```

**To make a Duck when you know the size:**

```
Duck2 d = new Duck2(15);
```

**To make a Duck when you do not know the size:**

```
Duck2 d2 = new Duck2();
```

So this two-options-to-make-a-Duck idea needs two constructors. One that takes an int and one that doesn't. **If you have more than one constructor in a class, it means you have overloaded constructors.**

## Doesn't the compiler always make a no-arg constructor for you? No!

You might think that if you write *only* a constructor with arguments, the compiler will see that you don't have a no-arg constructor and stick one in for you. But that's not how it works. The compiler gets involved with constructor-making *only if you don't say anything at all about constructors.*

**If you write a constructor that takes arguments and you still want a no-arg constructor, you'll have to build the no-arg constructor yourself!**

As soon as you provide a constructor, ANY kind of constructor, the compiler backs off and says, "OK fair enough, looks like you're in charge of constructors now."

**If you have more than one constructor in a class, the constructors MUST have different argument lists.**

The argument list includes the order and types of the arguments. As long as they're different, you can have more than one constructor. You can do this with methods as well, but we'll get to that in another chapter.



## Overloaded constructors means you have more than one constructor in your class.

### To compile, each constructor must have a different argument list!

The class below is legal because all five constructors have different argument lists. If you had two constructors that took only an int, for example, the class wouldn't compile. What you name the parameter variable doesn't count. It's the variable *type* (int, Dog, etc.) and *order* that matters. You *can* have two constructors that have identical types, **as long as the order is different**. A constructor that takes a String followed by an int is *not* the same as one that takes an int followed by a String.

Five different constructors  
means five different ways to  
make a new mushroom.



These two have the same args, but in a different order, so it's OK\*

```
public class Mushroom {  
    public Mushroom(int size) {}  
    public Mushroom() {}  
    public Mushroom(boolean isMagic) {}  
    public Mushroom(boolean isMagic, int size) {}  
    public Mushroom(int size, boolean isMagic) {}  
}  
*If the arguments were the same type, how would the compiler know they were two different things?
```

When you know the size, but you don't know if it's magic

When you don't know anything

When you know if it's magic or not, but don't know the size

When you know whether or not it's magic, AND you know the size as well

### BULLET POINTS

- Instance variables live within the object they belong to, on the Heap.
- If the instance variable is a reference to an object, both the reference and the object it refers to are on the Heap.
- A constructor is the code that runs when you say `new` on a class type.
- A constructor must have the same name as the class, and must *not* have a return type.
- You can use a constructor to initialize the state (i.e., the instance variables) of the object being constructed.
- If you don't put a constructor in your class, the compiler will put in a default constructor.
- The default constructor is always a no-arg constructor.
- If you put a constructor—any constructor—in your class, the compiler will not build the default constructor.
- If you want a no-arg constructor and you've already put in a constructor with arguments, you'll have to build the no-arg constructor yourself.
- Always provide a no-arg constructor if you can, to make it easy for programmers to make a working object. Supply default values.
- Overloaded constructors means you have more than one constructor in your class.
- Overloaded constructors must have different argument lists.
- You cannot have two constructors with the same argument lists. An argument list includes the order and type of arguments.
- Instance variables are assigned a default value, even when you don't explicitly assign one. The default values are 0/0/false for primitives, and null for references.



Yours to solve.

Match the `new Duck()` call with the constructor that runs when that Duck is instantiated. We did the easy one to get you started.

```
public class TestDuck {
    public static void main(String[] args) {
        int weight = 8;
        float density = 2.3F;
        String name = "Donald";
        long[] feathers = {1, 2, 3, 4, 5, 6};
        boolean canFly = true;
        int airspeed = 22;

        Duck[] d = new Duck[7];
        d[0] = new Duck();
        d[1] = new Duck(density, weight);
        d[2] = new Duck(name, feathers);
        d[3] = new Duck(canFly);
        d[4] = new Duck(3.3F, airspeed);
        d[5] = new Duck(false);
        d[6] = new Duck(airspeed, density);
    }
}
```

## there are no Dumb Questions

**Q:** Earlier you said that it's good to have a no-argument constructor so that if people call the no-arg constructor, we can supply default values for the "missing" arguments. But aren't there times when it's impossible to come up with defaults? Are there times when you should not have a no-arg constructor in your class?

**A:** You're right. There are times when a no-arg constructor doesn't make sense. You'll see this in the Java API—some classes don't have a no-arg constructor. The Color class, for example, represents a...color. Color objects are used to, for example, set or change the color of a screen font or GUI button. When you make a Color instance, that instance is of a particular color (you know, Death-by-Chocolate Brown, Blue-Screen-of-Death Blue, Scandalous Red, etc.).

```
class Duck {
    private int kilos = 6;
    private float floatability = 2.1F;
    private String name = "Generic";
    private long[] feathers = {1, 2, 3,
                               4, 5, 6, 7};
    private boolean canFly = true;
    private int maxSpeed = 25;

    public Duck() {
        System.out.println("type 1 duck");
    }

    public Duck(boolean fly) {
        canFly = fly;
        System.out.println("type 2 duck");
    }

    public Duck(String n, long[] f) {
        name = n;
        feathers = f;
        System.out.println("type 3 duck");
    }

    public Duck(int w, float f) {
        kilos = w;
        floatability = f;
        System.out.println("type 4 duck");
    }

    public Duck(float density, int max) {
        floatability = density;
        maxSpeed = max;
        System.out.println("type 5 duck");
    }
}
```

If you make a Color object, you must specify the color in some way.

`Color c = new Color(3,45,200);`  
(We're using three ints for RGB values here. We'll get into using Color later, in Chapter 15, *Work on Your Swing*.) Otherwise, what would you get? The Java API programmers could have decided that if you call a no-arg Color constructor you'll get a lovely shade of mauve. But good taste prevailed. If you try to make a Color without supplying an argument:

```
Color c = new Color();
```

the compiler freaks out because it can't find a matching no-arg constructor in the Color class.

```
File Edit Window Help StopBeingStupid
cannot resolve symbol
:constructor Color()
location: class java.awt.
Color
Color c = new Color();
^
1 error
```

## Nanoreview: four things to remember about constructors

- ① A constructor is the code that runs when somebody says `new` on a class type:

```
Duck d = new Duck();
```

- ② A constructor must have the same name as the class, and `no` return type:

```
public Duck(int size) { }
```

- ③ If you don't put a constructor in your class, the compiler puts in a default constructor. The default constructor is always a no-arg constructor.

```
public Duck() { }
```

- ④ You can have more than one constructor in your class, as long as the argument lists are different. Having more than one constructor in a class means you have overloaded constructors.

```
public Duck() { }

public Duck(int size) { }

public Duck(String name) { }

public Duck(String name, int size) { }
```



### What about superclasses?

**When you make a Dog,  
should the Canine  
constructor run too?**

**If the superclass is abstract,  
should it even *have* a  
constructor?**

We'll look at this on the next few pages, so stop now and think about the implications of constructors and superclasses.\*

there are no  
**Dumb Questions**

**Q:** Do constructors have to be `public`?

**A:** No. Constructors can be `public`, `protected`, `private`, or `default` (which means no access modifier at all). We'll look more at `default` access in appendix B.

**Q:** How could a `private` constructor ever be useful? Nobody could ever call it, so nobody could ever make a new object!

**A:** Not exactly right. Marking something `private` doesn't mean *nobody* can access it; it just means that *nobody outside the class* can access it. Bet you're thinking Catch 22. Only code from the *same* class as the class-with-private-constructor can make a new object from that class, but without first making an object, how do you ever get to run code from that class in the first place? How do you ever get to anything in that class? *Patience grasshopper*. We'll get there in the next chapter.

\*Doing all the Brain Power exercises has been shown to produce a 42% increase in neuron size. And you know what they say, "Big neurons..."

space for an object's superclass parts

## Wait a minute...we never DID talk about superclasses and inheritance and how that all fits in with constructors

Here's where it gets fun. Remember in the previous chapter we looked at the Snowboard object wrapping around an inner core representing the Object portion of the Snowboard class? The Big Point there was that every object holds not just its *own* declared instance variables, but also *everything from its superclasses* (which, at a minimum, means class Object, since *every* class extends Object).

So when an object is created (because somebody said **new**; there is **no other way** to create an object other than someone, somewhere saying **new** on the class type), the object gets space for *all* the instance variables, from all the way up the inheritance tree. Think about it for a moment... a superclass might have setter methods encapsulating a private variable. But that variable has to live *somewhere*. When an object is created, it's almost as though *multiple* objects materialize—the object being new'd and one object per each superclass. Conceptually, though, it's much better to think of it like the picture below, where the object being created has *layers* of itself representing each superclass.

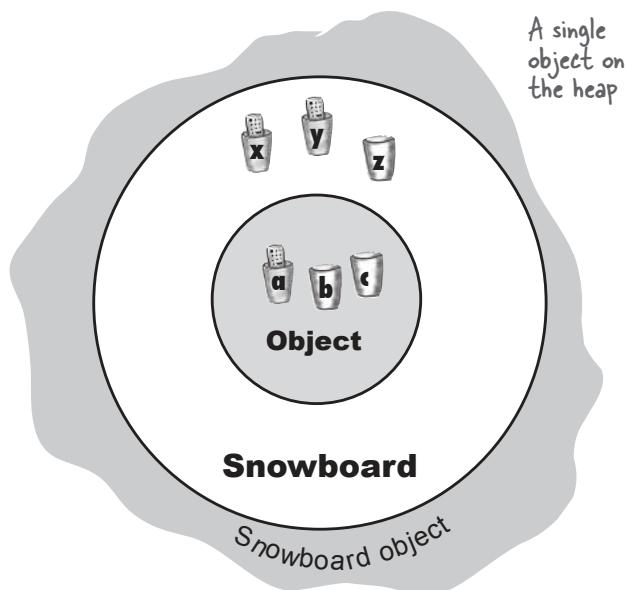
Object
Foo a; int b; int c;  equals() getClass() hashCode() toString()

↑

Snowboard
Foo x Foo y int z  turn() shred() getAir() loseControl()

Object has instance variables encapsulated by access methods. Those instance variables are created when any subclass is instantiated. (These aren't the *REAL* Object variables, but we don't care what they are since they're encapsulated.)

Snowboard also has instance variables of its own, so to make a Snowboard object we need space for the instance variables of both classes.



There is only ONE object on the heap here. A Snowboard object. But it contains both the Snowboard parts of itself and the Object parts of itself. All instance variables from both classes have to be here.

# The role of superclass constructors in an object's life

**All the constructors in an object's inheritance tree must run when you make a new object.**

Let that sink in.

That means every superclass has a constructor (because every class has a constructor), and each constructor up the hierarchy runs at the time an object of a subclass is created.

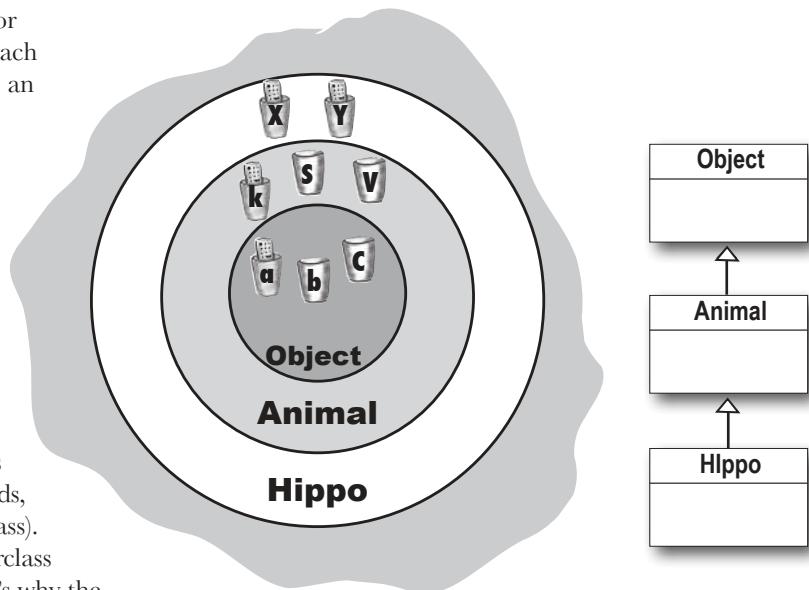
Saying **new** is a Big Deal. It starts the whole constructor chain reaction. And yes, even abstract classes have constructors. Although you can never say new on an abstract class, an abstract class is still a superclass, so its constructor runs when someone makes an instance of a concrete subclass.

The superclass constructors run to build out the superclass parts of the object.

Remember, a subclass might inherit methods that depend on superclass state (in other words, the value of instance variables in the superclass). For an object to be fully formed, all the superclass parts of itself must be fully formed, and that's why the superclass constructor *must* run. All instance variables from every class in the inheritance tree have to be declared and initialized. Even if Animal has instance variables that Hippo doesn't inherit (if the variables are private, for example), the Hippo still depends on the Animal methods that *use* those variables.

When a constructor runs, it immediately calls its superclass constructor, all the way up the chain until you get to the class Object constructor.

On the next few pages, you'll learn how superclass constructors are called, and how you can call them yourself. You'll also learn what to do if your superclass constructor has arguments!



A single Hippo object on the heap

**A new Hippo object also IS-A Animal and IS-A Object. If you want to make a Hippo, you must also make the Animal and Object parts of the Hippo.**

**This all happens in a process called Constructor Chaining.**

# Making a Hippo means making the Animal and Object parts too...

```
public class Animal {
    public Animal() {
        System.out.println("Making an Animal");
    }
}
```

```
public class Hippo extends Animal {
    public Hippo() {
        System.out.println("Making a Hippo");
    }
}
```

```
public class TestHippo {
    public static void main(String[] args) {
        System.out.println("Starting...");
        Hippo h = new Hippo();
    }
}
```

Given the class hierarchy in the code above, we can step through the process of creating a new Hippo object.

## Sharpen your pencil

What's the real output? Given the code on the left, what prints out when you run TestHippo? A or B?  
(The answer is at the bottom of the page.)

A

```
File Edit Window Help Swear
% java TestHippo
Starting...
Making an Animal
Making a Hippo
```

B

```
File Edit Window Help Swear
% java TestHippo
Starting...
Making a Hippo
Making an Animal
```

① Code from another class calls `new Hippo()`, and the `Hippo()` constructor goes into a stack frame at the top of the stack.



② `Hippo()` invokes the superclass constructor, which pushes the `Animal()` constructor onto the top of the stack.



③ `Animal()` invokes the superclass constructor, which pushes the `Object()` constructor onto the top of the stack, since `Object` is the superclass of `Animal`.



④ `Object()` completes, and its stack frame is popped off the stack. Execution goes back to the `Animal()` constructor and picks up at the line following `Animal`'s call to its superclass constructor.



The first one, A. The `Hippo()` constructor is invoked first, but it's the `Animal` constructor that finishes first.

# How do you invoke a superclass constructor?

You might think that somewhere in, say, a Duck constructor, if Duck extends Animal you'd call `Animal()`. But that's not how it works:

```
public class Duck extends Animal {
    int size;

    public Duck(int newSize) {
        BAD! → Animal(); ← NO! This is not legal!
        size = newSize;
    }
}
```

The only way to call a superclass constructor is by calling `super()`. That's right—`super()` calls the **superclass constructor**.

What are the odds?

```
public class Duck extends Animal {
    int size;

    public Duck(int newSize) {
        super(); ← You just call super()
        size = newSize;
    }
}
```

A call to `super()` in your constructor puts the superclass constructor on the top of the Stack. And what do you think that superclass constructor does? *Calls its superclass constructor.* And so it goes until the `Object` constructor is on the top of the Stack. Once `Object()` finishes, it's popped off the Stack, and the next thing down the Stack (the subclass constructor that called `Object()`) is now on top. *That* constructor finishes and so it goes until the original constructor is on the top of the Stack, where *it* can now finish.

## And how is it that we've gotten away without calling `super()` before?

You probably figured that out.

**Our good friend the compiler puts in a call to `super()` if you don't.**

So the compiler gets involved in constructor-making in two ways:

### ① If you don't provide a constructor

The compiler puts one in that looks like:

```
public ClassName() {
    super();
}
```

### ② If you do provide a constructor but you do not put in the call to `super()`

The compiler will put a call to `super()` in each of your overloaded constructors.\* The compiler-supplied call looks like:

```
super();
```

It always looks like that. The compiler-inserted call to `super()` is always a no-arg call. If the superclass has overloaded constructors, only the no-arg constructor is called.

\*Unless the constructor calls another overloaded constructor (you'll see that in a few pages).

# Can the child exist before the parents?

If you think of a superclass as the parent to the subclass child, you can figure out which has to exist first. ***The superclass parts of an object have to be fully formed (completely built) before the subclass parts can be constructed.***

Remember, the subclass object might depend on things it inherits from the superclass, so it's important that those inherited things be finished. No way around it. The superclass constructor must finish before its subclass constructor.

Look at the Stack series on page 254 again, and you can see that while the Hippo constructor is the *first* to be invoked (it's the first thing on the Stack), it's the *last* one to complete! Each subclass constructor immediately invokes its own superclass constructor, until the Object constructor is on the top of the Stack. Then Object's constructor completes, and we bounce back down the Stack to Animal's constructor. Only after Animal's constructor completes do we finally come back down to finish the rest of the Hippo constructor. For that reason:

**The call to super() must be the first statement in each constructor!\***



## Possible constructors for class Boop

```
 public Boop() {
    super();
}
```

These are OK because the programmer explicitly coded the call to super() as the first statement.

```
 public Boop(int i) {
    super();
    size = i;
}
```

```
 public Boop() {
```

}

```
 public Boop(int i) {
```

size = i;

}

```
 public Boop(int i) {
```

size = i;

super();

}

These are OK because the compiler will put a call to super() in as the first statement.

BAD!! This won't compile! You can't explicitly put the call to super() below anything else.

\*There's an exception to this rule; you'll learn it on page 258.

## Superclass constructors with arguments

What if the superclass constructor has arguments? Can you pass something in to the super() call? Of course. If you couldn't, you'd never be able to extend a class that didn't have a no-arg constructor. Imagine this scenario: all animals have a name. There's a getName() method in class Animal that returns the value of the name instance variable. The instance variable is marked private, but the subclass (in this case, Hippo) inherits the getName() method. So here's the problem: Hippo has a getName() method (through inheritance) but does not have the name instance variable. Hippo has to depend on the Animal part of himself to keep the name instance variable, and return it when someone calls getName() on a Hippo object. But...how does the Animal part get the name? The only reference Hippo has to the Animal part of himself is through super(), so that's the place where Hippo sends the Hippo's name up to the Animal part of himself, so that the Animal part can store it in the private name instance variable.

```
public abstract class Animal {
    private String name; ← All animals (including
                           subclasses) have a name.

    public String getName() ← A getter method that
                           Hippo inherits.
        return name;
    }

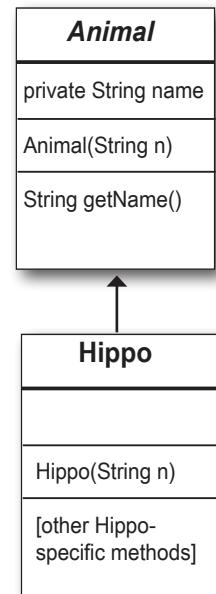
    public Animal(String theName) { The constructor that
        name = theName;           takes the name and assigns
    }                           it the name instance
                                variable.
}
```

---

```
public class Hippo extends Animal {
    public Hippo(String name) { Hippo constructor takes a name.
        super(name);
    }
} ← It sends the name up the Stack
      to the Animal constructor.
```

---

```
public class MakeHippo {
    public static void main(String[] args) {
        Hippo h = new Hippo("Buffy"); ← Make a Hippo, passing the
                                         name "Buffy" to the Hippo
                                         constructor. Then call the
                                         Hippo's inherited getName().
    }
}
```



```

File Edit Window Help Hide
% java MakeHippo
Buffy
  
```

## Invoking one overloaded constructor from another

What if you have overloaded constructors that, with the exception of handling different argument types, all do the same thing? You know that you don't want *duplicate* code sitting in each of the constructors (pain to maintain, etc.), so you'd like to put the bulk of the constructor code (including the call to `super()`) in only *one* of the overloaded constructors. You want whichever constructor is first invoked to call The Real Constructor and let The Real Constructor finish the job of construction. It's simple: just say `this()`. Or `this(aString)`. Or `this(27, x)`. In other words, just imagine that the keyword `this` is a reference to the **current object**.

You can say `this()` only within a constructor, and it must be the first statement in the constructor!

But that's a problem, isn't it? Earlier we said that `super()` must be the first statement in the constructor. Well, that means you get a choice.

### **Every constructor can have a call to super() or this(), but never both!**

You'll need to choose which to call based on which values you have, which ones you need to set, and which constructors are provided in this class or the superclass.

```
import java.awt.Color;
```

```
class Mini extends Car {
    private Color color;

    public Mini() {
        this(Color.RED);
    }

    public Mini(Color c) {
        super("Mini");
        color = c;
        // more initialization
    }

    public Mini(int size) {
        this(Color.RED);
        super(size);
    }
}
```

The no-arg constructor supplies a default Color and calls the overloaded Real Constructor (the one that calls `super()`).

This is The Real Constructor that does The Real Work of initializing the object (including the call to `super()`).

Won't work!! Can't have `super()` and `this()` in the same constructor, because they each must be the first statement!

Use `this()` to call a constructor from another overloaded constructor in the same class.

The call to `this()` can be used only in a constructor, and must be the first statement in a constructor.

A constructor can have a call to `super()` OR `this()`, but never both!

```
File Edit Window Help Drive
javac Mini.java
Mini.java:16: call to super must
be first statement in constructor
        super();
               ^

```



→ Yours to solve.

Some of the constructors in the SonOfBoo class will not compile. See if you can recognize which constructors are not legal. Match the compiler errors with the SonOfBoo constructors that caused them, by drawing a line from the compiler error to the "bad" constructor.

```
public class Boo {
    public Boo(int i) { }
    public Boo(String s) { }
    public Boo(String s, int i) { }
}
```

```
class SonOfBoo extends Boo {
    public SonOfBoo() {
        super("boo");
    }

    public SonOfBoo(int i) {
        super("Fred");
    }

    public SonOfBoo(String s) {
        super(42);
    }

    public SonOfBoo(int i, String s) {
    }

    public SonOfBoo(String a, String b, String c) {
        super(a, b);
    }

    public SonOfBoo(int i, int j) {
        super("man", j);
    }

    public SonOfBoo(int i, int x, int y) {
        super(i, "star");
    }
}
```



```
File Edit Window Help
%javac SonOfBoo.java
cannot resolve symbol
symbol : constructor Boo
(java.lang.String,java.lang.String)
```

```
File Edit Window Help Yadayadaya
%javac SonOfBoo.java
cannot resolve symbol
symbol : constructor Boo
(int,java.lang.String)
```

```
File Edit Window Help ImNotListening
%javac SonOfBoo.java
cannot resolve symbol
symbol:constructor Boo()
```

## Now we know how an object is born, but how long does an object live?

An *object's* life depends entirely on the life of references referring to it. If the reference is considered “alive,” the object is still alive on the Heap. If the reference dies (and we'll look at what that means in just a moment), the object will die.

### So if an object's life depends on the reference variable's life, how long does a variable live?

That depends on whether the variable is a *local* variable or an *instance* variable. The code below shows the life of a local variable. In the example, the variable is a primitive, but variable lifetime is the same whether it's a primitive or reference variable.

```
public class TestLifeOne {
    public void read() {
        int s = 42;           ← "s" is scoped to the
        sleep();             method, so it can't be used
    }
}

public void sleep() {
    s = 7;
}                                ↑ BAD!! Not legal to
                                use "s" here!
                                sleep() can't see the "s" variable. Since
                                it's not in sleep()'s own Stack frame,
                                sleep() doesn't know anything about it.
                                read()   s
                                ↑
The variable "s" is alive, but in scope only within the
read() method. When sleep() completes and read() is
on top of the Stack and running again, read() can
still see "s." When read() completes and is popped off
the Stack, "s" is dead. Pushing up digital daisies.
```

### ① A local variable lives only within the method that declared the variable.

```
public void read() {
    int s = 42;
    // 's' can be used only
    // within this method.
    // When this method ends,
    // 's' disappears completely.
}
```

Variable “s” can be used *only* within the *read()* method. In other words, **the variable is in scope only within its own method**. No other code in the class (or any other class) can see “s.”

### ② An instance variable lives as long as the object does. If the object is still alive, so are its instance variables.

```
public class Life {
    int size;

    public void setSize(int s) {
        size = s;
        // 's' disappears at the
        // end of this method,
        // but 'size' can be used
        // anywhere in the class
    }
}
```

Variable ‘s’ (this time a method parameter) is in scope only within the *setSize()* method. But instance variable *size* is scoped to the life of the *object* as opposed to the life of the *method*.

## The difference between **life** and **scope** for local variables:

### Life

A local variable is *alive* as long as its Stack frame is on the Stack. In other words, *until the method completes*.

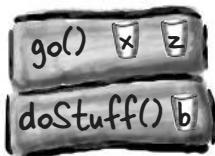
### Scope

A local variable is in *scope* only within the method in which the variable was declared. When its own method calls another, the variable is alive, but not in scope until its method resumes. **You can use a variable only when it is in scope.**

Let's walk through what happens on the stack when something calls the doStuff() method.



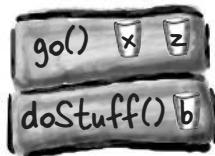
- 1 doStuff() goes on the Stack. Variable "b" is alive and in scope.



- 2 go() plops on top of the Stack. "x" and "z" are alive and in scope, and "b" is alive but *not* in scope.



- 3 crazy() is pushed onto the Stack, with "c" now alive and in scope. The other three variables are alive but out of scope.



- 4 crazy() completes and is popped off the Stack, so 'c' is out of scope and dead. When go() resumes where it left off, "x" and "x" are both alive and back in scope. Variable "b" is still alive but out of scope (until go() completes).

While a local variable is alive, its state persists. As long as method doStuff() is on the Stack, for example, the "b" variable keeps its value. But the "b" variable can be used only while doStuff()'s Stack frame is at the top. In other words, you can use a local variable *only* while that local variable's method is actually running (as opposed to waiting for higher Stack frames to complete).

```

1 public void doStuff() {
    boolean b = true;
    go(4);
}

2 public void go(int x) {
    int z = x + 24;
    crazy();
    // imagine more code here
}

4 public void crazy() {
    char c = 'a';
}

```

## What about reference variables?

The rules are the same for primitives and references. A reference variable can be used only when it's in scope, which means you can't use an object's remote control unless you've got a reference variable that's in scope. The *real* question is:

### "How does *variable* life affect *object* life?"

An object is alive as long as there are live references to it. If a reference variable goes out of scope but is still alive, the object it *refers* to is still alive on the Heap. And then you have to ask...“What happens when the Stack frame holding the reference gets popped off the Stack at the end of the method?”

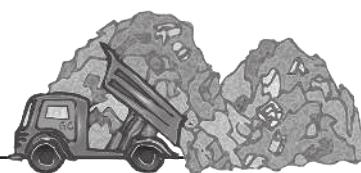
If that was the *only* live reference to the object, the object is now abandoned on the Heap. The reference variable disintegrated with the Stack frame, so the abandoned object is now, *officially*, toast. The trick is to know the point at which an object becomes **eligible for garbage collection**.

Once an object is eligible for garbage collection (GC), you don't have to worry about reclaiming the memory that object was using. If your program gets low on memory, GC will destroy some or all of the eligible objects, to keep you from running out of RAM. You can still run out of memory, but *not* before all eligible objects have been hauled off to the dump. Your job is to make sure that you abandon objects (i.e., make them eligible for GC) when you're done with them, so that the garbage collector has something to reclaim. If you hang on to objects, GC can't help you, and you run the risk of your program dying a painful out-of-memory death.

**An object's life has no value, no meaning, no point, unless somebody has a reference to it.**

**If you can't get to it, you can't ask it to do anything and it's just a big fat waste of bits.**

**But if an object is unreachable, the Garbage Collector will figure that out. Sooner or later, that object's goin' down.**



**An object becomes eligible for GC when its last live reference disappears.**

### Three ways to get rid of an object's reference:

- ① The reference goes out of scope, permanently

```
void go () {  
    Life z = new Life ();  
}
```

reference 'z' dies at  
end of method.

- ② The reference is assigned another object

```
Life z = new Life ();  
z = new Life ();
```

the first object is abandoned  
when z is 'reprogrammed' to  
a new object.

- ③ The reference is explicitly set to null

```
Life z = new Life ();  
z = null;
```

the first object is abandoned  
when z is 'deprogrammed.'

## Object-killer #1

**Reference goes out of scope, permanently.**



```
public class StackRef {
    public void foof() {
        barf();
    }

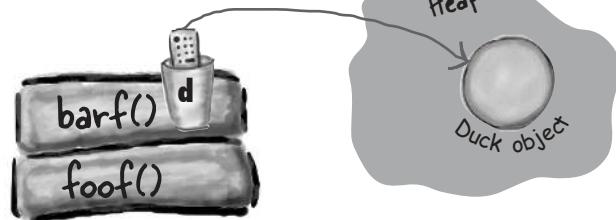
    public void barf() {
        Duck d = new Duck();
    }
}
```



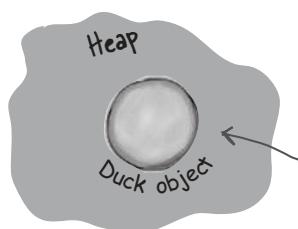
- 1 *foof()* is pushed onto the Stack; no variables are declared.



- 2 *barf()* is pushed onto the Stack, where it declares a reference variable, and creates a new object assigned to that reference. The object is created on the Heap, and the reference is alive and in scope.



- 3 *barf()* completes and pops off the Stack. Its frame disintegrates, so "d" is now dead and gone. Execution returns to *foof()*, but *foof()* can't use "d."



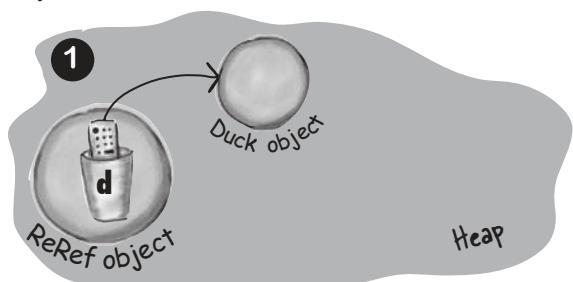
## Object-killer #2

Assign the reference  
to another object



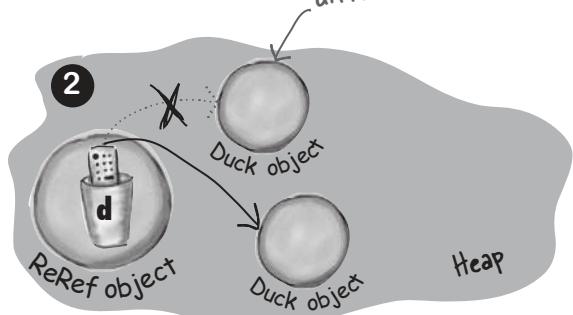
```
public class ReRef {
    Duck d = new Duck();

    public void go() {
        d = new Duck();
    }
}
```



The new Duck goes on the Heap, referenced by "d." Since "d" is an instance variable, the Duck will live as long as the ReRef object that instantiated it is alive. Unless...

When someone calls the go() method, this Duck is abandoned. Its only reference has been reprogrammed for a different Duck.



'd' is assigned a new Duck object, leaving the original (first) Duck object abandoned. That first Duck is now as good as dead.

Dude,  
all you had to do  
was reset the reference.  
Guess they didn't have  
memory management back  
then.



## Object-killer #3

Explicitly set the reference to null



```
public class ReRef {
    Duck d = new Duck();

    public void go() {
        d = null;
    }
}
```

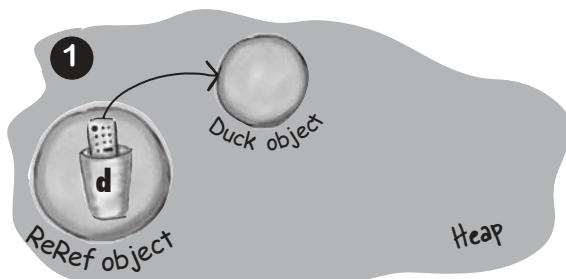
### The meaning of null

When you set a reference to `null`, you're deprogramming the remote control.

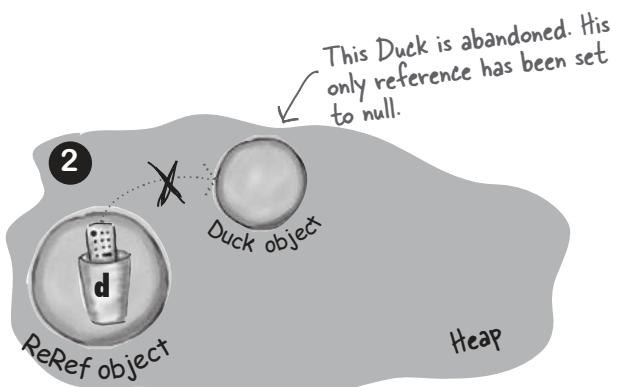
In other words, you've got a remote control, but no TV at the other end. A null reference has bits representing "null" (we don't know or care what those bits are, as long as the JVM knows).

If you have an unprogrammed remote control, in the real world, the buttons don't do anything when you press them. But in Java, you can't press the buttons (i.e., use the dot operator) on a null reference, because the JVM knows (this is a runtime issue, not a compiler error) that you're expecting a bark, but there's no Dog there to do it!

**If you use the dot operator on a null reference, you'll get a `NullPointerException` at runtime.** You'll learn all about exceptions in Chapter 13, *Risky Behavior*.

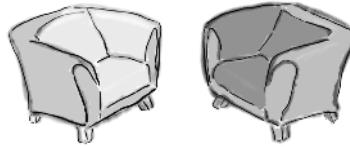


The new Duck goes on the Heap, referenced by "d." Since "d" is an instance variable, the Duck will live as long as the ReRef object that instantiated it is alive. Unless...



'd' is set to null, which is just like having a remote control that isn't programmed to anything. You're not even allowed to use the dot operator on 'd' until it's reprogrammed (assigned an object).

## Fireside Chats



### Instance Variable

I'd like to go first, because I tend to be more important to a program than a local variable. I'm there to support an object, usually throughout the object's entire life. After all, what's an object without state? And what is state? Values kept in *instance variables*.

No, don't get me wrong, I do understand your role in a method; it's just that your life is so short. So temporary. That's why they call you guys "temporary variables."

My apologies. I understand completely.

I never really thought about it like that. What are you doing while the other methods are running and you're waiting for your frame to be the top of the Stack again?

### Tonight's Talk: An instance variable and a local variable discuss life and death (with remarkable civility)

### Local Variable

I appreciate your point of view, and I certainly appreciate the value of object state and all, but I don't want folks to be misled. Local variables are *really* important. To use your phrase, "After all, what's an object without *behavior*?" And what is behavior? Algorithms in methods. And you can bet your bits there'll be some *local variables* in there to make those algorithms work.

Within the local-variable community, the phrase "temporary variable" is considered derogatory. We prefer "local," "stack," "automatic," or "Scope-challenged."

Anyway, it's true that we don't have a long life, and it's not a particularly *good* life either. First, we're shoved into a Stack frame with all the other local variables. And then, if the method we're part of calls another method, another frame is pushed on top of us. And if *that* method calls *another* method...and so on. Sometimes we have to wait forever for all the other methods on top of the Stack to complete so that our method can run again.

Nothing. Nothing at all. It's like being in stasis—that thing they do to people in science-fiction movies when they have to travel long distances. Suspended animation, really. We just sit there on hold. As long as our frame is still there, we're safe and the value we hold is secure, but it's a mixed blessing when our frame gets to run again. On the one hand, we get to be active again. On the other

**Instance Variable**

We saw an educational video about it once. Looks like a pretty brutal ending. I mean, when that method hits its ending curly brace, the frame is literally *blown* off the Stack! Now *that's* gotta hurt.

I live on the Heap, with the objects. Well, not *with* the objects, actually *in* an object. The object whose state I store. I have to admit life can be pretty luxurious on the Heap. A lot of us feel guilty, especially around the holidays.

OK, hypothetically, yes, if I'm an instance variable of the Collar and the Collar gets GC'd, then the Collar's instance variables would indeed be tossed out like so many pizza boxes. But I was told that this almost never happens.

They let us *drink*?

**Local Variable**

hand, the clock starts ticking again on our short lives. The more time our method spends running, the closer we get to the end of the method. We *all* know what happens then.

Tell me about it. In computer science they use the term *popped* as in “the frame was popped off the Stack.” That makes it sound fun, or maybe like an extreme sport. But, well, you saw the footage. So why don't we talk about you? I know what my little Stack frame looks like, but where do *you* live?

But you don't *always* live as long as the object who declared you, right? Say there's a Dog object with a Collar instance variable. Imagine *you're* an instance variable of the Collar object, maybe a reference to a Buckle or something, sitting there all happy inside the Collar object who's all happy inside the Dog object. But...what happens if the Dog wants a new Collar or *nulls* out its Collar instance variable? That makes the Collar object eligible for GC. So...if *you're* an instance variable inside the Collar and the whole Collar is abandoned, what happens to *you*?

And you believed it? That's what they say to keep us motivated and productive. But aren't you forgetting something else? What if you're an instance variable inside an object, and that object is referenced *only* by a *local* variable? If I'm the only reference to the object you're in, when I go, you're coming with me. Like it or not, our fates may be connected. So I say we forget about all this and go get a drink while we still can. Carpe RAM and all that.



## BE the Garbage Collector

Which of the lines of code on the right, if added to the class on the left at point A, would cause exactly one additional object to be eligible for the Garbage Collector? (Assume that point A (`//call more methods`) will execute for a long time, giving the Garbage Collector time to do its stuff.)

```
public class GC {  
    public static GC doStuff() {  
        GC newGC = new GC();  
        doStuff2(newGC);  
        return newGC;  
    }  
  
    public static void main(String[] args) {  
        GC gc1;  
        GC gc2 = new GC();  
        GC gc3 = new GC();  
        GC gc4 = gc3;  
        gc1 = doStuff();  
  
        // call more methods  
    }  
  
    public static void doStuff2(GC copyGC) {  
        GC localGC = copyGC;  
    }  
}
```

→ Answers on page 272.

- 1    `copyGC = 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;`



# Popular Objects

```

class Bees {
    Honey[] beeHoney;
}

class Raccoon {
    Kit rk;
    Honey rh;
}

class Kit {
    Honey honey;
}

class Bear {
    Honey hunny;
}

public class Honey {
    public static void main(String[] args) {
        Honey honeyPot = new Honey();
        Honey[] ha = {honeyPot, honeyPot, honeyPot, honeyPot};
        Bees bees = new Bees();
        bees.beeHoney = ha;
        Bear[] bears = new Bear[5];
        for (int i = 0; i < 5; i++) {
            bears[i] = new Bear();
            bears[i].hunny = honeyPot;
        }
        Kit kit = new Kit();
        kit.honey = honeyPot;
        Raccoon raccoon = new Raccoon();
        raccoon.rh = honeyPot;
        raccoon.rk = kit;
        kit = null;
    } // end of main
}

```

Here's a new Raccoon object!

Here's its reference variable 'raccoon.'

In this code example, several new objects are created. Your challenge is to find the object that is “most popular,” i.e., the one that has the most reference variables referring to it. Then list how many total references there are for that object, and what they are! We’ll start by pointing out one of the new objects and its reference variable.

Good luck!

→ Answers on page 272.



## Five-Minute Mystery



“We’ve run the simulation four times, and the main module’s temperature consistently drifts out of nominal toward cold,” Sarah said, exasperated. “We installed the new temp-bots last week. The readings on the radiator bots, designed to cool the living quarters, seem to be within spec, so we’ve focused our analysis on the heat retention bots, the bots that help to warm the quarters.” Tom sighed, at first it had seemed that nanotechnology was going to really put them ahead of schedule. Now, with only five weeks left until launch, some of the orbiter’s key life support systems were still not passing the simulation gauntlet.

“What ratios are you simulating?” Tom asked.

“Well, if I see where you’re going, we already thought of that,” Sarah replied. “Mission control will not sign off on critical systems if we run them out of spec. We are required to run the v3 radiator bot’s SimUnits in a 2:1 ratio with the v2 radiator’s SimUnits,” Sarah continued. “Overall, the ratio of retention bots to radiator bots is supposed to run 4:3.”

“How’s power consumption, Sarah?” Tom asked. Sarah paused, “Well, that’s another thing, power consumption is running higher than anticipated. We’ve got a team tracking that down too, but because the nanos are wireless, it’s been hard to isolate the power consumption of the radiators from the retention bots.” “Overall power consumption ratios,” Sarah continued, “are designed to run 3:2 with the radiators pulling more power from the wireless grid.”

“OK, Sarah,” Tom said. “Let’s take a look at some of the simulation initiation code. We’ve got to find this problem, and find it quick!”

```
import java.util.ArrayList;

class V2Radiator {
    V2Radiator(ArrayList<SimUnit> list) {
        for (int x = 0; x < 5; x++) {
            list.add(new SimUnit("V2Radiator"));
        }
    }
}

class V3Radiator extends V2Radiator {
    V3Radiator(ArrayList<SimUnit> lglist) {
        super(lglist);
        for (int g = 0; g < 10; g++) {
            lglist.add(new SimUnit("V3Radiator"));
        }
    }
}

class RetentionBot {
    RetentionBot(ArrayList<SimUnit> rlist) {
        rlist.add(new SimUnit("Retention"));
    }
}
```

# Five-Minute Mystery continued...

```

import java.util.ArrayList;

public class TestLifeSupportSim {
    public static void main(String[] args) {
        ArrayList<SimUnit> aList = new ArrayList<SimUnit>();
        V2Radiator v2 = new V2Radiator(aList);
        V3Radiator v3 = new V3Radiator(aList);
        for (int z = 0; z < 20; z++) {
            RetentionBot ret = new RetentionBot(aList);
        }
    }
}

class SimUnit {
    String botType;

    SimUnit(String type) {
        botType = type;
    }

    int powerUse() {
        if ("Retention".equals(botType)) {
            return 2;
        } else {
            return 4;
        }
    }
}

```

Tom gave the code a quick look, and a small smile crept across his lips. “I think I’ve found the problem, Sarah, and I bet I know by what percentage your power usage readings are off too!”

*What did Tom suspect? How could he guess the power readings errors, and what few lines of code could you add to help debug this program?*

—————> Answers on page 273.



## Exercise Solutions

## Be the Garbage Collector

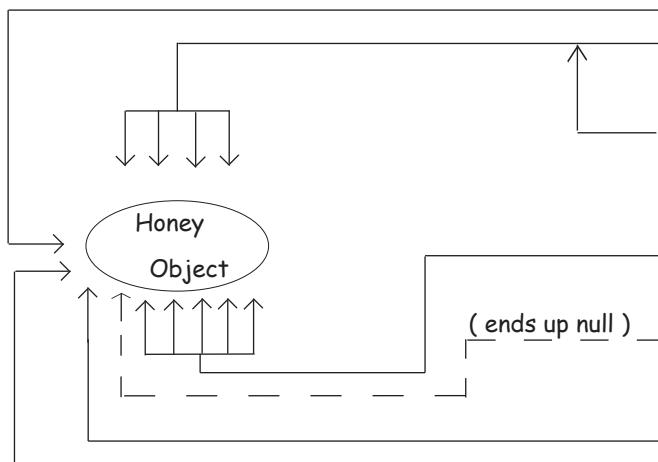
(from page 268)

- 1 copyGC = null; No—this line attempts to access a variable that is out of scope.
- 2 gc2 = null; OK—gc2 was the only reference variable referring to that object.
- 3 newGC = gc3; No—another out of scope variable.
  
- 4 gc1 = null; OK—gc1 had the only reference because newGC is out of scope.
- 5 newGC = null; No—newGC is out of scope.
  
- 6 gc4 = null; No—gc3 is still referring to that object.
  
- 7 gc3 = gc2; No—gc4 is still referring to that object.
  
- 8 gc1 = gc4; OK—Reassigning the only reference to that object.
  
- 9 gc3 = null; No—gc4 is still referring to that object.

## Popular Objects

(from page 269)

It probably wasn't too hard to figure out that the Honey object first referred to by the honeyPot variable is by far the most “popular” object in this class. But maybe it was a little trickier to see that all of the variables that point from the code to the Honey object refer to the **same object!** There are a total of 12 active references to this object right before the main() method completes. The kit.honeyPot variable is valid for a while, but kit gets nulled at the end. Since raccoon.rk still refers to the Kit object, raccoon.kit.honeyPot (although never explicitly declared) refers to the object!



```
public class Honey {
    public static void main(String[] args) {
        Honey honeyPot = new Honey();
        Honey[] ha = {honeyPot, honeyPot,
                      honeyPot, honeyPot};
        Bees bees = new Bees();
        bees.beeHoney = ha;
        Bear[] bears = new Bear[5];
        for (int i = 0; i < 5; i++) {
            bears[i] = new Bear();
            bears[i].hunny = honeyPot;
        }
        Kit kit = new Kit();
        kit.honey = honeyPot;
        Raccoon raccoon = new Raccoon();

        raccoon.rh = honeyPot;
        raccoon.rk = kit;
        kit = null;
    } // end of main
}
```



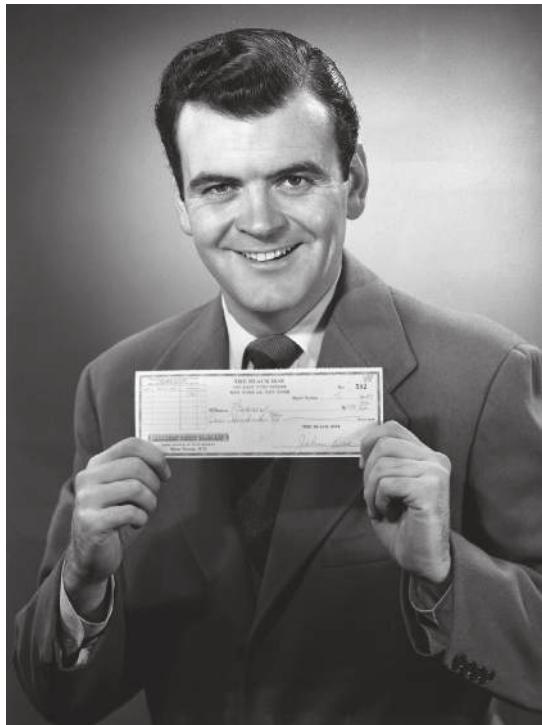
## Five-Minute Mystery (from pages 270–271)

Tom noticed that the constructor for the `V2Radiator` class took an `ArrayList`. That meant that every time the `V3Radiator` constructor was called, it passed an `ArrayList` in its `super()` call to the `V2Radiator` constructor. That meant that an extra five `V2Radiator` `SimUnits` were created. If Tom was right, total power use would have been 120, not the 100 that Sarah's expected ratios predicted.

Since all the Bot classes create `SimUnits`, writing a constructor for the `SimUnit` class, which printed out a line every time a `SimUnit` was created, would have quickly highlighted the problem!



# Numbers Matter



**Do the Math.** But there's more to working with numbers than just doing primitive arithmetic. You might want to get the absolute value of a number, or round a number, or find the larger of two numbers. You might want your numbers to print with exactly two decimal places, or you might want to put commas into your large numbers to make them easier to read. And what about parsing a String into a number? Or turning a number into a String? Someday you're gonna want to put a bunch of numbers into a collection like ArrayList that takes only objects. You're in luck. Java and the Java API are full of handy number-tweaking capabilities and methods, ready and easy to use. But most of them are **static**, so we'll start by learning what it means for a variable or method to be static, including constants in Java, also known as static *final* variables.

## MATH methods: as close as you'll ever get to a *global* method

Except there's no global *anything* in Java. But think about this: what if you have a method whose behavior doesn't depend on an instance variable value. Take the round() method in the Math class, for example. It does the same thing every time—rounds a floating-point number (the argument to the method) to the nearest integer. Every time. If you had 10,000 instances of class Math, and ran the round(42.2) method, you'd get an integer value of 42. Every time. In other words, the method acts on the argument but is never affected by an instance variable state. The only value that changes the way the round() method runs is the argument passed to the method!

Doesn't it seem like a waste of perfectly good heap space to make an instance of class Math simply to run the round() method? And what about *other* Math methods like min(), which takes two numerical primitives and returns the smaller of the two? Or max(). Or abs(), which returns the absolute value of a number.

### **These methods never use instance variable values.**

In fact, the Math class doesn't *have* any instance variables. So there's nothing to be gained by making an instance of class Math. So guess what? You don't have to. As a matter of fact, you can't.

### If you try to make an instance of class Math:

```
Math mathObject = new Math();
```

### You'll get this error:

```
File Edit Window Help IwasToldThereWouldBeNoMaths
% javac TestMath

TestMath.java:3: Math() has private
access in java.lang.Math
    Math mathObject = new Math();
                           ^
1 error
```

← This error shows that the Math constructor is marked private! That means you can NEVER say 'new' on the Math class to make a new Math object.

Methods in the Math class don't use any instance variable values. And because the methods are "static," you don't need to have an instance of Math. All you need is the Math class.

```
long x = Math.round(42.2);
int y = Math.min(56, 12);
int z = Math.abs(-343);
```

↑  
These methods never use instance variables, so their behavior doesn't need to know about a specific object.

# The difference between regular (non-static) and static methods

Java is object-oriented, but once in a while you have a special case, typically a utility method (like the Math methods), where there is no need to have an instance of the class. The keyword **static** lets a method run **without any instance of the class**. A static method means “behavior not dependent on an instance variable, so no instance/object is required. Just the class.”

## regular (non-static) method

```
public class Song {  
    String title;
```

Instance variable value affects the behavior of the play() method.

```
    public Song(String t) {  
        title = t;  
    }
```

```
    public void play() {  
        SoundPlayer player = new SoundPlayer();  
        player.playSound(title);  
    }  
}
```

Song
title
play()

The current value of the 'title' instance variable is the song that plays when you call play().

two instances of class Song



s2  
Song

s2.play();

Calling play() on this reference will cause "Politik" to play.



s3  
Song

s3.play();

Calling play() on this reference will cause "My Way" to play.

## static method

```
public static int min(int a, int b) {  
    //returns the smallest of a and b  
}
```

Math
min()
max()
abs()
...

No instance variables. The method behavior doesn't change with instance variable state.

Math.min(42, 36);

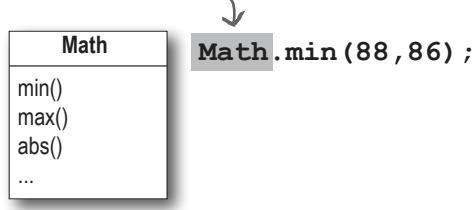
Use the Class name, rather than a reference variable name.



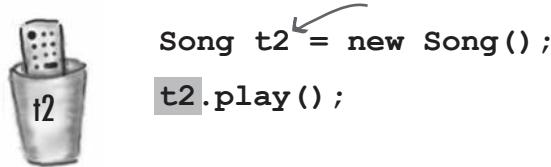
NO OBJECTS!!

Absolutely NO OBJECTS anywhere in this picture!

## Call a static method using a class name



## Call a non-static method using a reference variable name



## What it means to have a class with static methods

Often (although not always), a class with static methods is not meant to be instantiated. In Chapter 8, *Serious Polymorphism*, we talked about abstract classes, and how marking a class with the `abstract` modifier makes it impossible for anyone to say “new” on that class type. In other words, **it’s impossible to instantiate an abstract class.**

But you can restrict other code from instantiating a *non-abstract* class by marking the constructor `private`. Remember, a *method* marked private means that only code from within the class can invoke the method. A *constructor* marked private means essentially the same thing—only code from within the class can invoke the constructor. Nobody can say “new” from *outside* the class. That’s how it works with the `Math` class, for example. The constructor is private; you cannot make a new instance of `Math`. The compiler knows that your code doesn’t have access to that private constructor.

This does *not* mean that a class with one or more static methods should never be instantiated. In fact, every class you put a `main()` method in is a class with a static method in it!

Typically, you make a `main()` method so that you can launch or test another class, nearly always by instantiating a class in `main` and then invoking a method on that new instance.

So you’re free to combine static and non-static methods in a class, although even a single non-static method means there must be *some* way to make an instance of the class. The only ways to get a new object are through “new” or serialization (or something called the Java Reflection API that we don’t go into). No other way. But exactly *who* says new can be an interesting question, and one we’ll look at a little later in this chapter.

# Static methods can't use non-static (instance) variables!

Static methods run without knowing about any particular instance of the static method's class. And as you saw on the previous pages, there might not even *be* any instances of that class. Since a static method is called using the *class* (`Math.random()`) as opposed to an *instance reference* (`t2.play()`), a static method can't refer to any instance variables of the class. The static method doesn't know *which* instance's variable value to use.

## If you try to compile this code:

```
public class Duck {
    private int size;

    public static void main(String[] args) {
        System.out.println("Size of duck is " + size);
    }

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

    public int getSize() {
        return size;
    }
}
```

*Which Duck?  
Whose size?*

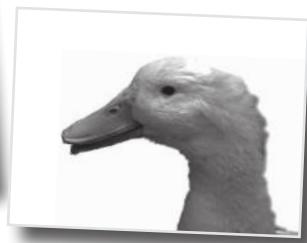
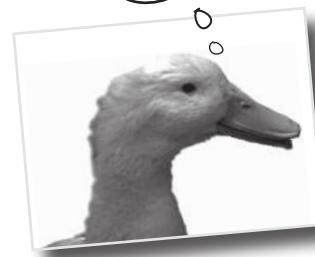
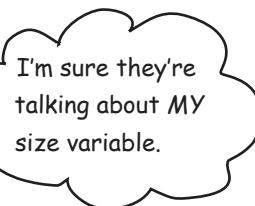
*If there's a Duck on  
the heap somewhere, we  
don't know about it.*

*Static context. Everything  
else in the class is NOT static.*

## You'll get this error:

```
File Edit Window Help Quack
% javac Duck.java
Duck.java:6: non-static variable
size cannot be referenced from a
static context
        System.out.println("Size
of duck is " + size);
^
```

If you try to use an instance variable from inside a static method, the compiler thinks, “I don’t know which object’s instance variable you’re talking about!” If you have ten Duck objects on the heap, a static method doesn’t know about any of them.



# Static methods can't use non-static methods, either!

What do non-static methods do? **They usually use instance variable state to affect the behavior of the method.** A getName() method returns the value of the name variable. Whose name? The object used to invoke the getName() method.

## This won't compile:

```
public class Duck {
    private int size;

    public static void main(String[] args) {
        System.out.println("Size is " + getSize());
    }

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

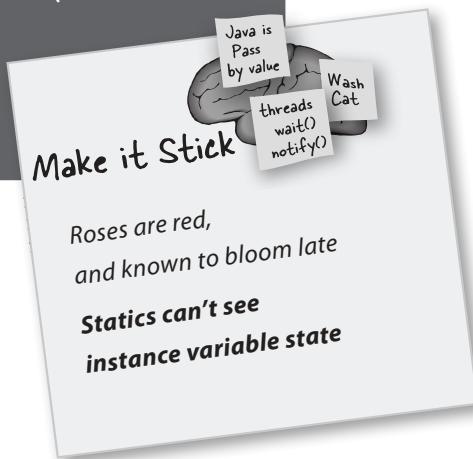
    public int getSize() {
        return size;
    }
}
```

Calling getSize() just postpones the inevitable—getSize() uses the size instance variable.

Back to the same problem...  
whose size?

```
File Edit Window Help Jack-in
% javac Duck.java
Duck.java:5: error: non-static
method getSize() cannot be refer-
enced from a static context

    System.out.println("Size is " +
 getSize());
^
1 error
```



there are no  
**Dumb Questions**

**Q:** What if you try to call a non-static method from a static method, but the non-static method doesn't use any instance variables. Will the compiler allow that?

**A:** No. The compiler knows that whether you do or do not use instance variables in a non-static method, you can. And think about the implications...if you were allowed to compile a scenario like that, then what happens if in the future you want to change the implementation of that non-static method so that one day it does use an instance variable? Or worse, what happens if a subclass overrides the method and uses an instance variable in the overriding version?

**Q:** I could swear I've seen code that calls a static method using a reference variable instead of the class name.

**A:** You can do that, but as your mother always told you, "Just because it's legal doesn't mean it's good." Although it works to call a static method using any instance of the class, it makes for misleading (less-readable) code. You can say,

```
Duck d = new Duck();
String[] s = {};
d.main(s);
```

This code is legal, but the compiler just resolves it back to the real class anyway ("OK, d is of type Duck, and main() is static, so I'll call the static main() in class Duck"). In other words, using d to invoke main() doesn't imply that main() will have any special knowledge of the object that d is referencing. It's just an alternate way to invoke a static method, but the method is still static!

# Static variable: value is the same for ALL instances of the class

Imagine you wanted to count how many Duck instances are being created while your program is running. How would you do it? Maybe an instance variable that you increment in the constructor?

```
class Duck {
    int duckCount = 0;
    public Duck() {
        duckCount++;
```

this would always set  
duckCount to 1 each time  
a Duck was made

No, that wouldn't work because duckCount is an instance variable, and starts at 0 for each Duck. You could try calling a method in some other class, but that's kludgy. You need a class that's got only a single copy of the variable, and all instances share that one copy.

That's what a static variable gives you: a value shared by all instances of a class. In other words, one value per *class*, instead of one value per *instance*.

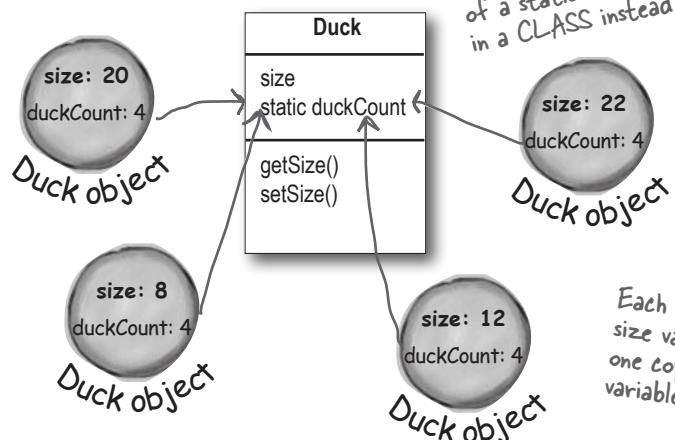
```
public class Duck {
    private int size;
    private static int duckCount = 0;
```

```
public Duck() {
    duckCount++;
```

Now it will keep  
incrementing each time  
the Duck constructor runs,  
because duckCount is static  
and won't be reset to 0.

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

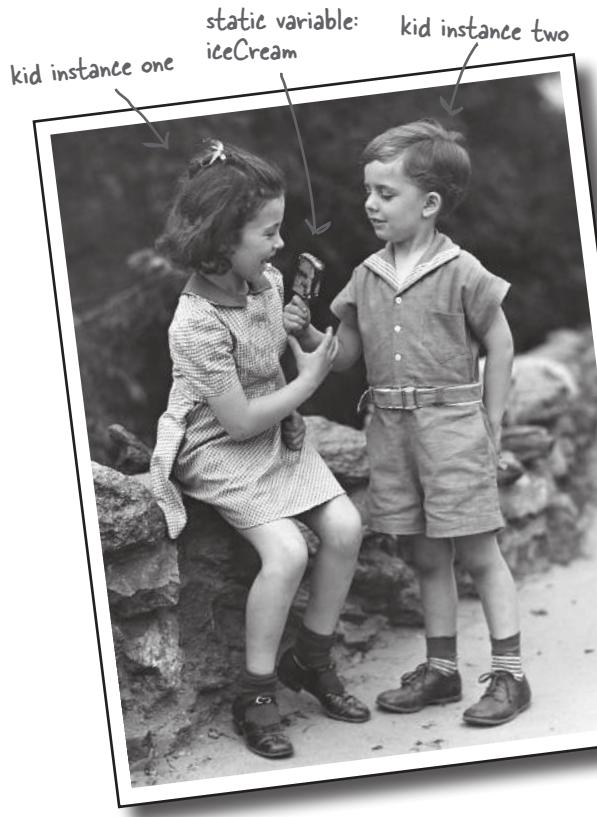
```
public int getSize() {
    return size;
}
```



A Duck object doesn't keep its own copy  
of duckCount.  
Because duckCount is static, Duck objects  
all share a single copy of it. You can think  
of a static variable as a variable that lives  
in a CLASS instead of in an object.

Each Duck object has its own  
size variable, but there's only  
one copy of the duckCount  
variable—the one in the class.

## static variables



**Static variables are shared.**

All instances of the same class share a single copy of the static variables.

instance variables: 1 per **instance**  
static variables: 1 per **class**



## Brain Barbell

Earlier in this chapter, we saw that a private constructor means that the class can't be instantiated from code running outside the class. In other words, only code from within the class can make a new instance of a class with a private constructor. (There's a kind of chicken-and-egg problem here.)

What if you want to write a class in such a way that only ONE instance of it can be created, and anyone who wants to use an instance of the class will always use that one, single instance?

# Initializing a static variable

Static variables are initialized when a *class is loaded*. A class is loaded because the JVM decides it's time to load it. Typically, the JVM loads a class because somebody's trying to make a new instance of the class, for the first time, or use a static method or variable of the class. As a programmer, you also have the option of telling the JVM to load a class, but you're not likely to need to do that. In nearly all cases, you're better off letting the JVM decide when to *load* the class.

And there are two guarantees about static initialization:

- Static variables in a class are initialized before any *object* of that class can be created.
- Static variables in a class are initialized before any *static method* of the class runs.

```
class Player {
    static int playerCount = 0;
    private String name;
    public Player(String n) {
        name = n;
        playerCount++;
    }
}

public class PlayerTestDrive {
    public static void main(String[] args) {
        System.out.println(Player.playerCount);
        Player one = new Player("Tiger Woods");
        System.out.println(Player.playerCount);
    }
}
```

The playerCount is initialized when the class is loaded. We explicitly initialized it to 0, but we don't need to since 0 is the default value for ints. Static variables get default values just like instance variables.

Access a static variable just like a static method—with the class name.

If you don't explicitly initialize a static variable (by assigning it a value at the time you declare it), it gets a default value, so int variables are initialized to zero, which means we didn't need to explicitly say playerCount = 0. Declaring, but not initializing, a static variable means the static variable will get the default value for that variable type, in exactly the same way that instance variables are given default values when declared.

**All static variables in a class are initialized before any object of that class can be created.**

Default values for declared but uninitialized static and instance variables are the same:

Primitive integers (long, short, etc.): 0

Primitive floating points (float, double): 0.0

boolean: false

object references: null

File Edit Window Help What?

% java PlayerTestDrive

0 ← Before any instances are made

1 ← After an object is created

## static final variables are constants

A variable marked **final** means that—once initialized—it can never change. In other words, the value of the static final variable will stay the same as long as the class is loaded. Look up Math.PI in the API, and you'll find:

```
public static final double PI = 3.141592653589793;
```

The variable is marked **public** so that any code can access it.

The variable is marked **static** so that you don't need an instance of class Math (which, remember, you're not allowed to create).

The variable is marked **final** because PI doesn't change (as far as Java is concerned).

There is no other way to designate a variable as a constant, but there is a naming convention that helps you to recognize one. **Constant variable names are usually in all caps!**

A **static initializer** is a block of code that runs when a class is loaded, before any other code can use the class, so it's a great place to initialize a static final variable.

```
class ConstantInit1 {  
    final static int X;  
    static {  
        X = 42;  
    }  
}
```

### Initialize a *final* static variable:

- ① At the time you declare it:

```
public class ConstantInit2 {  
    public static final int X_VALUE = 25;  
}  
} Notice the naming convention—static  
final variables are constants, so the  
name should be all uppercase, with an  
underscore separating the words.
```

OR

- ② In a static initializer:

```
public class ConstantInit3 {  
    public static final double VAL;  
  
    static {  
        VAL = Math.random();  
    }  
}
```

This code runs as soon as the class is loaded, before any static method variable can be used.

If you don't give a value to a final variable in one of those two places:

```
public class ConstantInit3 {  
    public static final double VAL;  
}  
} no initialization!
```

The compiler will catch it:

```
File Edit Window Help Init?  
% javac ConstantInit3.java  
ConstantInit3.java:2: error: vari-  
able VAL not initialized in the  
default constructor  
    public static final double VAL;  
                           ^  
1 error
```

# final isn't just for static variables...

You can use the keyword **final** to modify non-static variables too, including instance variables, local variables, and even method parameters. In each case, it means the same thing: the value can't be changed. But you can also use final to stop someone from overriding a method or making a subclass.

## non-static final variables

```
class Foof {
    final int size = 3;   ← now you can't change size
    final int whuffie;

    Foof() {
        whuffie = 42;   ← now you can't change whuffie
    }

    void doStuff(final int x) {
        // you can't change x
    }

    void doMore() {
        final int z = 7;
        // you can't change z
    }
}
```

## final method

```
class Poof {
    final void calcWhuffie() {
        // important things
        // that must never be overridden
    }
}
```

## final class

```
final class MyMostPerfectClass {
    // cannot be extended
}
```

**A final variable means you can't change its value.**

**A final method means you can't override the method.**

**A final class means you can't extend the class (i.e., you can't make a subclass).**

It's all so...so final.  
I mean, if I'd known  
I wouldn't be able to  
change things...



## there are no Dumb Questions

**Q:** A static method can't access a non-static variable. But can a non-static method access a static variable?

**A:** Of course. A non-static method in a class can always call a static method in the class or access a static variable of the class.

**Q:** Why would I want to make a class final? Doesn't that defeat the whole purpose of OO?

**A:** Yes and no. A typical reason for making a class final is for security. You can't, for example, make a subclass of the String class. Imagine the havoc if someone extended the String class and substituted their own String subclass objects, polymorphically, where String objects are expected. If you need to count on a particular implementation of the methods in a class, make the class final.

**Q:** Isn't it redundant to have to mark the methods final if the class is final?

**A:** If the class is final, you don't need to mark the methods final. Think about it—if a class is final, it can never be subclassed, so none of the methods can ever be overridden.

On the other hand, if you do want to allow others to extend your class and you want them to be able to override some, but not all, of the methods, then don't mark the class final, but go in and selectively mark specific methods as final. A final method means that a subclass can't override that particular method.

### BULLET POINTS

- A **static method** should be called using the class name rather than an object reference variable: `Math.random()` versus `myFoo.go()`
- A static method can be invoked without any instances of the method's class on the heap.
- A static method is good for a utility method that does not (and will never) depend on a particular instance variable value.
- A static method is not associated with a particular instance—only the class—so it cannot access any instance variable values of its class. It wouldn't know which instance's values to use.
- A static method cannot access a non-static method, since non-static methods are usually associated with instance variable state.
- If you have a class with only static methods and you do not want the class to be instantiated, you can mark the constructor private.
- A **static variable** is a variable shared by all members of a given class. There is only one copy of a static variable in a class, rather than one copy *per each object* for instance variables.
- A static method can access a static variable.
- To make a constant in Java, mark a variable as both static and final.
- A final static variable must be assigned a value either at the time it is declared or in a static initializer:

```
static {  
    DOG_CODE = 420;  
}
```
- The naming convention for constants (final static variables) is to make the name all uppercase and use underscores (\_) to separate the words.
- A final variable value cannot be changed once it has been assigned.
- Assigning a value to a final *instance* variable must be either at the time it is declared or in the constructor.
- A final method cannot be overridden.
- A final class cannot be extended (subclassed).



## What's Legal?

Given everything you've just learned about static and final, which of these would compile?



- ```

① public class Foo {
    static int x;

    public void go() {
        System.out.println(x);
    }
}

② public class Foo2 {
    int x;

    public static void go() {
        System.out.println(x);
    }
}

③ public class Foo3 {
    final int x;

    public void go() {
        System.out.println(x);
    }
}

④ public class Foo4 {
    static final int x = 12;

    public void go() {
        System.out.println(x);
    }
}

⑤ public class Foo5 {
    static final int x = 12;

    public void go(final int x) {
        System.out.println(x);
    }
}

⑥ public class Foo6 {
    int x = 12;

    public static void go(final int x) {
        System.out.println(x);
    }
}
  
```

→ Answers on page 308.

# Math methods

Now that we know how static methods work, let's look at some static methods in class Math. This isn't all of them, just the highlights. Check your API for the rest including cos(), sin(), tan(), ceil(), floor(), and asin().

| All Methods       | Static Methods                          | Concrete Methods                                                                                                |  |
|-------------------|-----------------------------------------|-----------------------------------------------------------------------------------------------------------------|--|
| Modifier and Type | Method                                  | Description                                                                                                     |  |
| static double     | abs(double a)                           | Returns the absolute value of a double.                                                                         |  |
| static float      | abs(float a)                            | Returns the absolute value of a float.                                                                          |  |
| static int        | abs(int a)                              | Returns the absolute value of an integer.                                                                       |  |
| static long       | abs(long a)                             | Returns the absolute value of a long.                                                                           |  |
| static int        | absExact(int a)                         | Returns the mathematical absolute value of an integer. Throws an ArithmeticException if the result is negative. |  |
| static long       | absExact(long a)                        | Returns the mathematical absolute value of a long. Throws an ArithmeticException if the result is negative.     |  |
| static double     | acos(double a)                          | Returns the arc cosine of a value.                                                                              |  |
| static int        | addExact(int x, int y)                  | Returns the sum of its arguments.                                                                               |  |
| static long       | addExact(long x, long y)                | Returns the sum of its arguments.                                                                               |  |
| static double     | asin(double a)                          | Returns the arc sine of a value; the result is in radians.                                                      |  |
| static double     | atan(double a)                          | Returns the arc tangent of a value; the result is in radians.                                                   |  |
| static double     | atan2(double y, double x)               | Returns the angle theta from the x-axis to the point (x, y).                                                    |  |
| static double     | cbrt(double a)                          | Returns the cube root of a double.                                                                              |  |
| static double     | ceil(double a)                          | Returns the smallest (closest to negative infinity) mathematical integer.                                       |  |
| static double     | copySign(double magnitude, double sign) | Returns the first floating-point argument with the sign of the second.                                          |  |
| static float      | copySign(float magnitude, float sign)   | Returns the first floating-point argument with the sign of the second.                                          |  |
| static double     | cos(double a)                           | Returns the trigonometric cosine of an angle.                                                                   |  |
| static double     | cosh(double x)                          | Returns the hyperbolic cosine of a double.                                                                      |  |

## Math.abs()

Returns a double that is the absolute value of the argument. The method is overloaded, so if you pass it an int, it returns an int. Pass it a double, it returns a double.

```
int x = Math.abs(-240);           // returns 240
double d = Math.abs(240.45);     // returns 240.45
```

## Math.random()

Returns a double between (and including) 0.0 through (but not including) 1.0.

```
double r1 = Math.random();
int r2 = (int) (Math.random() * 5);
```

We've been using this method so far, but there's also java.util.Random, which is a bit nicer to use.

## Math.round()

Returns an int or a long (depending on whether the argument is a float or a double) rounded to the nearest integer value.

*Remember, floating-point literals are assumed to be doubles unless you add the 'f.'*

```
int x = Math.round(-24.8f); // returns -25
int y = Math.round(24.45f); // returns 24

long z = Math.round(24.45); // returns 24L
                           ↘
                           This is a double.
```

## Math.min()

Returns a value that is the minimum of the two arguments. The method is overloaded to take ints, longs, floats, or doubles.

```
int x = Math.min(24, 240); // returns 24
double y = Math.min(90876.5, 90876.49); // returns 90876.49
```

## Math.max()

Returns a value that is the maximum of the two arguments. The method is overloaded to take ints, longs, floats, or doubles.

```
int x = Math.max(24, 240); // returns 240
double y = Math.max(90876.5, 90876.49); // returns 90876.5
```

## Math.sqrt()

Returns the positive square root of the argument. The method takes a double, but of course you can pass in anything that fits in a double.

```
double x = Math.sqrt(9); // return 3
double y = Math.sqrt(42.0); // returns 6.48074069840786
```

# Wrapping a primitive

Sometimes you want to treat a primitive like an object. For example, collections like ArrayList only work with Objects:

```
ArrayList<???> list;
```

*Can we create an  
ArrayList for ints?*

There's a wrapper class for every primitive type, and since the wrapper classes are in the java.lang package, you don't need to import them. You can recognize wrapper classes because each one is named after the primitive type it wraps, but with the first letter capitalized to follow the class naming convention.

Oh yeah, for reasons absolutely nobody on the planet is certain of, the API designers decided not to map the names *exactly* from primitive type to class type. You'll see what we mean:

**Boolean**

**Character**

**Byte**

**Short**

**Integer**

**Long**

**Float**

**Double**

## wrapping a value

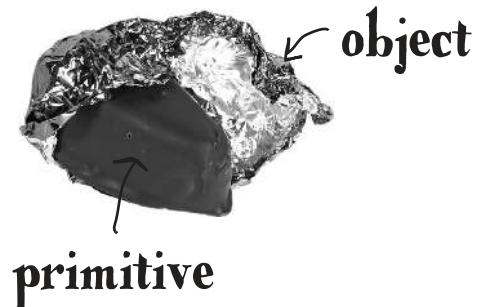
```
int i = 288;
Integer iWrap = new Integer(i);
```

*Give the primitive to the wrapper constructor. That's it.*

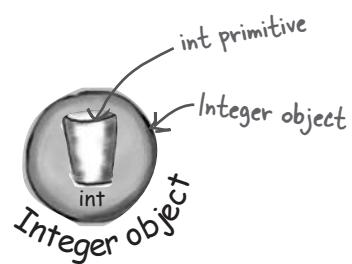
*All the wrappers work like this. Boolean has a booleanValue(), Character has a charValue(), etc.*

## unwrapping a value

```
int unWrapped = iWrap.intValue();
```



**When you need to treat a primitive like an object, wrap it.**



Note: the picture at the top is a chocolate in a foil wrapper. Get it? Wrapper? Some people think it looks like a baked potato, but that works too.



This is stupid. You mean I can't just make an ArrayList of ints??? I have to wrap every single frickin' one in a new Integer object and then unwrap it when I try to access that value in the ArrayList? That's a waste of time and an error waiting to happen...

## Java will Autobox primitives for you

In The Olden Days (pre-Java 5), we did have to do all this ourselves, manually wrapping and unwrapping primitives. Fortunately, now it's all done for us *automatically*.

Let's see what happens when we want to make an ArrayList to hold ints.

### An ArrayList of primitive ints

```
public void autoboxing() {
    int x = 32;
    ArrayList<Integer> list = new ArrayList<Integer>();
    list.add(x); Just add it!
    int num = list.get(0);
}
```

And the compiler automatically unwraps (unboxes) the Integer object so you can assign the int value directly to a primitive without having to call the intValue() method on the Integer object.

Make an ArrayList of type Integer.



Although there is NOT a method in ArrayList for add(int), the compiler does all the wrapping (boxing) for you. In other words, there really IS an Integer object stored in the ArrayList, but you get to "pretend" that the ArrayList takes ints. (You can add both ints and Integers to an ArrayList<Integer>.)

*there are no*  
**Dumb Questions**

**Q:** Why not declare an ArrayList<int> if you want to hold ints?

**A:** Because...you can't. At least, not in the versions of Java this book covers (the language is constantly evolving and things may change!). Remember, the rule for generic types is that you can specify only class or interface types, *not primitives*. So ArrayList<int> will not compile. But as you can see from the code above, it doesn't really matter, since the compiler lets you put ints into the ArrayList<Integer>. In fact, there's really no way to prevent you from putting primitives into an ArrayList where the type of the list is the type of that primitive's wrapper, since autoboxing will happen automatically. So, you can put boolean primitives in an ArrayList<Boolean> and chars into an ArrayList<Character>.

# Autoboxing works almost everywhere

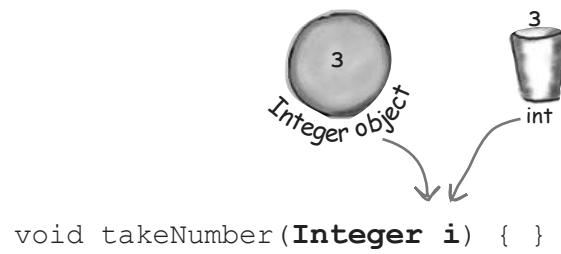
Autoboxing lets you do more than just the obvious wrapping and unwrapping to use primitives in a collection...it also lets you use either a primitive or its wrapper type virtually anywhere one or the other is expected. Think about that!

## Fun with autoboxing

---

### Method arguments

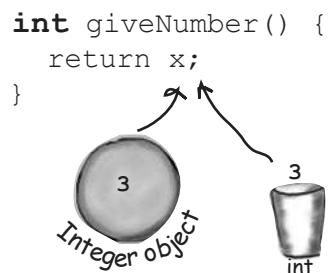
If a method takes a wrapper type, you can pass a reference to a wrapper or a primitive of the matching type. And of course the reverse is true—if a method takes a primitive, you can pass in either a compatible primitive or a reference to a wrapper of that primitive type.



---

### Return values

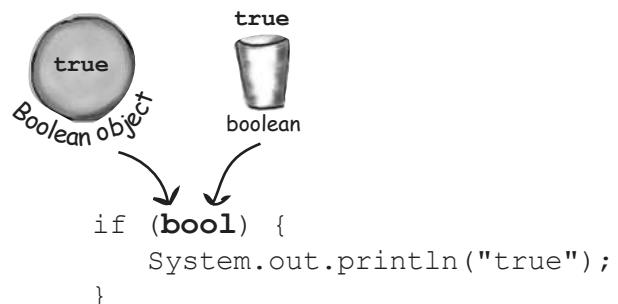
If a method declares a primitive return type, you can return either a compatible primitive or a reference to the wrapper of that primitive type. And if a method declares a wrapper return type, you can return either a reference to the wrapper type or a primitive of the matching type.



---

### Boolean expressions

Any place a boolean value is expected, you can use either an expression that evaluates to a boolean ( $4 > 2$ ), a primitive boolean, or a reference to a Boolean wrapper.



## Operations on numbers

This is probably the strangest one—yes, you can use a wrapper type as an operand in operations where the primitive type is expected. That means you can apply, say, the increment operator against a reference to an Integer object!

But don't worry—this is just a compiler trick. The language wasn't modified to make the operators work on objects; the compiler simply converts the object to its primitive type before the operation. It sure looks weird, though.

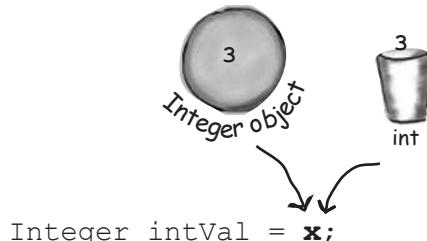
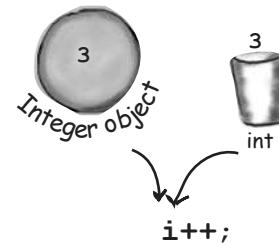
```
Integer i = new Integer(42);
i++;
```

And that means you can also do things like:

```
Integer j = new Integer(5);
Integer k = j + 3;
```

## Assignments

You can assign either a wrapper or primitive to a variable declared as a matching wrapper or primitive. For example, a primitive int variable can be assigned to an Integer reference variable, and vice versa—a reference to an Integer object can be assigned to a variable declared as an int primitive.



## Sharpen your pencil

Will this code compile? Will it run? If it runs, what will it do?

Take your time and think about this one; it brings up an implication of autoboxing that we didn't talk about.

You'll have to go to your compiler to find the answers. (Yes, we're forcing you to experiment, for your own good, of course.)

→ Yours to solve.

```
public class TestBox {
    private Integer i;
    private int j;

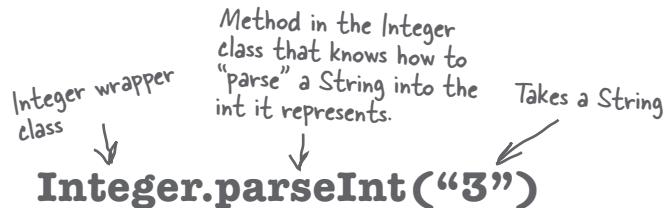
    public static void main(String[] args) {
        TestBox t = new TestBox();
        t.go();
    }

    public void go() {
        j = i;
        System.out.println(j);
        System.out.println(i);
    }
}
```

# But wait! There's more! Wrappers have static utility methods too!

Besides acting like a normal class, the wrappers have a bunch of really useful static methods.

For example, the *parse* methods take a String and give you back a primitive value.



**Converting a String to a primitive value is easy:**

```
String s = "2";
int x = Integer.parseInt(s);
double d = Double.parseDouble("420.24");
boolean b = Boolean.parseBoolean("True");
```

No problem to parse "2" into 2.

The `parseBoolean()` method ignores the cases of the characters in the String argument.

**But if you try to do this:**

```
String t = "two";
int y = Integer.parseInt(t);
```

Uh-oh. This compiles just fine, but at runtime it blows up. Anything that can't be parsed as a number will cause a `NumberFormatException`.

**You'll get a runtime exception:**

```
File Edit Window Help Clue
% java Wrappers
Exception in thread "main" java.lang.NumberFormatException:
For input string: "two"
        at java.base/java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)
        at java.base/java.lang.Integer.parseInt(Integer.java:652)
        at java.base/java.lang.Integer.parseInt(Integer.java:770)
        at Snippets.badParse(Snippets.java:48)
        at Snippets.main(Snippets.java:54)
```

**Every method or constructor that parses a String can throw a `NumberFormatException`. It's a runtime exception, so you don't have to handle or declare it. But you might want to.**

(We'll talk about exceptions in Chapter 13, *Risky Behavior*.)

# And now in reverse...turning a primitive number into a String

You may want to turn a number into a String, for example when you want to show this number to a user or put it into a message. There are several ways to turn a number into a String. The easiest is to simply concatenate the number to an existing String.

```
double d = 42.5;
String doubleString = "" + d;
```

Remember the 't' operator is overloaded in Java (the only overloaded operator) as a String concatenator. Anything added to a String becomes Stringified.

```
double d = 42.5;
String doubleString = Double.toString(d);
```

Another way to do it using a static method in class Double.

```
double d = 42.5;
String doubleString = String.valueOf(d);
```

There's also an overloaded a static method "valueOf" on String that will get the String value of pretty much anything.

Yeah,  
but how do I make  
it look like money? With a  
dollar sign and two decimal  
places like \$56.87 or what if I  
want commas like 45,687,890  
or what if I want it in...

Where's my printf  
like I have in C? Is  
number formatting part of  
the I/O classes?



# Number formatting

In Java, formatting numbers and dates doesn't have to be coupled with I/O. Think about it. One of the most typical ways to display numbers to a user is through a GUI. You put Strings into a scrolling text area, or maybe a table. If formatting was built only into print statements, you'd never be able to format a number into a nice String to display in a GUI.

The Java API provides powerful and flexible formatting using the `Formatter` class in `java.util`. But often you don't need to create and call methods on the `Formatter` class yourself, because the Java API has convenience methods in some of the I/O classes (including `printf()`) and the `String` class. So it can be a simple matter of calling a static `String.format()` method and passing it the thing you want formatted along with formatting instructions.

Of course, you do have to know how to supply the formatting instructions, and that takes a little effort unless you're familiar with the `printf()` function in C/C++. Fortunately, even if you *don't* know `printf()`, you can simply follow recipes for the most basic things (that we're showing in this chapter). But you *will* want to learn how to format if you want to mix and match to get *anything* you want.

We'll start here with a basic example and then look at how it works. (Note: we'll revisit formatting again in Chapter 16, *Saving Objects*.)

## Making big numbers more readable with underscores, a quick detour

Before we get into formatting numbers, let's take a small, useful detour. Sometimes you'll want to declare variables with large initial values. Let's look at three declarations that assign the same large value, a billion, to long primitives:

```
long hardToRead = 1000000000;
long easierToRead = 1_000_000_000; ←
long legalButSilly = 10_0000_0000;
```

*When you're assigning large values, properly located underscores will make your life easier!*

## Formatting a number to use commas

```
public class TestFormats {
    public static void main(String[] args) {
        long myBillion = 1_000_000_000;
        String s = String.format("%,d", myBillion);
        System.out.println(s);
    }
}
```

*The number to format (we want it to have commas).*

*The formatting instructions for how to format the second argument (which in this case is an int value). Remember, there are only two arguments to this method here—the first comma is INSIDE the String literal, so it isn't separating arguments to the format method.*

*Now we get commas inserted into the number.*

1,000,000,000

# Formatting deconstructed...

At the most basic level, formatting consists of two main parts (there is more, but we'll start with this to keep it cleaner):

## ① Formatting instructions

You use special format specifiers that describe how the argument should be formatted.

## ② The argument to be formatted.

Although there can be more than one argument, we'll start with just one. The argument type can't be just *anything*...it has to be something that can be formatted using the format specifiers in the formatting instructions. For example, if your formatting instructions specify a *floating-point number*, you can't pass in a Dog or even a String that looks like a floating-point number.

Note: if you already know printf() from C/C++, you can probably just skim the next few pages. Otherwise, read carefully!

Do this...                                          to this.

①                                                      ②

```
format("%,d", 1_000_000_000);
```

Use these instructions...on this argument.

## What do these instructions actually say?

“Take the second argument to this method, and format it as a **d**ecimal integer and insert **commas**.”

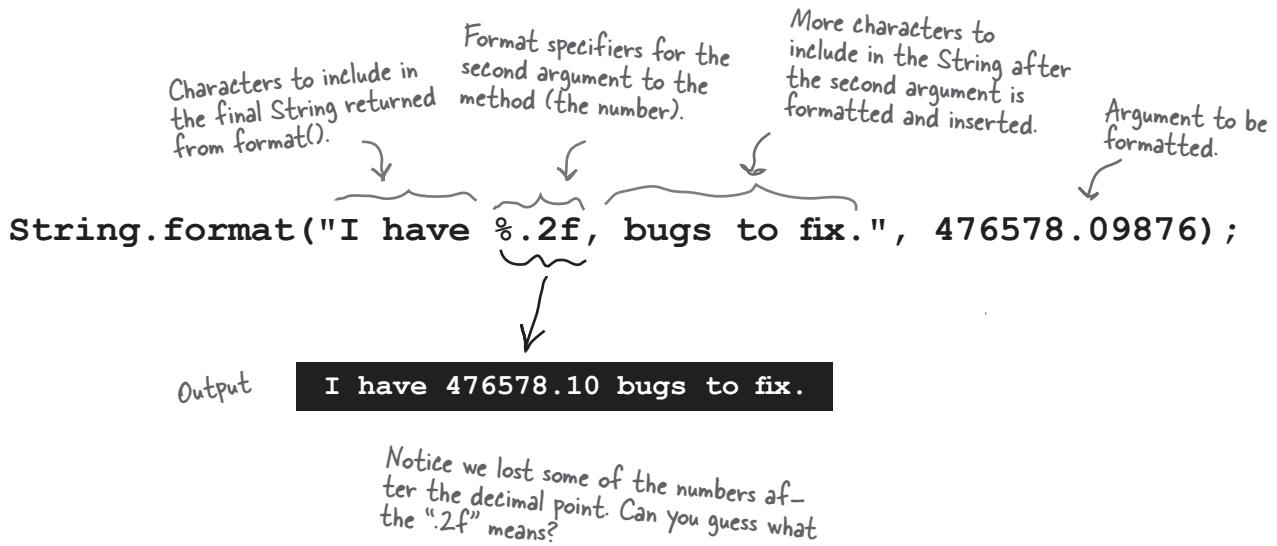
## How do they say that?

On the next page we'll look in more detail at what the syntax “%**d**” actually means, but for starters, any time you see the percent sign (%) in a format String (which is always the first argument to a format() method), think of it as representing a variable, and the variable is the other argument to the method. The rest of the characters after the percent sign describe the formatting instructions for the argument.

## the format() method

# The percent (%) says, “insert argument here” (and format it using these instructions)

The first argument to a format() method is called the format String, and it can actually include characters that you just want printed as-is, without extra formatting. When you see the % sign, though, think of the percent sign as a variable that represents the other argument to the method.



The “%” sign tells the formatter to insert the other method argument (the second argument to format(), the number) here, AND format it using the “.2f” characters after the percent sign. Then the rest of the format String, “bugs to fix,” is added to the final output.

## Adding a comma

```
String.format("I have %,.2f bugs to fix.", 476578.09876);
```

```
I have 476,578.10 bugs to fix.
```

By changing the format instructions from "%,.2f" to "%,2f", we got a comma in the formatted number.



But how does it even KNOW where the instructions end and the rest of the characters begin? How come it doesn't print out the "f" in "%.2f"? Or the "2"? How does it know that the .2f was part of the instructions and NOT part of the String?

## The format String uses its own little language syntax

You obviously can't put just *anything* after the "%" sign. The syntax for what goes after the percent sign follows very specific rules, and describes how to format the argument that gets inserted at that point in the result (formatted) String.

You've already seen some examples:

**%,d** means “insert commas and format the number as a decimal integer.”

and

**.2f** means “format the number as a floating point with a precision of two decimal places.”

and

**,.2f** means “insert commas and format the number as a floating point with a precision of two decimal places.”

Really the question is: “How do I know what to put after the percent sign to get it to do what I want?” And that includes knowing the symbols (like “d” for decimal and “f” for floating point) as well as the order in which the instructions must be placed following the percent sign. For example, if you put the comma after the “d” like “%d,” instead of “%,d” it won’t work!

Or will it? What do you think this will do:

```
String.format("I have %.2f, bugs to fix.", 476578.09876);
```

(We'll answer that on the next page.)

# The format specifier

Everything after the percent sign up to and including the type indicator (like “d” or “f”) is part of the formatting instructions. After the type indicator, the formatter assumes the next set of characters is meant to be part of the output String, until or unless it hits another percent (%) sign. Hmmmm...is that even possible? Can you have more than one formatted argument variable? Put that thought on hold for right now; we'll come back to it in a few minutes. For now, let's look at the syntax for the format specifiers—the things that go after the percent (%) sign and describe how the argument should be formatted.

**A format specifier can have up to five different parts (not including the “%”). Everything in brackets [ ] below is optional, so only the percent (%) and the type are required. But the order is also mandatory, so any parts you DO use must go in this order.**

% [argument number] [flags] [width] [.precision] **type**

We'll get to this later...  
it lets you say WHICH  
argument if there's more  
than one. (Don't worry  
about it just yet.)

These are for special  
formatting options  
like inserting commas,  
putting negative  
numbers in paren-  
theses, or making  
the numbers left  
justified.

This defines the MINI-  
MUM number of char-  
acters that will be used.  
That's \*minimum\* not  
TOTAL. If the number  
is longer than the width,  
it'll still be used in full,  
but if it's less than the  
width, it'll be padded  
with zeros.

You already know  
this one...it defines  
the precision. In  
other words, it  
sets the number  
of decimal places.  
Don't forget to  
include the “.” in  
there.

Type is mandatory  
(see the next page)  
and will usually be  
“d” for a decimal  
integer or “f” for  
a floating-point  
number.

% [argument number] [flags] [width] [.precision] **type**

format ("% , 6.1f", 42.000);

There's no “argument number”  
specified in this format String,  
but all the other pieces are there.

The value we  
want to format.  
Quite important.

# The only required specifier is for TYPE

Although type is the only required specifier, remember that if you *do* put in anything else, type must always come last! There are more than a dozen different type modifiers (not including dates and times; they have their own set), but most of the time you'll probably use %d (decimal) or %f (floating point). And typically you'll combine %f with a precision indicator to set the number of decimal places you want in your output.

**The TYPE is mandatory, everything else is optional.**

**%d**    **decimal**  
`format("%d", 42);`  


A 42.25 would not work! It would be the same as trying to directly assign a double to an int variable.

The argument must be compatible with an int, so that means only byte, short, int, and char (or their wrapper types).

**%f**    **floating point**  
`format("%.3f", 42.000000)`  


Here we combined the "f" with a precision indicator ".3" so we ended up with three zeros.

The argument must be of a floating-point type, so that means only a float or double (primitive or wrapper) as well as something called BigDecimal (which we don't look at in this book).

You must include a type in your format instructions, and if you specify things besides type, the type must always come last. Most of the time, you'll probably format numbers using either "d" for decimal or "f" for floating point.

**%x**    **hexadecimal**  
`format("%x", 42)`  


The argument must be a byte, short, int, long (including both primitive and wrapper types), and BigInteger.

**%c**    **character**  
`format("%c", 42)`  


The number 42 represents the char "\*".

The argument must be a byte, short, char, or int (including both primitive and wrapper types).

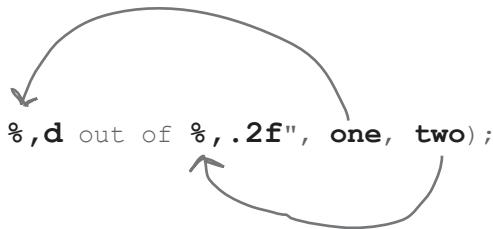
# What happens if I have more than one argument?

Imagine you want a String that looks like this:

“The rank is **20,456,654** out of **100,567,890.24**.”

But the numbers are coming from variables. What do you do? You simply add *two* arguments after the format String (first argument), so that means your call to `format()` will have three arguments instead of two. And inside that first argument (the format String), you’ll have two different format specifiers (two things that start with “%”). The first format specifier will insert the second argument to the method, and the second format specifier will insert the third argument to the method. In other words, the variable insertions in the format String use the order in which the other arguments are passed into the `format()` method.

```
int one = 20456654;  
double two = 100567890.248907;  
String s = String.format("The rank is %,d out of %,.2f", one, two);
```



**The rank is 20,456,654 out of 100,567,890.25**

We added commas to both variables and restricted the floating-point number (the second variable) to two decimal places.

When you have more than one argument, they’re inserted using the order in which you pass them to the `format()` method.

As you’ll see when we get to date formatting, you might actually want to apply different formatting specifiers to the same argument. That’s probably hard to imagine until you see how *date* formatting (as opposed to the *number* formatting we’ve been doing) works. Just know that in a minute, you’ll see how to be more specific about which format specifiers are applied to which arguments.

there are no  
Dumb Questions

**Q:** Um, there’s something REALLY strange going on here. Just how many arguments can I pass? I mean, how many overloaded `format()` methods are IN the `String` class? So, what happens if I want to pass, say, ten different arguments to be formatted for a single output String?

**A:** Good catch. Yes, there *is* something strange going on, and no there are *not* a bunch of overloaded `format()` methods to take a different number of possible arguments. In order to support this formatting (printf-like) API in Java, the language needed another feature—*variable argument lists* (called *varargs* for short). We’ll talk about varargs more in Appendix B.

# Just one more thing...static imports

Static imports are a real mixed blessing. Some people love this idea, some people hate it. Static imports exist to make your code a little shorter. If you hate to type or hate long lines of code, you might just like this feature. The downside to static imports is that—if you're not careful—using them can make your code a lot harder to read.

The basic idea is that whenever you're using a static class, a static variable, or an enum (more on those later), you can import them and save yourself some typing.

## Without static imports:

```
class NoStatic {
    public static void main(String[] args) {
        System.out.println("sqrt " + Math.sqrt(2.0));
        System.out.println("tan " + Math.tan(60));
    }
}
```

## Same code, with static imports:

```
import static java.lang.Math.*;
import static java.lang.System.out;
class WithStatic {
    public static void main(String[] args) {
        out.println("sqrt " + sqrt(2.0));
        out.println("tan " + tan(60));
    }
}
```

*This might be a BAD place to use a static import. Removing the "System" class makes it unclear what this is and where it came from. Also it may lead to naming conflicts; you can't create any other variables called "out" now.*

*The syntax to use when declaring static imports.*

*Static imports in action.*

*You might want to use static imports for these methods. It makes the code shorter, and you don't need the "Math." prefix to understand what these operations are.*

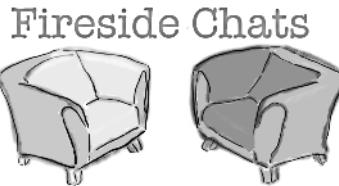
## Use carefully:

Static imports can make your code confusing to read.

Always re-read your code after using a static import and think: "Will I understand this in six months time?"

## Caveats & Gotchas

- Using a static import removes the information about which class the static came from. We'd advise using static imports only when the static method or variable still means something when it's not prefixed with the class name.
- A big issue with static imports is that it's easy to create naming conflicts. For example, if you have two different classes with an "add()" method, how will you and the compiler know which one to use? So it's best not to use a static import if it's possible to create a conflict.
- Notice that you can use wildcards (\*), in your static import declaration.



## Fireside Chats

Tonight's Talk: **An instance variable takes cheap shots at a static variable**

### Instance Variable

I don't even know why we're doing this. Everyone knows static variables are just used for constants. And how many of those are there? I think the whole API must have, what, four? And it's not like anybody ever uses them.

Full of it. Yeah, you can say that again. OK, so there are a few in the Swing library, but everybody knows Swing is just a special case.

Ok, but besides a few GUI things, give me an example of just one static variable that anyone would actually use. In the real world.

Well, that's another special case. And nobody uses that except for debugging anyway.

### Static Variable

You really should check your facts. When was the last time you looked at the API? It's frickin' loaded with statics! It even has entire classes dedicated to holding constant values. There's a class called `SwingConstants`, for example, that's just full of them.

It might be a special case, but it's a really important one! And what about the `Color` class? What a pain if you had to remember the RGB values to make the standard colors! But the `color` class already has constants defined for blue, purple, white, red, etc. Very handy.

How's `System.out` for starters? The `out` in `System.out` is a static variable of the `System` class. You personally don't make a new instance of the `System`; you just ask the `System` class for its `out` variable.

Oh, like debugging isn't important?

And here's something that probably never crossed your narrow mind—let's face it, static variables are more efficient. One per class instead of one per instance. The memory savings might be huge!

**Instance Variable**

Um, aren't you forgetting something?

Static variables are about as un-OO as it gets!!  
 Gee, why not just go take a giant backward step  
 and do some procedural programming while  
 we're at it.

You're like a global variable, and any programmer  
 worth their sticker-covered laptop knows that's  
 usually a Bad Thing.

Yeah, you live in a class, but they don't call it  
*Class-Oriented* programming. That's just stupid.  
 You're a relic. Something to help the old-timers  
 make the leap to Java.

Well, OK, every once in a while sure, it makes  
 sense to use a static, but let me tell you, abuse of  
 static variables (and methods) is the mark of an  
 immature OO programmer. A designer should be  
 thinking about *object* state, not *class* state.

Static methods are the worst things of all, because  
 it usually means the programmer is thinking  
 procedurally instead of about objects doing things  
 based on their unique object state.

Riiiiiight. Whatever you need to tell yourself.

**Static Variable**

What?

What do you mean *un-OO*?

I am NOT a global variable. There's no such  
 thing. I live in a class! That's pretty OO you know,  
 a CLASS. I'm not just sitting out there in space  
 somewhere; I'm a natural part of the state of an  
 object; the only difference is that I'm shared by all  
 instances of a class. Very efficient.

Alright just stop right there. THAT is definitely  
 not true. Some static variables are absolutely  
 crucial to a system. And even the ones that aren't  
 crucial sure are handy.

Why do you say that? And what's wrong with  
 static methods?

Sure, I know that objects should be the focus of  
 an OO design, but just because there are some  
 clueless programmers out there...don't throw the  
 baby out with the bytecode. There's a time and  
 place for statics, and when you need one, nothing  
 else beats it.



## BE the Compiler

The Java file on this page represents a complete program. Your job is to play compiler and determine whether this file will compile. If it won't compile, how would you fix it? When it runs, what would be its output?



```
class StaticSuper {  
    static {  
        System.out.println("super static block");  
    }  
  
    StaticSuper () {  
        System.out.println("super constructor");  
    }  
}  
  
public class StaticTests extends StaticSuper {  
    static int rand;  
  
    static {  
        rand = (int) (Math.random() * 6);  
        System.out.println("static block " + rand);  
    }  
  
    StaticTests() {  
        System.out.println("constructor");  
    }  
  
    public static void main(String[] args) {  
        System.out.println("in main");  
        StaticTests st = new StaticTests();  
    }  
}
```

Which of these is the output?

### Possible Output

```
File Edit Window Help Cling  
%java StaticTests  
static block 4  
in main  
super static block  
super constructor  
constructor
```

### Possible Output

```
File Edit Window Help Electricity  
%java StaticTests  
super static block  
static block 3  
in main  
super constructor  
constructor
```

→ Answers on page 308.



This chapter explored the wonderful, static world of Java. Your job is to decide whether each of the following statements is true or false.

## TRUE OR FALSE

1. To use the Math class, the first step is to make an instance of it.
2. You can mark a constructor with the **static** keyword.
3. Static methods don't have access to instance variable state of the "this" object.
4. It is good practice to call a static method using a reference variable.
5. Static variables could be used to count the instances of a class.
6. Constructors are called before static variables are initialized.
7. MAX\_SIZE would be a good name for a static final variable.
8. A static initializer block runs before a class's constructor runs.
9. If a class is marked final, all of its methods must be marked final.
10. A final method can be overridden only if its class is extended.
11. There is no wrapper class for boolean primitives.
12. A wrapper is used when you want to treat a primitive like an object.
13. The parseXxx methods always return a String.
14. Formatting classes (which are decoupled from I/O) are in the java.format package.

—————> Answers on page 308.

## Exercise Solutions

### Sharpen your pencil (from page 287)

1, 4, 5, and 6 are legal.

2 doesn't compile because the static method references a non-static instance variable.

3 doesn't compile because the instance variable is final but hasn't been initialized.

## BE the Compiler (from page 306)

```
StaticSuper () {  
    System.out.println(  
        "super constructor");  
}
```

StaticSuper is a constructor and must have ( ) in its signature. Notice that as the output below demonstrates, the static blocks for both classes run before either of the constructors run.

Note that this will be a randomly generated number from 0 to 5 inclusive.

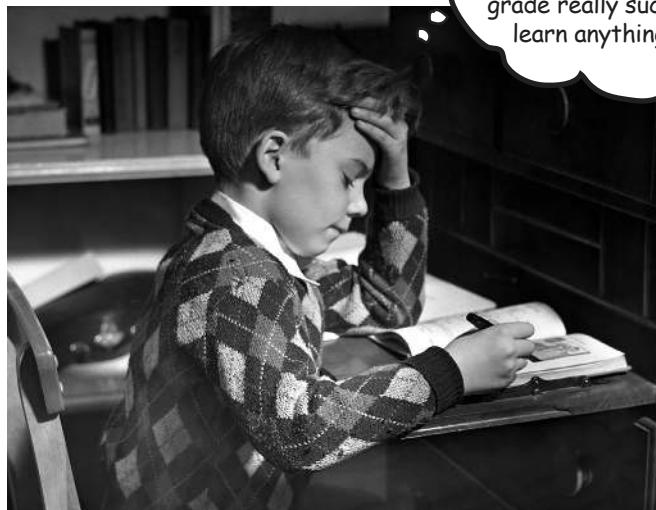
**Output**

```
File Edit Window Help Cling  
%java StaticTests  
super static block  
static block 3  
in main  
super constructor  
constructor
```

## TRUE OR FALSE (from page 307)

- |                                                                                       |              |
|---------------------------------------------------------------------------------------|--------------|
| 1. To use the Math class, the first step is to make an instance of it.                | <b>False</b> |
| 2. You can mark a constructor with the keyword "static."                              | <b>False</b> |
| 3. Static methods don't have access to an object's instance variables.                | <b>True</b>  |
| 4. It is good practice to call a static method using a reference variable.            | <b>False</b> |
| 5. Static variables could be used to count the instances of a class.                  | <b>True</b>  |
| 6. Constructors are called before static variables are initialized.                   | <b>False</b> |
| 7. MAX_SIZE would be a good name for a static final variable.                         | <b>True</b>  |
| 8. A static initializer block runs before a class's constructor runs.                 | <b>True</b>  |
| 9. If a class is marked final, all of its methods must be marked final.               | <b>False</b> |
| 10. A final method can be overridden only if its class is extended.                   | <b>False</b> |
| 11. There is no wrapper class for boolean primitives.                                 | <b>False</b> |
| 12. A wrapper is used when you want to treat a primitive like an object.              | <b>True</b>  |
| 13. The parseXxx methods always return a String.                                      | <b>False</b> |
| 14. Formatting classes (which are decoupled from I/O) are in the java.format package. | <b>False</b> |

# Data Structures



Sheesh...and all  
this time I could have  
just let Java put things in  
alphabetical order? Third  
grade really sucks. We never  
learn anything useful...

**Sorting is a snap in Java.** You have all the tools for collecting and manipulating your data without having to write your own sort algorithms (unless you're reading this right now sitting in your Computer Science 101 class, in which case, trust us—you are SO going to be writing sort code while the rest of us just call a method in the Java API). In this chapter, you're going to get a peek at when Java can save you some typing and figure out the types that you need.

The Java Collections Framework has a data structure that should work for virtually anything you'll ever need to do. Want to keep a list that you can easily keep adding to? Want to find something by name? Want to create a list that automatically takes out all the duplicates? Sort your co-workers by the number of times they've tried to speak with their mic muted on a video call? Sort your pets by number of tricks learned? It's all here...

# Tracking song popularity on your jukebox

Congratulations on your new job—managing the automated jukebox system at Lou’s Diner. There’s no Java inside the jukebox itself, but each time someone plays a song, the song data is appended to a simple text file.



Your job is to manage the data to track song popularity, generate reports, and manipulate the playlists. You’re not writing the entire app—some of the other software developers are involved as well, but you’re responsible for managing and sorting the data inside the Java app. And since Lou has a thing against databases, this is strictly an in-memory data collection. Another programmer will be writing the code to read the song data from a file and put the songs into a List. (In a few chapters you’ll learn how to read data from files, and write data to files.) All you’re going to get is a List with the song data the jukebox keeps adding to.

Let’s not wait for that other programmer to give us the actual file of songs; let’s create a small test program to provide us with some sample data we can work with. We’ve agreed with the other programmer that she’ll ultimately provide a Songs class with a `getSongs` method we’ll use to get the data. Armed with that information, we can write a small class to temporarily “stand in” for the actual code. Code that stands in for other code is often called “**mock**” code.

You’ll often want to write some temporary code that stands in for the real code that will come later. This is called “mocking.”

*We'll use this “mock” class to test our code.*

```

class MockSongs {
    public static List<String> getSongStrings() {
        List<String> songs = new ArrayList<>();
        songs.add("somersault");
        songs.add("cassidy");
        songs.add("$10");
        songs.add("havana");
        songs.add("Cassidy");
        songs.add("50 Ways");
        return songs;
    }
}

```

*We'll make this method static, because this class doesn't have any instance fields and doesn't need any.*

*This will be our list of six song titles to work with.*

Because `ArrayList` IS-A `List`, we can create an `ArrayList`, store it in a `List`, and return `List` from the method.

In the real world you’ll often see Java code that returns the interface type (`List`) and hides the implementation type (`ArrayList`).

# Your first job, sort the songs in alphabetical order

We'll start by creating code that reads in data from the mock Songs class and prints out what it got. Since an ArrayList's elements are placed in the order in which they were added, it's no surprise that the song titles are not yet alphabetized.

```
import java.util.*;

public class Jukebox1 {
    public static void main(String[] args) {
        new Jukebox1().go();
    }

    public void go() {
        List<String> songList = MockSongs.getSongStrings();
        System.out.println(songList);
    }
}

// Below is the "mock" code. A stand in for the actual
// I/O code that the other programmer will provide later

class MockSongs {
    public static List<String> getSongStrings() {
        List<String> songs = new ArrayList<>();
        songs.add("somersault");
        songs.add("cassidy");
        songs.add("$10");
        songs.add("havana");
        songs.add("Cassidy");
        songs.add("50 Ways");
        return songs;
    }
}
```

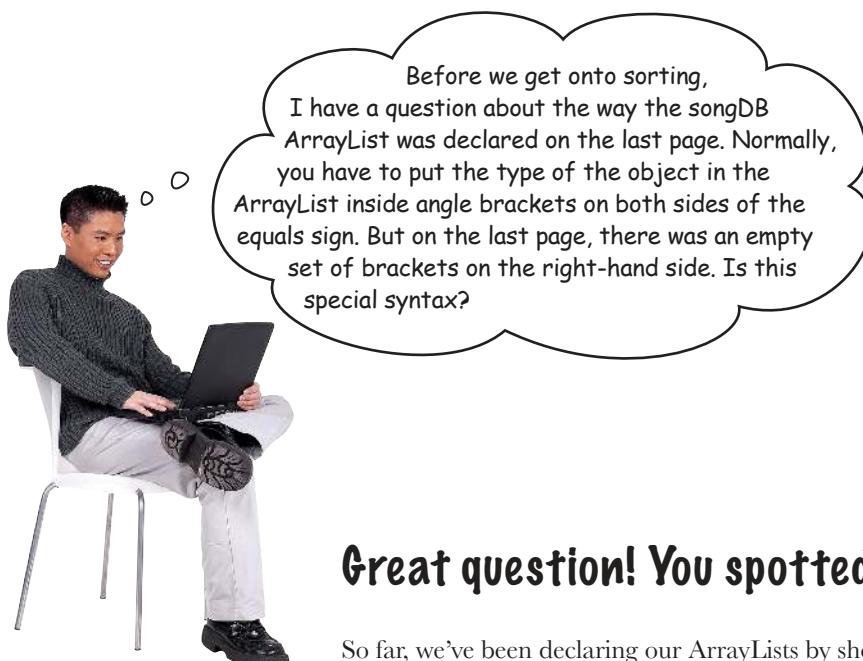
*We'll store the song titles in a List of Strings.*

*Then print the contents of the songList.*

*Nothing special here...just some sample data we can use to work on our sorting code.*

```
File Edit Window Help Dance
%java Jukebox1
[somersault, cassidy, $10, havana,
Cassidy, 50 Ways]
```

*This is definitely NOT alphabetical!*



## Great question! You spotted the diamond operator

So far, we've been declaring our ArrayLists by showing the element type twice:

```
ArrayList<String> songs = new ArrayList<String>();
```

Most of the time, we don't need to say the same thing twice. The compiler can tell from what you wrote on the left-hand side what you probably want on the right-hand side. It uses *type inference* to infer (work out) the type you need.

```
ArrayList<String> songs = new ArrayList<>();
```

No type needed

This syntax is called the diamond operator (because, well, it's diamond-shaped!) and was introduced in Java 7, so it's been around a while and is probably available in your version of Java.

Over time, Java has evolved to remove unnecessary code duplication from its syntax. If the compiler can figure out a type, you don't always need to write it out in full.

there are no  
**Dumb Questions**

**Q:** Should I be using the diamond operator all the time? Are there any downsides?

**A:** The diamond operator is “syntactic sugar,” which means it’s there to make our lives easier as we write (and read) code, but it doesn’t make any difference to the underlying byte code. So if you’re worried about whether using the diamond means something different to using the specific type, don’t worry! It’s basically the same thing.

However, sometimes you might choose to write out the full type. The main reason you might want to is to help people reading your code. For example, if the variable is declared a long way from where it’s initialized, you might want to use the type when you initialize it so you can see clearly what objects go into it.

```
ArrayList<String> songs;
// lots of code between these lines...
songs = new ArrayList<String>();
```

**Q:** Are there any other places the compiler can work out the types for me?

**A:** Yes! For example, the `var` keyword (“Local Variable Type Inference”), which we’ll talk about in Appendix B. And another important example, lambda expressions, which we will see later in this chapter.

**Q:** I saw you were creating an `ArrayList` but assigning it to a `List` reference, and that you created an `ArrayList` but returned a `List` from the method. Why not just use `ArrayList` everywhere?

**A:** One of the advantages of polymorphism is that code doesn’t need to know the specific implementation type of an object to work well with it. `List` is a well-known, well-understood interface (which we’ll see more of in this chapter). Code that is working with an `ArrayList` doesn’t usually need to know it’s an `ArrayList`. It could be a `LinkedList`. Or a specialized type of `List`. Code that’s working with the `List` only needs to know it can call `List` methods on it (like `add()`, `size()` etc). It’s usually safer to pass the interface type (i.e., `List`) around instead of the implementation. That way, other code can’t go rooting around inside your object in a way that was never intended.

It also means that should you ever want to change from an `ArrayList` to a `LinkedList`, or a `CopyOnWriteArrayList` (see Chapter 18, *Dealing with Concurrency Issues*) at a later date, you can without having to change all the places the `List` is used.

## Exploring the `java.util` API, List and Collections

We know that with an ArrayList, or any List, the elements are kept in the order in which they were added. So we're going to need to sort the elements in the song list. In this chapter, we'll be looking at some of the most important and commonly used collection classes in the `java.util` package, but for now, let's limit ourselves to two classes: `java.util.List` and `java.util.Collections`.

We've been using ArrayList for a while now. Because ArrayList IS-A List and because many of the methods we're familiar with on ArrayList come from List, we can comfortably transfer most of what we know about working with ArrayLists to List.

The Collections class is known as a “utility” class. It's a class that has a lot of handy methods for working with the various collection types.

### Excerpts from the API

`java.util.List`  
`sort(Comparator)`: Sorts this list according to the order induced by the specified **Comparator**.

`java.util.Collections`  
`sort(List)`: Sorts the specified list into ascending order, according to the **natural ordering** of its elements.  
`sort(List, Comparator)`: Sorts the specified list according to the order defined by the **Comparator**.

## In the “Real-World”™ there are lots of ways to sort

We don't always want our lists sorted alphabetically. We might want to sort clothes by size, or movies by how many five-star reviews they get. Java lets you sort the good old-fashioned way, alphabetically, and it also lets you create your own custom sorting approaches. Those references you see above to “Comparator” have to do with custom sorting, which we'll get to later this chapter. So for now, let's stick with “natural ordering” (alphabetical).

Since we know we have a List, it looks like we've found the perfect method, **Collections.sort()**.

# “Natural Ordering,” what Java means by alphabetical

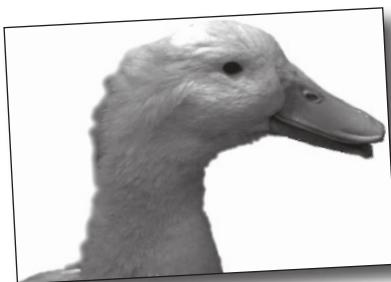
Lou wants you to sort songs “alphabetically,” but what exactly does that mean? The A–Z part is obvious, but how about lowercase versus uppercase letters? How about numbers and special characters? Well, this is another can-of-worms topic, but Java uses Unicode, and for many of us in “the West” that means that numbers sort before uppercase letters, uppercase letters sort before lowercase letters, and some special characters sort before numbers and some sort after numbers. That’s pretty clear, right? Ha! Well, the upshot is that, by default, sorting in Java happens in what’s called “natural order,” which is more or less alphabetical. Let’s take a look at what happens when we sort our list of songs:

```
public void go() {
    List<String> songList = MockSongs.getSongStrings();
    System.out.println(songList);
    Collections.sort(songList); ← Sort our song titles using
    System.out.println(songList);
}
}
```

```
File Edit Window Help Dance
%java Jukebox1
[somersault, cassidy, $10, havana, Cassidy, 50 Ways]
[$10, 50 Ways, Cassidy, cassidy, havana, somersault]
```

Our songs unsorted,  
in the order they  
were added.

Our songs sorted.  
Notice how the special  
characters, numbers,  
and uppercase letters  
got sorted.



oO

Just FYI, we ducks are  
very particular about how we  
get sorted.

# But now you need Song objects, not just simple Strings

Now your boss Lou wants actual Song class instances in the list, not just Strings, so that each Song can have more data. The new jukebox device outputs more information, so the actual song file will have *three* pieces of information for each song.

The Song class is really simple, with only one interesting feature—the overridden `toString()` method. Remember, the `toString()` method is defined in class `Object`, so every class in Java inherits the method. And since the `toString()` method is called on an object when it's printed (`System.out.println(anObject)`), you should override it to print something more readable than the default unique identifier code. When you print a list, the `toString()` method will be called on each object.

```
class SongV2 {
    private String title;
    private String artist;
    private int bpm;
}

SongV2(String title, String artist, int bpm) {
    this.title = title;
    this.artist = artist;
    this.bpm = bpm;
}

public String getTitle() {
    return title;
}

public String getArtist() {
    return artist;
}

public int.getBpm() {
    return bpm;
}

public String toString() {
    return title;
}
}
```

Three instance variables for the three song attributes in the file.

The variables are all set in the constructor whenever a new Song is created.

```
class MockSongs {
    public static List<String> getSongStrings() { ... }

    public static List<SongV2> getSongsV2() {
        List<SongV2> songs = new ArrayList<>();
        songs.add(new SongV2("somersault", "zero 7", 147));
        songs.add(new SongV2("cassidy", "grateful dead", 158));
        songs.add(new SongV2("$10", "hitchhiker", 140));

        songs.add(new SongV2("havana", "cabelllo", 105));
        songs.add(new SongV2("Cassidy", "grateful dead", 158));
        songs.add(new SongV2("50 ways", "simon", 102));
        return songs;
    }
}
```

The getter methods for the three attributes.

We override `toString()`, because when you do a `System.out.println(aSongObject)`, we want to see the title. When you do a `System.out.println(aListOfSongs)`, it calls the `toString()` method of EACH element in the list.

We made a new method in the `MockSongs` class to mock the new song data.

# Changing the Jukebox code to use Songs instead of Strings

Your code changes only a little. The big change is that the List will be of type <SongV2> instead of <String>.

```
import java.util.*;

public class Jukebox2 {
    public static void main(String[] args) {
        new Jukebox2().go();
    }

    public void go() {
        List<SongV2> songList = MockSongs.getSongsv2();
        System.out.println(songList);

        Collections.sort(songList);
        System.out.println(songList);
    }
}
```

*Change to a List of SongV2 objects instead of Strings.*

*Call the mock class to load song data into our List of songs.*

*And once again, call the sort method to sort the songs.*



## It won't compile!

He's right to be curious, something's wrong...the Collections class clearly shows there's a sort() method that takes a List. It *should* work.

### **But it doesn't!**

The compiler says it can't find a sort method that takes a List<SongV2>, so maybe it doesn't like a List of Song objects? It didn't mind a List<String>, so what's the important difference between Song and String? What's the difference that's making the compiler fail?

```
File Edit Window Help Bummer
%javac Jukebox2.java
Jukebox2.java:13: error: no suitable method found for
sort(List<SongV2>)
    Collections.sort(songList);
          ^
...
1 error
```

And of course you probably already asked yourself, “What would it be sorting *on*?” How would the sort method even *know* what made one Song greater or less than another Song? Obviously if you want the song’s *title* to be the value that determines how the songs are sorted, you’ll need some way to tell the sort method that it needs to use the title and not, say, the beats per minute.

We’ll get into all that a few pages from now, but first, let’s find out why the compiler won’t even let us pass a SongV2 List to the sort() method.



What is this??? I have no idea how to read the method declaration on this. It says that sort() takes a List<T>, but what is T? And what is that big thing before the return type?

## The sort() method declaration

```
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.

From the API docs (looking up the `java.util.Collections` class and scrolling to the `sort()` method), it looks like the `sort()` method is declared...*strangely*. Or at least different from anything we've seen so far.

That's because the `sort()` method (along with other things in the whole collection framework in Java) makes heavy use of *generics*. Any time you see something with angle brackets in Java source code or documentation, it means generics—a feature added in Java 5. So it looks like we'll have to learn how to interpret the documentation before we can figure out why we were able to sort `String` objects in a `List`, but not a `List` of `Song` objects.

## Generics means more type-safety

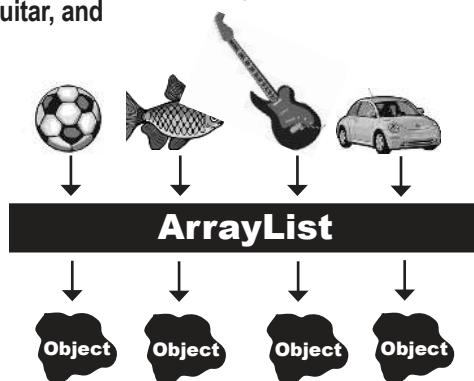
Although generics can be used in other ways, you'll often use generics to write type-safe collections. In other words, code that makes the compiler stop you from putting a Dog into a list of Ducks.

Without generics the compiler could not care less what you put into a collection, because all collection implementations hold type Object. You could put *anything* in any ArrayList without generics; it's like the ArrayList is declared as ArrayList<Object>.

### WITHOUT generics

Objects go IN as a reference to SoccerBall, Fish, Guitar, and Car objects

Before generics, there was no way to declare the type of an ArrayList, so its add() method took type Object.



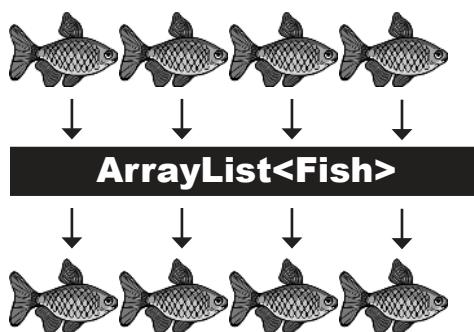
And come OUT as a reference of type Object

Without generics, the compiler would happily let you put a Pumpkin into an ArrayList that was supposed to hold only Cat objects.

With generics, you can create type-safe collections where more problems are caught at compile-time instead of runtime.

### WITH generics

Objects go IN as a reference to only Fish objects



And come out as a reference of type Fish

Now with generics, you can put only Fish objects in the ArrayList<Fish>, so the objects come out as Fish references. You don't have to worry about someone sticking a Volkswagen in there, or that what you get out won't really be castable to a Fish reference.

# Learning generics

Of the dozens of things you could learn about generics, there are really only three that matter to most programmers:

## ① Creating instances of generic classes (like ArrayList)

When you make an ArrayList, you have to tell it the type of objects you'll allow in the list, just as you do with plain old arrays.

## ② Declaring and assigning variables of generic types

How does polymorphism really work with generic types? If you have an ArrayList<Animal> reference variable, can you assign an ArrayList<Dog> to it? What about a List<Animal> reference? Can you assign an ArrayList<Animal> to it? You'll see...

## ③ Declaring (and invoking) methods that take generic types

If you have a method that has as a parameter, say, an ArrayList of Animal objects, what does that really mean? Can you also pass it an ArrayList of Dog objects? We'll look at some subtle and tricky polymorphism issues that are very different from the way you write methods that take plain old arrays.

(This is actually the same point as #2, but that shows you how important we think it is.)

## <sup>there are no</sup> Dumb Questions

**Q:** But don't I also need to learn how to create my OWN generic classes? What if I want to make a class type that lets people instantiating the class decide the type of things that class will use?

**A:** You probably won't do much of that. Think about it—the API designers made an entire library of collections classes covering most of the data structures you'd need, and the things that really need to be generic are collection classes or classes and methods that work on collections. There are some other cases too (like Optional, which we'll see in the next chapter). Generally, generic classes are classes that hold or operate on objects of some other type they don't know about.

Yes, it is possible that you might want to *create* generic classes, but that's pretty advanced, so we won't cover it here. (But you can figure it out from the things we *do* cover, anyway.)

```
new ArrayList<Song>()
```

```
List<Song> songList =
    new ArrayList<Song>()
```

```
void foo(List<Song> list)
```

```
x.foo(songList)
```

# Using generic CLASSES

Since ArrayList is one of our most-used generic classes, we'll start by looking at its documentation. The two key areas to look at in a generic class are:

1. The *class* declaration
2. The *method* declarations that let you add elements

## Understanding ArrayList documentation

(Or, what's the true meaning of "E"?)

```
The "E" is a placeholder for the  
REAL type you use when you  
declare and create an ArrayList.  
  
public class ArrayList<E> extends AbstractList<E> implements List<E> ... {  
  
    public boolean add(E o)  
    {  
        Here's the important part! Whatever "E" is  
        determines what kind of things you're allowed  
        to add to the ArrayList.  
    }  
    // more code  
}
```

The "E" represents the type used to create an instance of ArrayList. When you see an "E" in the ArrayList documentation, you can do a mental find/replace to exchange it for whatever <type> you use to instantiate ArrayList.

So, new ArrayList<Song> means that "E" becomes "Song" in any method or variable declaration that uses "E."

ArrayList is a subclass of AbstractList,  
so whatever type you specify for the  
type of the ArrayList, ArrayList is automatically used for the

The type (the value of <E>)  
becomes the type of the List  
interface as well.

Think of "E" as a stand-in for "the type of element you want this collection to hold and return." (E is for Element.)

# Using type parameters with ArrayList

**THIS code:**

```
List<String> thisList = new ArrayList<>
    public class ArrayList<E> extends AbstractList<E> ... {
        public boolean add(E o)
        // more code
    }
```

**Means ArrayList:**

**Is treated by the compiler as:**

```
public class ArrayList<String> extends AbstractList<String>... {

    public boolean add(String o)
    // more code
}
```

In other words, the “E” is replaced by the *real* type (also called the *type parameter*) that you use when you create the ArrayList. And that’s why the add() method for ArrayList won’t let you add anything except objects of a reference type that’s compatible with the type of “E.” So if you make an ArrayList<String>, the add() method suddenly becomes **add(String o)**. If you make the ArrayList of type **Dog**, suddenly the add() method becomes **add(Dog o)**.

there are no  
**Dumb Questions**

**Q:** Is “E” the only thing you can put there? Because the docs for sort used “T”....

**A:** You can use anything that’s a legal Java identifier. That means anything that you could use for a method or variable name will work as a type parameter. But you’ll often see single letter used. Another convention is to use “T” (“Type”) unless you’re specifically writing a collection class, where you’d use “E” to represent the “type of the Element the collection will hold.” Sometimes you’ll see “R” for “Return type.”

# Using generic METHODS

A generic *class* means that the *class declaration* includes a type parameter. A generic *method* means that the *method declaration* uses a type parameter in its signature.

You can use type parameters in a method in several different ways:

## ① Using a type parameter defined in the class declaration

```
public class ArrayList<E> extends AbstractList<E> ... {  
    public boolean add(E o) {  
        // You can use the "E" here ONLY because it's  
        // already been defined as part of the class.  
    }  
}
```

When you declare a type parameter for the class, you can simply use that type any place that you'd use a *real* class or interface type. The type declared in the method argument is essentially replaced with the type you use when you instantiate the class.

## ② Using a type parameter that was NOT defined in the class declaration

```
public <T extends Animal> void takeThing(ArrayList<T> list)
```

If the class itself doesn't use a type parameter, you can still specify one for a method, by declaring it in a really unusual (but available) space—*before the return type*. This method says that T can be “any type of Animal.”

Here we can use <T> because we declared "T" at the start of the method declaration.



## Here's where it gets weird...

*This:*

```
public <T extends Animal> void takeThing(ArrayList<T> list)
```

*Is NOT the same as this:*

```
public void takeThing(ArrayList<Animal> list)
```

Both are legal, but they're *different!*

The first one, where `<T extends Animal>` is part of the method declaration, means that any `ArrayList` declared of a type that is `Animal`, or one of `Animal`'s subtypes (like `Dog` or `Cat`), is legal. So you could invoke the top method using an `ArrayList<Dog>`, `ArrayList<Cat>`, or `ArrayList<Animal>`.

But...the one on the bottom, where the method argument is `(ArrayList<Animal> list)` means that *only* an `ArrayList<Animal>` is legal. In other words, while the first version takes an `ArrayList` of any type that is a type of `Animal` (`Animal`, `Dog`, `Cat`, etc.), the second version takes *only* an `ArrayList` of type `Animal`. Not `ArrayList<Dog>` or `ArrayList<Cat>`, but only `ArrayList<Animal>`.

And yes, it does appear to violate the point of polymorphism, but it will become clear when we revisit this in detail at the end of the chapter. For now, remember that we're only looking at this because we're still trying to figure out how to `sort()` that `SongList`, and that led us into looking at the API for the `sort()` method, which had this strange generic type declaration.

***For now, all you need to know is that the syntax of the top version is legal and that it means you can pass in a `ArrayList` object instantiated as `Animal` or any `Animal` subtype.***

And now back to our `sort()` method...



```
import java.util.*;

public class Jukebox2 {
    public static void main(String[] args) {
        new Jukebox2().go();
    }

    public void go() {

        List<SongV2> songList = MockSongs.getSongsV2();
        System.out.println(songList);

        Collections.sort(songList);
        System.out.println(songList);
    }
}
```

This is where it breaks! It worked fine when passed in a List<String>, but as soon as we tried to sort a List<SongV2>, it failed.

# Revisiting the sort() method

So here we are, trying to read the sort() method docs to find out why it was OK to sort a list of Strings, but not a list of Song objects. And it looks like the answer is...

```
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.
```

**The sort() method can take only lists of Comparable objects.**

**Song is NOT a subtype of Comparable, so you cannot sort() the list of Songs.**

**At least not yet...**

```
public static <T extends Comparable<? super T>> void sort(List<T> list)
```

This says "Whatever 'T' is must be of type Comparable."

(Ignore this part for now. But if you can't, it just means that the type parameter for Comparable must be of type T or one of T's supertypes.)

You can pass in only a List (or subtype of list, like ArrayList) that uses a parameterized type that "extends Comparable."

Um...I just checked the docs for String, and String doesn't EXTEND Comparable—it IMPLEMENTS it. Comparable is an interface. So it's nonsense to say <T extends Comparable>.



```
public final class String
    implements java.io.Serializable, Comparable<String>, CharSequence {
```

Great point, and one that deserves a full answer! Turn the page...

## In generics, “extends” means “extends or implements”

The Java engineers had to give you a way to put a constraint on a parameterized type so that you can restrict it to, say, only subclasses of Animal. But you also need to constrain a type to allow only classes that implement a particular interface. So here’s a situation where we need one kind of syntax to work for both situations—inheritance and implementation. In other words, that works for both *extends* and *implements*.

And the winning word was...*extends*. But it really means “IS-A” and works regardless of whether the type on the right is an interface or a class.

Comparable is an interface, so this  
REALLY reads, “T must be a type that  
implements the Comparable interface.”

```
public static <T extends Comparable<? super T>> void sort(List<T> list)
```

It doesn't matter whether the thing on the right is  
a class or interface...you still say "extends."

there are no  
Dumb Questions

**Q:** Why didn’t they just make a new keyword, “is”?

**A:** Adding a new keyword to the language is a REALLY big deal because it risks breaking Java code you wrote in an earlier version. Think about it—you might be using a variable “is” (which we do use in this book to represent input streams). And since you’re not allowed to use keywords as identifiers in your code, that means any earlier code that used the keyword *before* it was a reserved word, would break. So whenever there’s a chance for the language engineers to reuse an existing keyword, as they did here with “extends,” they’ll usually choose that. But sometimes they don’t have a choice.

In recent years, new “keyword-like” terms have been added to the language, without actually making it a keyword that would trample all over your earlier code. For example, the identifier **var**, which we’ll talk about in Appendix B, is a *reserved type name*, **not** a keyword. This means any existing code that used var (for example, as a variable name) will not break if it’s compiled with a version of Java that supports **var**.

In generics, the keyword  
“extends” really means “IS-A”  
and works for BOTH classes  
and interfaces.

# Finally we know what's wrong... The Song class needs to implement Comparable

We can pass the `ArrayList<Song>` to the `sort()` method only if the `Song` class implements `Comparable`, since that's the way the `sort()` method was declared. A quick check of the API docs shows the `Comparable` interface is really simple, with only one method to implement:

`java.lang.Comparable`

```
public interface Comparable<T> {
    int compareTo(T o);
}
```

**The big question is: what makes one song less than, equal to, or greater than another song?**

**You can't implement the Comparable interface until you make that decision.**

And the method documentation for `compareTo()` says:

**Returns:**  
 a negative integer if this object is less than the specified object;  
 a zero if they're equal;  
 a positive integer if this object is greater than the specified object.

It looks like the `compareTo()` method will be called on one `Song` object, passing that `Song` a reference to a different `Song`. The `Song` running the `compareTo()` method has to figure out if the `Song` it was passed should be sorted higher, lower, or the same in the list.

Your big job now is to decide what makes one song greater than another, and then implement the `compareTo()` method to reflect that. A negative number (any negative number) means the `Song` you were passed is greater than the `Song` running the method. Returning a positive number says that the `Song` running the method is greater than the `Song` passed to the `compareTo()` method. Returning zero means the `Songs` are equal (at least for the purpose of sorting...it doesn't necessarily mean they're the same object). You might, for example, have two `Songs` by different artists with the same title.

(That brings up a whole different can of worms we'll look at later...)

 **Sharpen your pencil**

Write in your idea and pseudocode (or better, REAL code) for implementing the `compareTo()` method in a way that will `sort()` the `Song` objects by title.

Hint: if you're on the right track, it should take less than three lines of code!

→ **Yours to solve.**

## The new, improved, comparable Song class

We decided we want to sort by title, so we implement the `compareTo()` method to compare the title of the `Song` passed to the method against the title of the song on which the `compareTo()` method was invoked. In other words, the song running the method has to decide how its title compares to the title of the method parameter.

Hmmm...we know that the `String` class must know about alphabetical order, because the `sort()` method worked on a list of `Strings`. We know `String` has a `compareTo()` method, so why not just call it? That way, we can simply let one title `String` compare itself to another, and we don't have to write the comparing/alphabetizing algorithm!

Usually these match...we're specifying the type that the implementing class can be compared against.

This means that `SongV3` objects can be compared to other `SongV3` objects, for the purpose of sorting.

The `sort()` method sends a `Song` to `compareTo()` to see how that `Song` compares to the `Song` on which the method was invoked.

Simple! We just pass the work on to the title `String` objects, since we know `String` objects have a `compareTo()` method.

```

class SongV3 implements Comparable<SongV3> {
    private String title;
    private String artist;
    private int bpm;

    public int compareTo(SongV3 s) {
        return title.compareTo(s.getTitle());
    }

    SongV3(String title, String artist, int bpm) {
        this.title = title;
        this.artist = artist;
        this.bpm = bpm;
    }

    public String getTitle() {
        return title;
    }

    public String getArtist() {
        return artist;
    }

    public int getBpm() {
        return bpm;
    }

    public String toString() {
        return title;
    }
}

```

This time it worked. It prints the list and then calls `sort()`, which puts the songs in alphabetical order by title.

```
%java Jukebox3
[somersault, cassidy, $10, havana, Cassidy, 50
ways]
[$10, 50 ways, Cassidy, cassidy, havana, somer-
sault]
```

## We can sort the list, but...

There's a new problem—Lou wants two different views of the song list, one by song title and one by artist!

But when you make a collection element comparable (by having it implement Comparable), you get only one chance to implement the compareTo() method. So what can you do?

The horrible way would be to use a flag variable in the Song class and then do an *if* test in compareTo() and give a different result depending on whether the flag is set to use title or artist for the comparison.

But that's an awful and brittle solution, and there's something much better. Something built into the API for just this purpose—when you want to sort the same thing in more than one way.

**Look at API documentation again. There's a second sort() method on Collections—and it takes a Comparator. There's also a sort method on List that takes a Comparator.**

That's not good enough.  
Sometimes I want it to sort  
by artist instead of title.



### Excerpts from the API

#### `java.util.Collections`

`sort(List)`: Sorts the specified list into ascending order, according to the **natural ordering** of its elements.

`sort(List, Comparator)`: Sorts the specified list according to the order induced by the specified **Comparator**.

#### `java.util.List`

`sort(Comparator)`: Sorts this list according to the order induced by the specified **Comparator**.

The `sort()` method on  
Collections is overloaded  
to take something called  
a **Comparator**.

There's also a  
`sort()` on List,  
which takes a  
**Comparator**.

Note to self: figure out how to get/make  
a **Comparator** that can compare and order  
the songs by artist instead of title.

# Using a custom Comparator

A **Comparable** element in a list can compare *itself* to another of its own type in only one way, using its `compareTo()` method. But a **Comparator** is external to the element type you're comparing—it's a separate class. So you can make as many of these as you like! Want to compare songs by artist? Make an `ArtistComparator`. Sort by beats per minute? Make a `BpmComparator`.

Then all you need to do is call a `sort()` method that takes a `Comparator` (`Collections.sort` or `List.sort`), which will use this `Comparator` to put things in order.

The `sort()` method that takes a `Comparator` will use the `Comparator` instead of the element's own `compareTo()` method when it puts the elements in order. In other words, if your `sort()` method gets a `Comparator`, it won't even *call* the `compareTo()` method of the elements in the list. The `sort()` method will instead invoke the **compare()** method on the `Comparator`.

To summarize, the rules are:

- ▶ Invoking the `Collections.sort(List list)` method means the list element's `compareTo()` method determines the order. The elements in the list **MUST** implement the **Comparable** interface.
- ▶ Invoking `List.sort(Comparator c)` or `Collections.sort(List list, Comparator c)` means the `Comparator`'s `compare()` method will be used. That means the elements in the list do **NOT** need to implement the `Comparable` interface, but if they do, the list element's `compareTo()` method will **NOT** be called.

## java.util.Comparator

```
public interface Comparator<T> {
    int compare(T o1, T o2);
}
```

If you pass a `Comparator` to the `sort()` method, the sort order is determined by the `Comparator`.

If you don't pass a `Comparator` and the element is `Comparable`, the sort order is determined by the element's `compareTo()` method.

## there are no Dumb Questions

**Q:** Why are there two sort methods that take a comparator on two different classes? Which sort method should I use?

**A:** Both methods that take a comparator, `Collections.sort(List, Comparator)` and `List.sort(Comparator)`, do the same thing; you can use either and expect exactly the same results.

`List.sort` was introduced in Java 8, so older code *must* use `Collections.sort(List, Comparator)`.

We use `List.sort` because it's a bit shorter, and generally you already have the list you want to sort, so it makes sense to call the `sort` method on that list.

# Updating the Jukebox to use a Comparator

We're going to update the Jukebox code in three ways:

1. Create a separate class that implements Comparator (and thus the `compare()` method that does the work previously done by `compareTo()`).
2. Make an instance of the Comparator class.
3. Call the List.sort() method, giving it the instance of the Comparator class.

```
import java.util.*;

public class Jukebox4 {
    public static void main(String[] args) {
        new Jukebox4().go();
    }

    public void go() {
        List<SongV3> songList = MockSongs.getSongsV3();
        System.out.println(songList);

        Collections.sort(songList);
        System.out.println(songList);

        ArtistCompare artistCompare = new ArtistCompare();
        songList.sort(artistCompare);
        System.out.println(songList);
    }
}

class ArtistCompare implements Comparator<SongV3> {
    public int compare(SongV3 one, SongV3 two) {
        return one.getArtist().compareTo(two.getArtist());
    }
} This is a String (the artist)
```

*Make an instance of the Comparator class.*

*Invoke sort() on our list, passing it a reference to the new custom Comparator object.*

*We're letting the String variables (for artist) do the actual comparison, since Strings already know how to alphabetize themselves.*

```
File Edit Window Help Ambient
% java Jukebox4
[somersault, cassidy, $10, havana, Cassidy, 50 ways]
[$10, 50 ways, Cassidy, cassidy, havana, somersault]
[havana, Cassidy, cassidy, $10, 50 ways, somersault]
```

*Unsorted songList*

*Sorted by title (using the Song's compareTo method)*

*Sorted by artist name (using ArtistComparator)*

**exercise: sharpen your pencil**



## Fill-in-the-blanks

For each of the questions below, fill in the blank with one of the words from the “possible answers” list, to correctly answer the question.

### Possible Answers:

Comparator,

Comparable,

compareTo( ),

compare( ),

yes,

no

Given the following compilable statement:

```
Collections.sort(myArrayList) ;
```

1. What must the class of the objects stored in myArrayList implement? \_\_\_\_\_
2. What method must the class of the objects stored in myArrayList implement? \_\_\_\_\_
3. Can the class of the objects stored in myArrayList implement both Comparator AND Comparable? \_\_\_\_\_

Given the following compilable statements (they both do the same thing):

```
Collections.sort(myArrayList, myCompare) ;  
myArrayList.sort(myCompare) ;
```

4. Can the class of the objects stored in myArrayList implement Comparable? \_\_\_\_\_
5. Can the class of the objects stored in myArrayList implement Comparator? \_\_\_\_\_
6. Must the class of the objects stored in myArrayList implement Comparable? \_\_\_\_\_
7. Must the class of the objects stored in myArrayList implement Comparator? \_\_\_\_\_
8. What must the class of the myCompare object implement? \_\_\_\_\_
9. What method must the class of the myCompare object implement? \_\_\_\_\_

—————> Answers on page 364.

# But wait! We're sorting in two different ways!

Now we're able to sort the song list two ways:

1. Using Collections.sort(songList), because Song implements **Comparable**
2. Using songLists.sort(artistCompare) because the ArtistCompare class implements **Comparator**

While our new code allows us to sort songs by title and by artist, it is reminiscent of Frankenstein's monster, cobbled together bit by bit.

```
public void go() {
    List<SongV3> songList = MockSongs.getSongsV3();
    System.out.println(songList);
    Collections.sort(songList);           ← This uses Comparable to sort
    System.out.println(songList);

    ArtistCompare artistCompare = new ArtistCompare();
    songList.sort(artistCompare);          ← This uses a custom
    System.out.println(songList);          Comparator to sort
}
```

A better approach would be to handle all of the sorting definitions in classes that implement Comparator.

there are no  
Dumb Questions

**Q:** So does this mean that if you have a class that doesn't implement Comparable, and you don't have the source code, you could still put the things in order by creating a Comparator?

**A:** That's right. The other option (if it's possible) would be to subclass the element and make the subclass implement Comparable.

**Q:** But why doesn't every class implement Comparable?

**A:** Do you really believe that *everything* can be ordered? If you have element types that just don't lend themselves to any kind of natural ordering, then you'd be misleading other programmers if you implement Comparable. And there's no problem if you *don't* implement Comparable, since a programmer can compare anything in any way that they choose using their own custom Comparator.

## Sorting using only Comparators

Having Song implement Comparable and creating a custom Comparator for sorting by Artist absolutely works, but it's confusing to rely on two different mechanisms for our sort. It's much clearer if our code uses the same technique to sort, regardless of how Lou wants his songs sorted. The code below has been updated to use Comparators for sorting by both Title and Artist; the new code is in bold.

```
public class Jukebox5 {
    public static void main(String[] args) {
        new Jukebox5().go();
    }

    public void go() {
        List<SongV3> songList = MockSongs.getSongsV3();
        System.out.println(songList);

        TitleCompare titleCompare = new TitleCompare();
        songList.sort(titleCompare);
        System.out.println(songList);

        ArtistCompare artistCompare = new ArtistCompare();
        songList.sort(artistCompare);
        System.out.println(songList);
    }
}

class TitleCompare implements Comparator<SongV3> {
    public int compare(SongV3 one, SongV3 two) {
        return one.getTitle().compareTo(two.getTitle());
    }
}

class ArtistCompare implements Comparator<SongV3> {
    public int compare(SongV3 one, SongV3 two) {
        return one.getArtist().compareTo(two.getArtist());
    }
}

// more specialized Comparator classes could go here, ○
// e.g. BpmCompare
```

Make an instance of the Comparator class and use the sort() method on List.

This is the new class that implements Comparator.

D  
That's an awful lot of code for sorting our songs in just two different orders. Isn't there a better way?



## Just the code that matters

The Jukebox class does have a lot of code that's needed for sorting. Let's zoom in on one of the Comparator classes we wrote for Lou. The first thing to notice is that all we really want to sort our collection is the one line of code in the middle of the class. The rest of the code is just the long-winded syntax that's necessary to let the compiler know what type of class this is and which method it implements.



### Code Up Close

```
class TitleCompare implements Comparator<Song> {
    public int compare(Song one, Song two) {
        return one.getTitle().compareTo(two.getTitle());
    }
}
```

The one line of code that's doing all the work

There's more than one way to declare small pieces of functionality like this. One approach is inner classes, which we'll look at in a later chapter. You can even declare the inner class right where you use it (instead of at the end of your class file); this is sometimes called an "argument-defined anonymous inner class." Sounds fun already:

```
songList.sort(new Comparator<SongV3>() {
    public int compare(SongV3 one, SongV3 two) {
        return one.getTitle().compareTo(two.getTitle());
    }
});
```

While this lets us declare the sorting logic in exactly the location we need it (where we call the sort method, instead of in a separate class), there's still a lot of code there for saying "sort by title please".



### Relax

We're not going to learn how to write "argument-defined anonymous inner classes"!

We just wanted you to see this example, in case you stumble across it in Real Code.

the compiler already knows

## What do we REALLY need in order to sort?

```
public class Jukebox5 {  
    public void go() {  
        ① List<SongV3> songList = MockSongs.getSongsV3();  
        ...  
        TitleCompare titleCompare = new TitleCompare();  
        ② songList.sort(titleCompare);  
        ...  
    }  
  
    class TitleCompare implements Comparator<SongV3> {  
        public int compare(SongV3 one, SongV3 two) {  
            return one.getTitle().compareTo(two.getTitle());  
        }  
    }  
}
```

The compiler knows the List contains SongV3 objects.

The compiler understands that sort() expects a Comparator for SongV3 objects.

Let's take a look at the API documentation for the sort method on List:

```
default void sort(Comparator<? super E> c)
```

Sorts this list according to the order induced by the specified Comparator.

If we were to explain out loud the following line of code:

```
songList.sort(titleCompare);
```

We could say:

**“Call the sort method on the list of songs ① and pass it a reference to a Comparator object, which is designed specifically to sort Song objects ②.”**

If we're honest, we could say all that without even looking at the TitleCompare class. We can work it all out just by looking at the documentation for sort and the type of the List that we're sorting!



### Brain Barbell

Do you think the compiler cares about the name “TitleCompare”? If the class was called “FooBar” instead, would the code still work?



Wouldn't it be wonderful  
if the code that described HOW  
you want to sort your Songs  
wasn't so far away from the  
sort method? And wouldn't it be  
great if you didn't have to write a  
bunch of code the compiler could  
probably figure out on its own...

# Enter lambdas! Leveraging what the compiler can infer

We could write a whole bunch of code to say how to sort a list (like we have been doing)...

```
...  
songList.sort(titleCompare);
```

The compiler cares  
not a whit what you  
call the class.

This is just a reference to an object that  
implements Comparator. Its name doesn't  
matter to the compiler.

```
class TitleCompare implements Comparator<Song> {
```

Yup, the compiler  
knows what this  
method should  
look like.

Doh! The compiler can infer this from  
the sort() docs.

```
    public int compare(Song one, Song two) {
```

The compiler can figure  
out that the two objects  
have to be Song objects  
since songList is a List of  
Song objects.

```
        return one.getTitle().compareTo(two.getTitle());
```

```
}
```

```
...  
}
```

This is ALL THE COMPILER NEEDS.  
Just tell it HOW to do the sort!

Or, we could use a lambda...

```
songList.sort((one, two) -> one.getTitle().compareTo(two.getTitle()));
```

These are Song one and  
Song two, the parameters  
to the compare method.



## Brain Barbell

What do you think would happen if Comparator  
needed you to implement more than one method?  
How much could the compiler fill in for you?

# Where did all that code go?

To answer this question, let's take a look at the API documentation for the Comparator interface.

## Method Summary

| Modifier and Type | Method                           | Description                                                        |
|-------------------|----------------------------------|--------------------------------------------------------------------|
| int               | <code>compare(T o1, T o2)</code> | Compares its two arguments for order.                              |
| boolean           | <code>equals(Object obj)</code>  | Indicates whether some other object is "equal to" this comparator. |

Because equals has been implemented by Object, if we create a custom comparator, we know we only *need* to implement the compare method.

We also know exactly the shape of that method—it has to return an int, and it takes two arguments of type T (remember generics?). Our lambda expression implements the compare() method, without having to declare the class or the method, only the details of what goes into the body of the compare() method.

Remember from Chapter 8 that every class and interface inherits methods from class Object, and that equals() is implemented in class Object.



## Some interfaces have only ONE method to implement

With interfaces like Comparator, we only have to implement a **single abstract method**, SAM for short. These interfaces are so important that they have several special names:

## SAM Interfaces a.k.a. Functional Interfaces

If an interface has only one method that needs to be implemented, that interface can be implemented as a **lambda expression**. You don't need to create a whole class to implement the interface; the compiler knows what the class and method would look like. What the compiler *doesn't* know is the logic that goes *inside* that method.



We'll look at lambda expressions and functional interfaces in much more detail in the next chapter. For now, back to Lou's diner.

# Updating the Jukebox code with lambdas

```

import java.util.*;

public class Jukebox6 {
    public static void main(String[] args) {
        new Jukebox6().go();
    }

    public void go() {
        List<SongV3> songList = MockSongs.getSongsV3();
        System.out.println(songList);

        songList.sort((one, two) -> one.getTitle().compareTo(two.getTitle()));
        System.out.println(songList);
    }
}

```

Here's our lambda expression in action—no need to create a custom Comparator class; just put the sorting logic right inside sort method call.

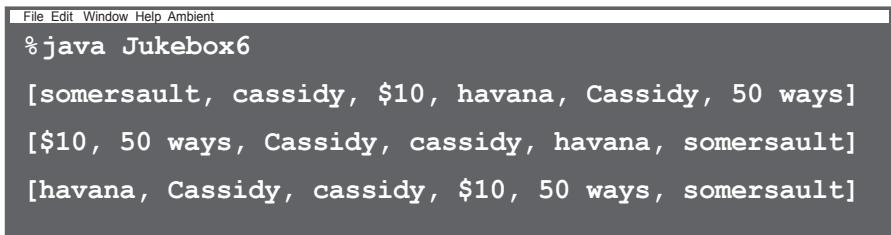
You can tell what the list will be sorted by, just by looking at the field used in the lambda.

```

songList.sort((one, two) -> one.getArtist().compareTo(two.getArtist()));
System.out.println(songList);

```

The output is exactly the same as when we used Comparator classes, but our code was much shorter.



```

File Edit Window Help Ambient
%java Jukebox6
[somersault, cassidy, $10, havana, Cassidy, 50 ways]
[$10, 50 ways, Cassidy, cassidy, havana, somersault]
[havana, Cassidy, cassidy, $10, 50 ways, somersault]

```



## How could you sort the songs differently?

Write lambda expressions to sort the songs in these ways (the answers are at the end of the chapter):

- Sort by BPM
- Sort by title in descending order

→ Answers on  
page 366.



## Reverse Engineer

Assume this code exists in a single file. Your job is to fill in the blanks so the program will create the output shown.

```

import _____;

public class SortMountains {
    public static void main(String [] args) {
        new SortMountains().go();
    }

    public void go() {
        List_____ mountains = new ArrayList<>();
        mountains.add(new Mountain("Longs", 14255));
        mountains.add(new Mountain("Elbert", 14433));
        mountains.add(new Mountain("Maroon", 14156));
        mountains.add(new Mountain("Castle", 14265));
        System.out.println("as entered:\n" + mountains);

        mountains._____ (____->_____);
        System.out.println("by name:\n" + mountains);

        _____._____(____->_____);
        System.out.println("by height:\n" + mountains);
    }
}

class Mountain {
    _____;
    _____;

    _____ {
        _____;
        _____;
    }
    _____ {
        _____;
    }
}
  
```

### Output:

```

File Edit Window Help ThisOne'sForBob
%java SortMountains
as entered:
[Longs 14255, Elbert 14433, Maroon 14156, Castle 14265]
by name:
[Castle 14265, Elbert 14433, Longs 14255, Maroon 14156]
by height:
[Elbert 14433, Castle 14265, Longs 14255, Maroon 14156]
  
```

→ Answers on page 365.

## Uh-oh. The sorting all works, but now we have duplicates...

The sorting works great; now we know how to sort on both *title* and *artist*. But there's a new problem we didn't notice with a test sample of the jukebox songs—***the sorted list contains duplicates***.

Unlike the mock code, Lou's real jukebox application appears to just keep writing to the file regardless of whether the same song has already been played (and thus written) to the text file. The *SongListMore.txt* jukebox text file is an example. It's a complete record of every song that was played, and might contain the same song multiple times.

```

File Edit Window Help TooManyNotes
%java Jukebox7

[somersault: zero 7, cassidy: grateful dead, $10: hitchhiker,
havana: cabello, $10: hitchhiker, cassidy: grateful dead, 50
ways: simon]

[$10: hitchhiker, $10: hitchhiker, 50 ways: simon, cassidy:
grateful dead, cassidy: grateful dead, havana: cabello,
somersault: zero 7]

[havana: cabello, cassidy: grateful dead, cassidy: grateful
dead, $10: hitchhiker, $10: hitchhiker, 50 ways: simon,
somersault: zero 7]

```

Before sorting

After sorting by song title

After sorting by artist name

Note that we changed the Song's *toString* to output the title and the artist.

This is what the actual song data file looks like. →

### SongListMore.txt

```

somersault, zero 7, 147
cassidy, grateful dead, 158
$10, hitchhiker, 140
havana, cabello, 105
$10, hitchhiker, 140
cassidy, grateful dead, 158
50 ways, simon, 102

```

The *SongListMore* text file now has duplicates in it, because the jukebox machine is writing every song played, in order.

To get the output above, we wrote a *MockMoreSongs* class with a *getSongs()* method that returned a *List* that has all the same entries as this text file.

# We need a Set instead of a List

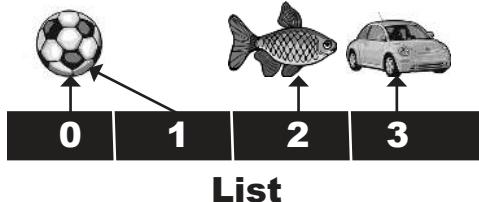
From the Collection API, we find three main interfaces, **List**, **Set**, and **Map**.  
ArrayList is a **List**, but it looks like **Set** is exactly what we need.

## ► LIST - when sequence matters

Collections that know about *index position*.

Lists know where something is in the list. You can have more than one element referencing the same object.

Duplicates OK.

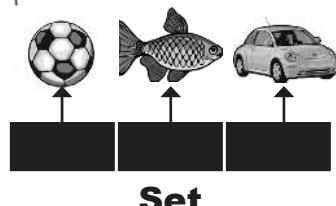


## ► SET - when uniqueness matters

Collections that *do not allow duplicates*.

Sets know whether something is already in the collection. You can never have more than one element referencing the same object (or more than one element referencing two objects that are considered equal—we'll look at what object equality means in a moment).

NO duplicates.

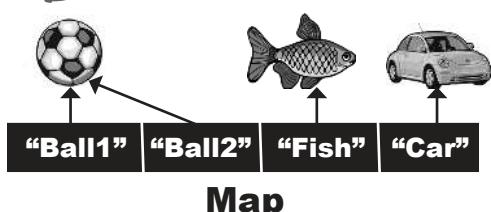


## ► MAP - when *finding something by key* matters

Collections that use *key-value pairs*.

Maps know the value associated with a given key. You can have two keys that reference the same value, but you cannot have duplicate keys. A key can be any object.

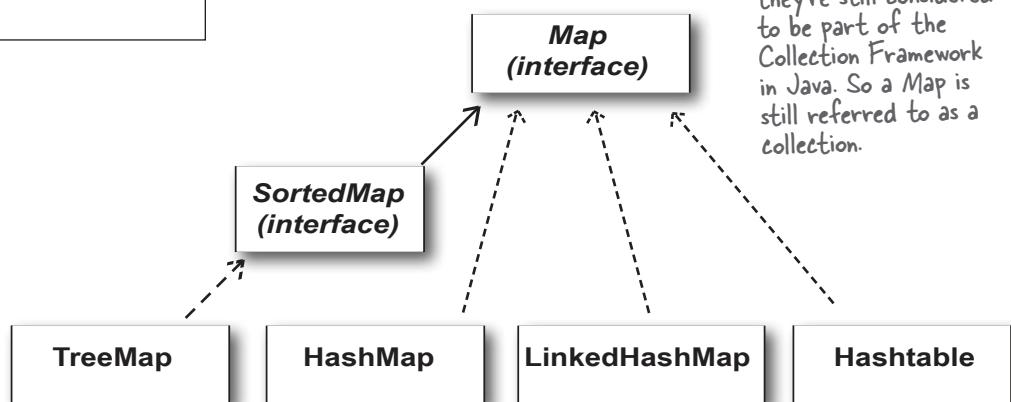
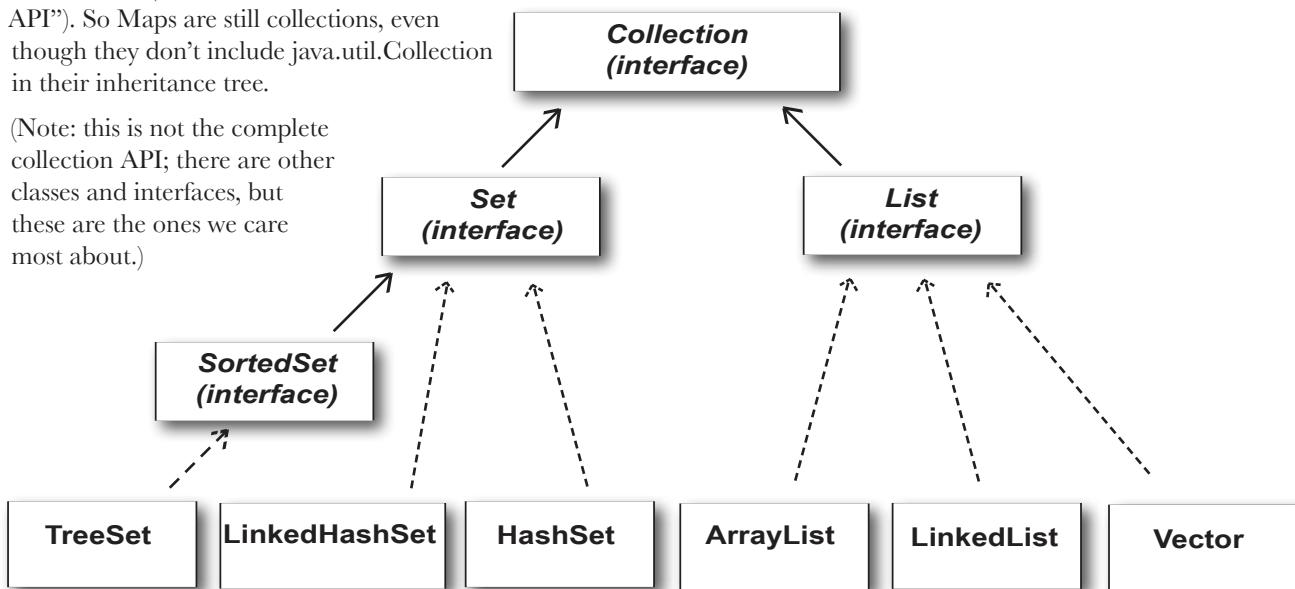
Duplicate values OK, but NO duplicate keys.



# The Collection API (part of it)

Notice that the Map interface doesn't actually extend the Collection interface, but Map is still considered part of the “Collection Framework” (also known as the “Collection API”). So Maps are still collections, even though they don't include `java.util.Collection` in their inheritance tree.

(Note: this is not the complete collection API; there are other classes and interfaces, but these are the ones we care most about.)



# Using a HashSet instead of ArrayList

We updated the Jukebox code to put the songs in a HashSet to try to eliminate our duplicate songs. (Note: we left out some of the Jukebox code, but you can copy it from earlier versions.)

```
import java.util.*;

public class Jukebox8 {
    public static void main(String[] args) {
        new Jukebox8().go();
    }

    public void go() {
        List<SongV3> songList = MockMoreSongs.getSongsV3();
        System.out.println(songList);

        songList.sort((one, two) -> one.getTitle().compareTo(two.getTitle()));
        System.out.println(songList);

        Set<SongV3> songSet = new HashSet<>(songList);
        System.out.println(songSet);
    }
}
```

We want the Set to hold SongV3 objects. HashSet IS-A Set, so we can store the HashSet in this Set variable.

We created a MockMoreSongs class to return a List of SongV3 objects that contain the same values as SongListMore.txt.

↗ HashSet has a constructor that takes a Collection, and it will create a set with all the items from that collection.

```
File Edit Window Help GetBetterMusic
%java Jukebox8
[somersault, cassidy, $10, havana, $10, cassidy, 50 ways]
[$10, $10, 50 ways, cassidy, cassidy, havana, somersault]
[$10, 50 ways, havana, cassidy, $10, cassidy, somersault]
```

The Set didn't help!!

We still have all the duplicates!

(And it lost its sort order when we put the list into a HashSet, but we'll worry about that one later...)

After putting it into a HashSet and printing the HashSet (we didn't call sort() again).

# What makes two objects equal?

To figure out why using a Set didn't remove the duplicates, we have to ask—what makes two Song references duplicates? They must be considered ***equal***. Is it simply two references to the very same object, or is it two separate objects that both have the same *title*?

This brings up a key issue: *reference* equality vs. *object* equality.

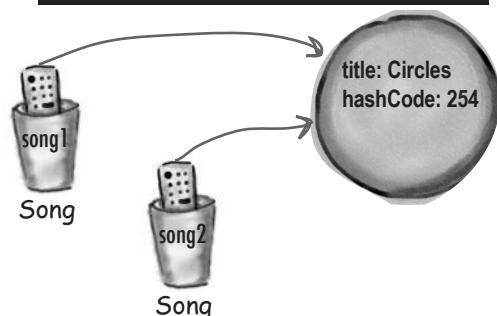
## ► Reference equality

Two references, one object on the heap.

Two references that refer to the same object on the heap are equal. Period. If you call the `hashCode()` method on both references, you'll get the same result. If you don't override the `hashCode()` method, the default behavior (remember, you inherited this from class `Object`) is that each object will get a unique number (most versions of Java assign a hashcode based on the object's memory address on the heap, so no two objects will have the same hashcode).

If you want to know if two *references* are really referring to the same object, use the `==` operator, which (remember) compares the bits in the variables. If both references point to the same object, the bits will be identical.

If two objects `foo` and `bar` are equal, `foo.equals(bar)` and `bar.equals(foo)` must be true, and both `foo` and `bar` must return the same value from `hashCode()`. For a Set to treat two objects as duplicates, you must override the `hashCode()` and `equals()` methods inherited from class `Object` so that you can make two different objects be viewed as equal.



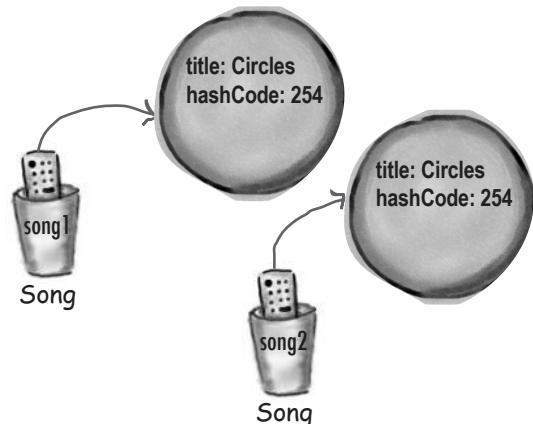
```
if (song1 == song2) {
    // both references are referring
    // to the same object on the heap
}
```

## ► Object equality

Two references, two objects on the heap, but the objects are considered ***meaningfully equivalent***.

If you want to treat two different Song objects as equal (for example if you decided that two Songs are the same if they have matching *title* variables), you must override *both* the `hashCode()` and `equals()` methods inherited from class `Object`.

As we said above, if you *don't* override `hashCode()`, the default behavior (from `Object`) is to give each object a unique hashcode value. So you must override `hashCode()` to be sure that two equivalent objects return the same hashcode. But you must also override `equals()` so that if you call it on *either* object, passing in the other object, always returns `true`.



```
if (song1.equals(song2) && song1.hashCode() == song2.hashCode()) {
    // both references are referring to either a
    // a single object, or to two objects that are equal
}
```

# How a HashSet checks for duplicates: hashCode() and equals()

When you put an object into a HashSet, it calls the object's hashCode method to determine where to put the object in the Set. But it also compares the object's hash code to the hash code of all the other objects in the HashSet, and if there's no matching hash code, the HashSet assumes that this new object is *not* a duplicate.

In other words, if the hash codes are different, the HashSet assumes there's no way the objects can be equal!

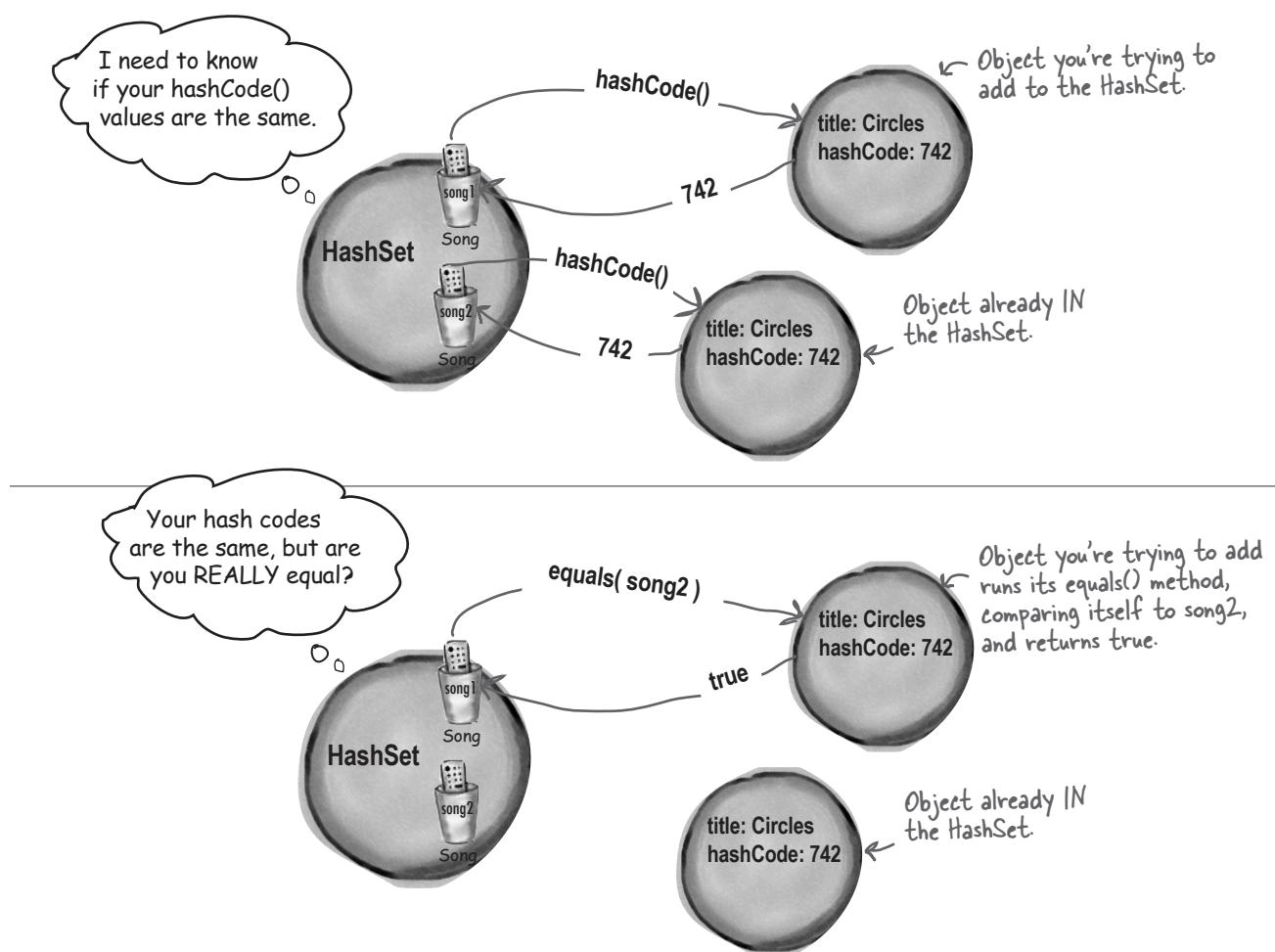
So you must override hashCode() to make sure the objects have the same value.

But two objects with the same hash code might *not* be equal (more on this on the next page), so if the HashSet

finds a matching hash code for two objects—one you're inserting and one already in the set—the HashSet will then call one of the object's equals() methods to see if these hash code-matched objects really *are* equal.

And if they're equal, the HashSet knows that the object you're attempting to add is a duplicate of something in the Set, so the add doesn't happen.

You don't get an exception, but the HashSet's add() method returns a boolean to tell you (if you care) whether the new object was added. So if the add() method returns *false*, you know the new object was a duplicate of something already in the set.



# The Song class with overridden hashCode() and equals()

```

class SongV4 implements Comparable<SongV4> {
    private String title;
    private String artist;
    private int bpm;

    public boolean equals(Object aSong) {
        SongV4 other = (SongV4) aSong;
        return title.equals(other.getTitle()); ←
    }

    public int hashCode() {
        return title.hashCode(); ←
    }

    public int compareTo(SongV4 s) {
        return title.compareTo(s.getTitle());
    }

    SongV4(String title, String artist, int bpm) {
        this.title = title;
        this.artist = artist;
        this.bpm = bpm;
    }

    public String getTitle() {
        return title;
    }

    public String getArtist() {
        return artist;
    }

    public int getBpm() {
        return bpm;
    }

    public String toString() {
        return title;
    }
}

```

The HashSet (or anyone else calling this method) sends it another Song.

The GREAT news is that title is a String, and Strings have an overridden equals() method. So all we have to do is ask one title if it's equal to the other song's title.

Same deal here...the String class has an overridden hashCode() method, so you can just return the result of calling hashCode() on the title. Notice how hashCode() and equals() are using the SAME instance variable (title).

Now it works! No duplicates when we print out the HashSet. But we didn't call sort() again, and when we put the ArrayList into the HashSet, the HashSet didn't preserve the sort order.

```

File Edit Window Help HashingItOut
%java Jukebox9
[somersault, cassidy, $10, havana, $10,
cassidy, 50 ways]

[$10, $10, 50 ways, cassidy, cassidy, havana,
somersault]

[havana, $10, 50 ways, cassidy, somersault]

```

## Java Object Law for hashCode() and equals()

The API docs for class Object state the rules you **MUST** follow:

- ▶ If two objects are equal, they **MUST** have matching hash codes.
- ▶ If two objects are equal, calling equals() on either object **MUST** return true. In other words, if (a.equals(b)) then (b.equals(a)).
- ▶ If two objects have the same hash code value, they are **NOT** required to be equal. But if they're equal, they **MUST** have the same hash code value.
- ▶ So, if you override equals(), you **MUST** override hashCode().
- ▶ The default behavior of hashCode() is to generate a unique integer for each object on the heap. So if you don't override hashCode() in a class, no two objects of that type can EVER be considered equal.
- ▶ The default behavior of equals() is to do an == comparison. In other words, to test whether the two references refer to a single object on the heap. So if you don't override equals() in a class, no two objects can EVER be considered equal since references to two different objects will always contain a different bit pattern.

**a.equals(b)** must also mean that  
**a.hashCode() == b.hashCode()**

But **a.hashCode() == b.hashCode()** does NOT have to mean **a.equals(b)**

## there are no Dumb Questions

**Q:** How come hash codes can be the same even if objects aren't equal?

**A:** HashSets use hash codes to store the elements in a way that makes it much faster to access. If you try to find an object in an ArrayList by giving the ArrayList a copy of the object (as opposed to an index value), the ArrayList has to start searching from the beginning, looking at each element in the list to see if it matches. But a HashSet can find an object much more quickly, because it uses the hash code as a kind of label on the "bucket" where it stored the element. So if you say, "I want you to find an object in the set that's exactly like this one..." the HashSet gets the hash code value from the copy of the Song you give it (say, 742), and then the HashSet says, "Oh, I know exactly where the object with hash code #742 is stored..." and it goes right to the #742 bucket.

This isn't the whole story you get in a computer science class, but it's enough for you to use HashSets effectively. In reality, developing a good hashing algorithm is the subject of many a PhD thesis, and more than we want to cover in this book.

The point is that hash codes can be the same without necessarily guaranteeing that the objects are equal, because the "hashing algorithm" used in the hashCode() method might happen to return the same value for multiple objects. And yes, that means that multiple objects would all land in the same hash code bucket in the HashSet, but that's not the end of the world. The HashSet might be a little less efficient, because if the HashSet finds more than one object in the same hash code bucket, it has to use the equals() on all those objects to see if there's a perfect match.

## TreeSets and sorting

# If we want the set to stay sorted, we've got TreeSet

TreeSet is similar to HashSet in that it prevents duplicates. But it also *keeps* the list sorted. It works just like the sort() method in that if you make a TreeSet without giving it a Comparator, the TreeSet uses each object's compareTo() method for the sort. But you have the option of passing a Comparator to the TreeSet constructor, to have the TreeSet use that instead.

The downside to TreeSet is that if you don't *need* sorting, you're still paying for it with a small performance hit. But you'll probably find that the hit is almost impossible to notice for most apps.

```
public class Jukebox10 {  
    public static void main(String[] args) {  
        new Jukebox10().go();  
    }  
  
    public void go() {  
        List<SongV4> songList = MockMoreSongs.getSongsV4();  
        System.out.println(songList);  
  
        songList.sort((one, two) -> one.getTitle().compareTo(two.getTitle()));  
        System.out.println(songList);  
  
        Set<SongV4> songSet = new TreeSet<>(songList);  
        System.out.println(songSet);  
    }  
}
```

Create a TreeSet instead of HashSet. The  
TreeSet will use SongV4's compareTo()  
method to sort the items in songList.

If we want the TreeSet to sort on something different (i.e., to NOT use SongV4's compareTo() method), we need to pass in a Comparator (or a lambda) to the TreeSet constructor. Then we'd use songSet.addAll() to add the songList values into the TreeSet.

```
Set<SongV4> songSet = new TreeSet<>((o1, o2) -> o1.getBpm() - o2.getBpm());  
songSet.addAll(songList);
```

Yep, another lambda for  
sorting. This one sorts  
by BPM. Remember,  
this lambda implements  
Comparator.

# What you MUST know about TreeSet...

TreeSet looks easy, but make sure you really understand what you need to do to use it. We thought it was so important that we made it an exercise so you'd *have* to think about it. Do NOT turn the page until you've done this. *We mean it.*

→ Answers on page 366.



Look at this code.  
Read it carefully, then  
answer the questions  
below. (Note: there  
are no syntax errors in  
this code.)

```
import java.util.*;

public class TestTree {
    public static void main(String[] args) {
        new TestTree().go();
    }

    public void go() {
        Book b1 = new Book("How Cats Work");
        Book b2 = new Book("Remix your Body");
        Book b3 = new Book("Finding Emo");

        Set<Book> tree = new TreeSet<>();
        tree.add(b1);
        tree.add(b2);
        tree.add(b3);
        System.out.println(tree);
    }
}

class Book {
    private String title;
    public Book(String t) {
        title = t;
    }
}
```

1. What is the result when you compile this code?

---

2. If it compiles, what is the result when you run the TestTree class?

---

3. If there is a problem (either compile-time or runtime) with this code, how would you fix it?

---

# TreeSet elements MUST be comparable

TreeSet can't read the programmer's mind to figure out how the objects should be sorted. You have to tell the TreeSet *how*.

## To use a TreeSet, one of these things must be true:

- The elements in the list must be of a type that implements Comparable

The Book class on the previous page didn't implement Comparable, so it wouldn't work at runtime. Think about it, the poor TreeSet's sole purpose in life is to keep your elements sorted, and once again—it had no idea how to sort Book objects! It doesn't fail at compile-time, because the TreeSet add() method doesn't take a Comparable type. The TreeSet add() method takes whatever type you used when you created the TreeSet. In other words, if you say new TreeSet<Book>(), the add() method is essentially add(Book). And there's no requirement that the Book class implement Comparable! But it fails at runtime when you add the second element to the set. That's the first time the set tries to call one of the object's compareTo() methods and...can't.

## OR

- You use the TreeSet's overloaded constructor that takes a Comparator

TreeSet works a lot like the sort() method—you have a choice of using the element's compareTo() method, assuming the element type implemented the Comparable interface, OR you can use a custom Comparator that knows how to sort the elements in the set. To use a custom Comparator, you call the TreeSet constructor that takes a Comparator.

```
class Book implements Comparable<Book> {  
    private String title;  
    public Book(String t) {  
        title = t;  
    }  
  
    public int compareTo(Book other) {  
        return title.compareTo(other.title);  
    }  
}
```

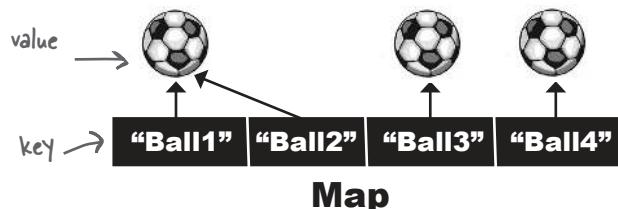
```
class BookCompare implements Comparator<Book> {  
    public int compare(Book one, Book two) {  
        return one.title.compareTo(two.title);  
    }  
}  
public class TestTreeComparator {  
    public void go() {  
        Book b1 = new Book("How Cats Work");  
        Book b2 = new Book("Remix your Body");  
        Book b3 = new Book("Finding Emo");  
        BookCompare bookCompare = new BookCompare();  
        Set<Book> tree = new TreeSet<>(bookCompare);  
        tree.add(b1);  
        tree.add(b2);  
        tree.add(b3);  
        System.out.println(tree);  
    }  
}
```

You could use a lambda instead of declaring a new Comparator class.

# We've seen Lists and Sets, now we'll use a Map

Lists and Sets are great, but sometimes a Map is the best collection (not Collection with a capital "C"—remember that Maps are part of Java collections but they don't implement the Collection interface).

Imagine you want a collection that acts like a property list, where you give it a name and it gives you back the value associated with that name. Keys can be any Java object (or, through autoboxing, a primitive), but you'll often see String keys (i.e., property names) or Integer keys (representing unique IDs, for example).



**Each element in a Map is actually TWO objects—a key and a value. You can have duplicate values, but NOT duplicate keys.**

## Map example

```

public class TestMap {
    public static void main(String[] args) {
        Map<String, Integer> scores = new HashMap<>();
    }
}
  
```

HashMap needs TWO type parameters—one for the key and one for the value.

```

scores.put("Kathy", 42);
scores.put("Bert", 343); ← Use put() instead of add(), and now of course
scores.put("Skyler", 420);
  
```

```

System.out.println(scores);
System.out.println(scores.get("Bert"));
}
  
```

The get() method takes a key and returns the value (in this case, an Integer).

```

File Edit Window Help WhereAmI
% java TestMap
{Skyler=420, Bert=343, Kathy=42}
343
  
```

When you print a Map, it gives you the key=value pairs, in braces { } instead of the brackets [ ] you see when you print lists and sets.

I keep seeing the same code popping up again and again for creating collections. There must be something we can do to make this easier for people.

Good point! Let me just whip up some Factory methods.

## Creating and filling collections

The code for creating, and then filling, a collection crops up again and again. You've already seen code for creating an ArrayList and adding elements to it quite a few times. Code like this:

```
List<String> songs = new ArrayList<>();  
songs.add("somersault");  
songs.add("cassidy");  
songs.add("$10");
```

Whether you're creating a List, a Set, or a Map, it looks pretty similar. What's more, these types of collections are often ones where we know what the data is right at the start, and then we don't intend to change it at all during the lifetime of the collection. If we wanted to really make sure that no-one changed the collection after we'd created it, we'd have to add an extra step:

```
List<String> songs = new ArrayList<>();  
songs.add("somersault");  
songs.add("cassidy");  
songs.add("$10");  
return Collections.unmodifiableList(songs);
```

That's a lot of code! And it's a lot of code for something common that we probably want to do a lot.

Fortunately for us, Java now has “Convenience Factory Methods of Collections” (they were added in Java 9). We can use these methods to create common data structures and fill them with data, with just one method call.

Return an “unmodifiable” version of the list we just created so we know no one else can change it.

We'll see in Chapters 12 and 18 why we might want to create data structures that can't be changed.



# Convenience Factory Methods for Collections

Convenience Factory Methods for Collections allow you to easily create a List, Set, or Map that's been prefilled with known data. There are a couple of things to understand about using them:

- ① **The resulting collections cannot be changed.** You can't add to them or alter the values; in fact, you can't even do the sorting that we've seen in this chapter.
- ② **The resulting collections are not the standard Collections we've seen.** These are not ArrayList, HashSet, HashMap, etc. You can rely on them to behave according to their interface: a List will always preserve the order in which the elements were placed; a Set will never have duplicates. But you can't rely on them being a specific implementation of List, Set, or Map.

Convenience Factory Methods are just that—a convenience that will work for most of the cases where you want to create a collection prefilled with data. And for those cases where these factory methods don't suit you, you can still use the Collections constructors and add() or put() methods instead.

## ► Creating a List: `List.of()`

To create the list of Strings from the last page, we don't need five lines of code; we just need one:

```
List<String> strings = List.of("somersault", "cassidy", "$10");
```

If you want to add Song objects instead of simple Strings, it's still short and descriptive:

```
List<SongV4> songs = List.of(new SongV4("somersault", "zero 7", 147),
                           new SongV4("cassidy", "grateful dead", 158),
                           new SongV4("$10", "hitchhiker", 140));
```

## ► Creating a Set: `Set.of()`

Creating a Set uses very similar syntax:

```
Set<Book> books = Set.of(new Book("How Cats Work"),
                           new Book("Remix your Body"),
                           new Book("Finding Emo"));
```

## ► Creating a Map: `Map.of()`, `Map.ofEntries()`

Maps are different, because they take two objects for each “entry”—a key and a value. If you want to put less than 10 entries into your Map, you can use Map.of, passing in key, value, key, value, etc.:

```
Map<String, Integer> scores = Map.of("Kathy", 42,
   "Bert", 343,
   "Skyler", 420);
```

If you have more than 10 entries, or if you want to be clearer about how your keys are paired up to their values, you can use Map.ofEntries instead:

```
Map<String, String> stores = Map.ofEntries(Map.entry("Riley", "Supersports"),
  Map.entry("Brooklyn", "Camera World"),
  Map.entry("Jay", "Homecase"));
```

To make the line shorter, you can use a *static import* on Map.entry (we talked about static imports in Chapter 10).

## Finally, back to generics

Remember earlier in the chapter we talked about how methods that take arguments with generic types can be...weird. And we mean weird in the polymorphic sense. If things start to feel strange here, just keep going—it takes a few pages to really tell the whole story. The examples are going to use a class hierarchy of Animals.

```
abstract class Animal {
    void eat() {
        System.out.println("animal eating");
    }
}
class Dog extends Animal {
    void bark() { }
}
class Cat extends Animal {
    void meow() { }
}
```



The simplified Animal class hierarchy

## Using polymorphic arguments and generics

Generics can be a little...counterintuitive when it comes to using polymorphism with a generic type (the class inside the angle brackets). Let's create a method that takes a List<Animal> and use this to experiment.

### Passing in List<Animal>

```
public class TestGenerics1 {
    public static void main(String[] args) {
        List<Animal> animals = List.of(new Dog(), new Cat(), new Dog());
        takeAnimals(animals); // Pass a List<Animal> into our testAnimals method
    }

    public static void takeAnimals(List<Animal> animals) {
        for (Animal a : animals) {
            a.eat();
        }
    }
}
```

Using the List.of factory method we just looked at

Method that has a generic class (List) as a parameter

**Compiles and runs just fine**

```
File Edit Window Help CatFoodIsBetter
% java TestGenerics1

animal eating
animal eating
animal eating
```

# But will it work with List<Dog>?

A List<Animal> argument can be passed to a method with a List<Animal> parameter. So the big question is, will the List<Animal> parameter accept a List<Dog>? Isn't that what polymorphism is for?

## Passing in List<Dog>

```
public void go() {
    List<Animal> animals = List.of(new Dog(), new Cat(), new Dog());
    takeAnimals(animals); ← We know this line worked fine.

    List<Dog> dogs = List.of(new Dog(), new Dog());
    takeAnimals(dogs); ← Will this work now that we changed
}                                     from an array to a List?

    public void takeAnimals(List<Animal> animals) {
        for (Animal a : animals) {
            a.eat();
        }
    }
}
```

Make a Dog List and  
put a couple dogs in.

## When we compile it:

```
File Edit Window Help CatsAreSmarter
% javac TestGenerics2.java

TestGenerics2.java:20: error: incompatible types:
List<Dog> cannot be converted to List<Animal>
    takeAnimals(dogs);
               ^
1 error
```

It looked so right,  
but went so wrong...



And I'm supposed  
to be OK with this? That totally  
screws my animal simulation where the  
veterinary program takes a list of any type  
of animal so that a dog kennel can send a list  
of dogs, and a cat kennel can send a list of  
cats...now you're saying I can't do  
that?

## What could happen if it were allowed...?

Imagine the compiler let you get away with that. It let you pass a List<Dog> to a method declared as:

```
public void takeAnimals(List<Animal> animals) {  
    for (Animal a : animals) {  
        a.eat();  
    }  
}
```

There's nothing in that method that *looks* harmful, right? After all, the whole point of polymorphism is that anything an Animal can do (in this case, the eat() method), a Dog can do as well. So what's the problem with having the method call eat() on each of the Dog references?

There's nothing wrong with *that* code. But imagine *this* code instead:

```
public void takeAnimals(List<Animal> animals) {  
    animals.add(new Cat()); ← Yikes!! We just stuck a Cat in what  
    might be a Dogs-only List.  
}
```

So that's the problem. There's certainly nothing wrong with adding a Cat to a List<Animal>, and that's the whole point of having a List of a supertype like Animal—so that you can put all types of animals in a single Animal List.

But if you passed a Dog List—one meant to hold ONLY Dogs—to this method that takes an Animal List, then suddenly you'd end up with a Cat in the Dog list. The compiler knows that if it lets you pass a Dog List into the method like that, someone could, at runtime, add a Cat to your Dog list. So instead, the compiler just won't let you take the risk.

If you declare a method to take List<Animal>, it can take ONLY a List<Animal>, not List<Dog> or List<Cat>.

It seems to me there should be a way to use polymorphic collection types as method arguments so that a vet program could take Dog lists and Cat lists. Then it would be possible to loop through the lists and call their immunize() method. It would have to be safe so that you couldn't add a Cat in to the Dog list.

## We can do this with wildcards

It looks unusual, but there *is* a way to create a method argument that can accept a List of any Animal subtype. The simplest way is to use a **wildcard**.

```
public void takeAnimals(List<? extends Animal> animals) {
    for (Animal a : animals) {
        a.eat();
    }
}
```

*Remember, the keyword "extends" here means either extends OR implements.*

So now you're wondering, "What's the *difference*? Don't you have the same problem as before?"

And you'd be right for wondering. The answer is NO. When you use the wildcard <?> in your declaration, the compiler won't let you do anything that adds to the list!

**When you use a wildcard in your method argument, the compiler will STOP you from doing anything that could hurt the list referenced by the method parameter.**

**You can still call methods on the elements in the list, but you cannot add elements to the list.**

**In other words, you can do things with the list elements, but you can't put new things in the list.**



## there are no Dumb Questions

**Q:** Back when we first saw generic methods, there was a similar-looking method that declared the generic type in front of the method name. Does that do the same thing as this takeAnimals method?

**A:** Well spotted! Back at the start of the chapter, there was a method like this:

```
<T extends Animal> void takeThing(List<T> list)
```

We actually could use this syntax to achieve a similar thing, but it works in a slightly different way. Yes, you can pass List<Animal> and List<Dog> into the method, but you get the added benefit of being able to use the generic type, T, elsewhere too.

## Using the method's generic type parameter

What can we do if we define our method like this instead?

```
public <T extends Animal> void takeAnimals(List<T> list) { }
```

Well, not much as the method stands right now, we don't need to use "T" for anything. But if we made a change to our method to return a List, for example of all the animals we had successfully vaccinated, we can declare that the List that's returned has the same generic type as the List that's passed in:

```
public <T extends Animal> List<T> takeAnimals(List<T> list) { }
```

When you call the method, you know you're going to get the same type back as you put in.

```
List<Dog> dogs = List.of(new Dog(), new Dog());  
List<Dog> vaccinatedDogs = takeAnimals(dogs);  
  
List<Animal> animals = List.of(new Dog(), new Cat());  
List<Animal> vaccinatedAnimals = takeAnimals(animals);
```

The List we get back from the  
takeAnimals method is always the  
same type as the list we pass in.

If the method used the wildcard for both method parameter and return type, there's nothing to guarantee they're the same type. In fact, anything calling the method has almost no idea what's going to be in the collection, other than "some sort of animal."

```
public void go() {  
    List<Dog> dogs = List.of(new Dog(), new Dog());  
    List<? extends Animal> vaccinatedSomethings = takeAnimals(dogs);  
}  
  
public List<? extends Animal> takeAnimals(List<? extends Animal> animals) { }
```

Using the wildcard ("? extends") is fine when you don't care much about the generic type, you just want to allow all subtypes of some type.

Using a type parameter ("T") is more helpful when you want to do more with the type itself, for example in the method's return.



## BE the Compiler, advanced

Your job is to play compiler and determine which of these statements would compile. Some of this code wasn't covered in the chapter, so you need to work out the answers based on what you DID learn, applying the "rules" to these new situations.

The signatures of the methods used in the exercise are in the boxes.

```
private void takeDogs(List<Dog> dogs) { }
```

```
private void takeAnimals(List<Animal> animals) { }
```

```
private void takeSomeAnimals(List<? extends Animal> animals) { }
```

```
private void takeObjects(ArrayList<Object> objects) { }
```

### Compiles?

- `takeAnimals(new ArrayList<Animal>());`
- `takeDogs(new ArrayList<Animal>());`
- `takeAnimals(new ArrayList<Dog>());`
- `takeDogs(new ArrayList<>());`
- `List<Dog> dogs = new ArrayList<>();  
takeDogs(dogs);`
- `takeSomeAnimals(new ArrayList<Dog>());`
- `takeSomeAnimals(new ArrayList<>());`
- `takeSomeAnimals(new ArrayList<Animal>());`
- `List<Animal> animals = new ArrayList<>();  
takeSomeAnimals(animals);`
- `List<Object> objects = new ArrayList<>();  
takeObjects(objects);`
- `takeObjects(new ArrayList<Dog>());`
- `takeObjects(new ArrayList<Object>());`

→ Answers on page 367.

**Exercise Solution**

**Fill-in-the-blanks** (from page 334)

**Possible Answers:**

Comparator,

Comparable,

compareTo( ),

compare( ),

yes,

no

Given the following compilable statement:

```
Collections.sort(myArrayList);
```

1. What must the class of the objects stored in myArrayList implement?
2. What method must the class of the objects stored in myArrayList implement?
3. Can the class of the objects stored in myArrayList implement both Comparator AND Comparable?

Comparable

compareTo( )

yes

Given the following compilable statement:

```
Collections.sort(myArrayList, myCompare);
```

4. Can the class of the objects stored in myArrayList implement Comparable?
5. Can the class of the objects stored in myArrayList implement Comparator?
6. Must the class of the objects stored in myArrayList implement Comparable?
7. Must the class of the objects stored in myArrayList implement Comparator?
8. What must the class of the myCompare object implement?
9. What method must the class of the myCompare object implement?

yes

yes

no

no

Comparator

compare( )

**Solution****“Reverse Engineer” lambdas exercise**

(from page 343)

```

import java.util.*;

public class SortMountains {
    public static void main(String[] args) {
        new SortMountains().go();
    }

    public void go() {
        List<Mountain> mountains = new ArrayList<>();
        mountains.add(new Mountain("Longs", 14255));
        mountains.add(new Mountain("Elbert", 14433));
        mountains.add(new Mountain("Maroon", 14156));
        mountains.add(new Mountain("Castle", 14265));
        System.out.println("as entered:\n" + mountains);

        mountains.sort((mount1, mount2) -> mount1.name.compareTo(mount2.name));
        System.out.println("by name:\n" + mountains);

        mountains.sort((mount1, mount2) -> mount2.height - mount1.height);
        System.out.println("by height:\n" + mountains); ←
    }
}

class Mountain {
    String name;
    int height;

    Mountain(String name, int height) {
        this.name = name;
        this.height = height;
    }

    public String toString() {
        return name + " " + height;
    }
}

```

Did you notice that the height list is  
in DESCENDING sequence? :)

**Output:**

```

File Edit Window Help ThisOne'sForBob
%java SortMountains
as entered:
[Longs 14255, Elbert 14433, Maroon 14156, Castle 14265]
by name:
[Castle 14265, Elbert 14433, Longs 14255, Maroon 14156]
by height:
[Elbert 14433, Castle 14265, Longs 14255, Maroon 14156]

```

## Solution



### Sorting with lambdas (from page 342)

Sort by BPM ascending

```
songList.sort((one, two) -> one.getBpm() - two.getBpm());
```

Sort by title descending

```
songList.sort((one, two) -> two.getTitle().compareTo(one.getTitle()));
```

#### Output:

```
File Edit Window Help IntNotString
%java SharpenLambdas
[50 ways, havana, $10, somersault, cassidy, Cassidy]
[somersault, havana, cassidy, Cassidy, 50 ways, $10]
```

## Solution



### TreeSet exercise

(from page 353)

1. What is the result when you compile this code?

**It compiles correctly**

---

2. If it compiles, what is the result when you run the TestTree class?

**It throws an exception:**

```
Exception in thread "main" java.lang.ClassCastException: class Book cannot be cast to class java.lang.Comparable
        at java.base/java.util.TreeMap.compare(TreeMap.java:1291)
        at java.base/java.util.TreeMap.put(TreeMap.java:536)
        at java.base/java.util.TreeSet.add(TreeSet.java:255)
        at TestTree.go(TestTree.java:16)
        at TestTree.main(TestTree.java:7)
```

3. If there is a problem (either compile-time or runtime) with this code, how would you fix it?

**Make Book implement Comparable, or pass the TreeSet a Comparator**

---

See page 574



## BE the Compiler solution

(from page 363)

### Compiles?

- takeAnimals(new ArrayList<Animal>());
- takeDogs(new ArrayList<Animal>());
- takeAnimals(new ArrayList<Dog>());
- takeDogs(new ArrayList<>()); ←
- List<Dog> dogs = new ArrayList<>();  
takeDogs(dogs);
- takeSomeAnimals(new ArrayList<Dog>());
- takeSomeAnimals(new ArrayList<>()); ←
- takeSomeAnimals(new ArrayList<Animal>());
- List<Animal> animals = new ArrayList<>();  
takeSomeAnimals(animals);
- List<Object> objects = new ArrayList<>();  
takeObjects(objects);
- takeObjects(new ArrayList<Dog>());
- takeObjects(new ArrayList<Object>());

If you use the diamond operator here, it works out the type from the method signature. Therefore, the compiler assumes this ArrayList is ArrayList<Dog>.

Here the diamond operator means this is ArrayList<Animal>.

This doesn't compile because takeObjects wants an ArrayList, not a List.



# Lambdas and Streams: What, Not How



Did you know, you don't have to write absolutely everything yourself? You can get the APIs to do the work for you!

**What if...you didn't need to tell the computer HOW to do something?** Programming involves a lot of telling the computer how to do something: **while** this is true **do** this thing; **for** all these items **if** it looks like this **then** do this; and so on.

We've also seen that we don't have to do everything ourselves. The JDK contains library code, like the Collections API we saw in the previous chapter, that we can use instead of writing everything from scratch. This library code isn't just limited to collections to put data into; there are methods that will do common tasks for us, so we just need to tell them **what** we want and not **how** to do it.

In this chapter we'll look at the Streams API. You'll see how helpful lambda expressions can be when you're using streams, and you'll learn how to use the Streams API to query and transform the data in a collection.

# Tell the computer **WHAT** you want

Imagine you have a list of colors, and you wanted to print out all the colors. You could use a for loop to do this.

```
List<String> allColors = List.of("Red", "Blue", "Yellow");
for loop {
    for (String color : allColors) {
        System.out.println(color);
    }
}
```

This is a “convenience factory method” for creating a new List from a known group of values. We saw this in Chapter 11.

For each item in the list create a temporary variable, color...

...then print out each color.

But doing something to every item in a list is a really common thing to want to do. So instead of creating a for loop every time we want to do something “for each” item in the list, we can call the **forEach** method from the Iterable interface—remember, List implements Iterable so it has all the methods from the Iterable interface.

```
List<String> allColors = List.of("Red", "Blue", "Yellow");
allColors.forEach(color -> System.out.println(color));
```

For each item in the list...

Create a temporary variable named color

Print out the color

```
File Edit Window Help SingARainbow
% java PrintColors
Red
Blue
Yellow
```

The **forEach** method of a list takes a lambda expression, which we saw for the first time in the previous chapter. This is a way for you to pass behavior (“follow these instructions”) into a method, instead of passing an object containing data (“here is an object for you to use”).

## Fireside Chats



### for loop

I am the default! The for loop is so important that loads of programming languages have me. It's one of the first things a programmer learns! If someone needs to loop a set number of times to do something, they're going to reach for their trusty for loop.

Sure, fashions change. But sometimes it's just a fad; things fall out of fashion too. A classic like me will be easy to read and write forever, even for non-Java programmers.

So much work?! Ha! A developer isn't scared of a little syntax to clearly specify what to do and how to do it. At least with me, someone reading my code can clearly see what's going on.

Well I'm faster. Everyone knows that.

I said you would disappear soon.

Tonight's Talk: **The for loop and forEach method battle over the question, "Which is better?"**

### forEach()

Pff. Please. You are *so old*; that's why you're in all the programming languages. But things change, languages evolve. There's a better way. A more modern way. Me.

But look how much work developers need to do to write you! They have to control when to start, increment, and stop the loop, as well as writing the code that needs to be run inside the loop. All sorts of things could go wrong! If they use me, they just have to think about what needs to happen to each item, they don't have to worry about how to loop to find each item.

Dude, they shouldn't *have* to see what's going on. It says very clearly in my method name exactly what I do—"for each" element I will apply the logic they specify. Job done.

Well actually, under the covers I'm using a for loop myself, but if something is invented later that's even faster, I can use that, and developers don't have to change a single thing to get faster code. In fact we're out of time now so....

# When for loops go wrong

Using **forEach** instead of a for loop means a bit less typing, and it's also nice to focus on telling the compiler *what* you want to do and not *how* to do it. There's another advantage to letting the libraries take care of routine code like this—it can mean fewer accidental errors.



A short Java program is listed below. One block of the program is missing. We expect the output of the program should be "1 2 3 4 5" but sometimes it's difficult to get a for loop just right.

Your challenge is to **match the candidate block of code** (on the left) **with the output** that you'd see if the block were inserted. Not all the lines of output will be used, and some of the lines of output might be used more than once.

```
class MixForLoops {
    public static void main(String [] args) {
        List<Integer> nums = List.of(1, 2, 3, 4, 5);
        String output = "";
        
        System.out.println(output);
    }
}
```

*Candidate code  
goes here*

## Candidates:

```
for (int i = 1; i < nums.size(); i++)
    output += nums.get(i) + " ";

for (Integer num : nums)
    output += num + " ";

for (int i = 0; i <= nums.length; i++)
    output += nums.get(i) + " ";

for (int i = 0; i <= nums.size(); i++)
    output += nums.get(i) + " ";
```

## Possible output:

1 2 3 4 5

Compiler error

2 3 4 5

Exception thrown

[1, 2, 3, 4, 5]  
[1, 2, 3, 4, 5]  
[1, 2, 3, 4, 5]  
[1, 2, 3, 4, 5]  
[1, 2, 3, 4, 5]

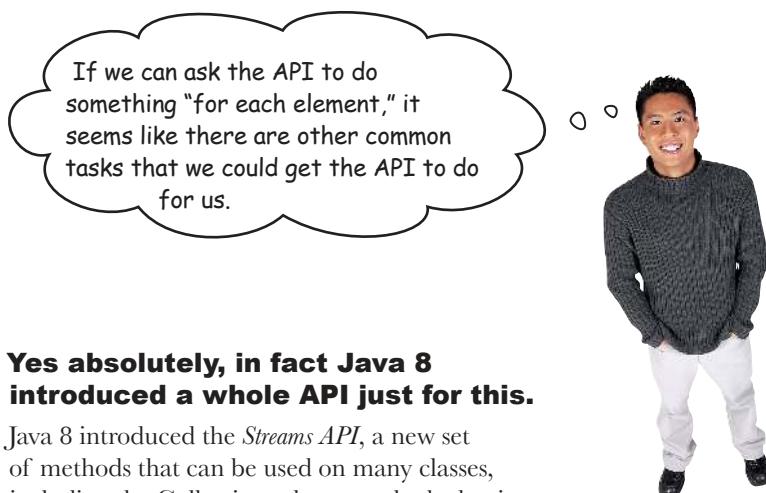
*Match each candidate with one of the possible outputs*

→ Answers on page 417.

# Small errors in common code can be hard to spot

The for loops from the previous exercise all look quite similar, and at first glance they all look like they would print out all the values in the List in order. Compiler errors can be easiest to spot, because your IDE or compiler will tell you the code is wrong, and Exceptions (which we'll see in Chapter 13, *Risky Behavior*) can also point to a problem in the code. But it can be trickier to spot code that produces incorrect output just by looking at the code.

Using a method like **forEach** takes care of the “boilerplate,” the repetitive and common code like the for loop. Using forEach, passing in only the thing we want to do, can reduce accidental errors in our code.



## **Yes absolutely, in fact Java 8 introduced a whole API just for this.**

Java 8 introduced the *Streams API*, a new set of methods that can be used on many classes, including the Collections classes we looked at in the previous chapter.

The Streams API isn't just a bunch of helpful methods, but also a slightly different way of working. It lets us build up a whole set of requirements, a recipe if you like, of what we want to know about our data.

**Brain Barbell**

Can you think of more examples of the types of things we might want to do to a collection? Are you going to want to ask similar questions about what's inside different types of collections? Can you think of different types of information you might want to output from a collection?

## Building blocks of common operations

The ways we search our collections, and the types of information we want to output from those collections, can be quite similar even on different types of collections containing different types of Objects.

Imagine what you might want do to with a Collection: “give me just the items that meet some criteria,” “change all the items using these steps,” “remove all duplicates,” and the example we worked through in the previous chapter: “sort the elements in this way.”

It’s not too hard to go one step further and assume each of these collection operations could be given a name that tells us what will happen to our collection.



We know this is all new, but have a go at matching each operation name to the description of what it does. Try not to look at the next page as you complete it, as that will give the game away!

|           |                                                                                      |
|-----------|--------------------------------------------------------------------------------------|
| filter    | Changes the current element in the stream into something else                        |
| skip      | Sets the maximum number of elements that can be output from this Stream              |
| limit     | While a given criteria is true, will not process elements                            |
| distinct  | Only allows elements that match the given criteria to remain in the Stream           |
| sorted    | Will only process elements while the given criteria is true                          |
| map       | States the result of the stream should be ordered in some way                        |
| dropWhile | This is the number of elements at the start of the Stream that will not be processed |
| takeWhile | Use this to make sure duplicates are removed                                         |

—————> Answers on page 417.

# Introducing the Streams API

The Streams API is a set of operations we can perform on a collection, so when we read these operations in our code, we can understand what we're trying to do with the collection data. If you were successful in the “Who Does What?” exercise on the previous page (the complete answers are at the end of this chapter), you should have seen that the names of the operations describe what they do.

**java.util.stream.Stream**

|                                                                                     |                                                                                                                 |
|-------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------|
| <b>Stream&lt;T&gt; distinct()</b>                                                   | >Returns a stream consisting of the distinct elements                                                           |
| <b>Stream&lt;T&gt; filter(Predicate&lt;? super T&gt; predicate)</b>                 | >Returns a stream of the elements that match the given predicate.                                               |
| <b>Stream&lt;T&gt; limit(long maxSize)</b>                                          | >Returns a stream of elements truncated to be no longer than max-size in length.                                |
| <b>&lt;R&gt; Stream&lt;R&gt; map(Function&lt;? super T, ? extends R&gt; mapper)</b> | >Returns a stream with the results of applying the given function to the elements of this stream.               |
| <b>Stream&lt;T&gt; skip(long n)</b>                                                 | >Returns a stream of the remaining elements of this stream after discarding the first n elements of the stream. |
| <b>Stream&lt;T&gt; sorted()</b>                                                     | >Returns a stream of the elements of this stream, sorted according to natural order.                            |

**// more**

(These are just a few of the methods in Stream... there are many more.)

These generics do look a little intimidating, but don't panic! We'll use the map method later, and you'll see it's not as complicated as it seems.

Streams, and lambda expressions, were introduced in Java 8.



You don't need to worry too much about the generic types on the Stream methods; you'll see that using Streams “just works” the way you'd expect.

In case you are interested:

- <**T**> is usually the Type of the object in the stream.
- <**R**> is usually the type of the Result of the method.

# Getting started with Streams

Before we start going into detail about what the Streams API is, what it does, and how to use it, we're going to give you some very basic tools to start experimenting.

To use the Streams methods, we need a Stream object (obviously). If we have a collection like a List, this doesn't implement Stream. However, the Collection interface has a method, **stream**, which returns a Stream object for the Collection.

```
List<String> strings = List.of("I", "am", "a", "list", "of", "Strings");
Stream<String> stream = strings.stream();
```

*Assuming we had a List of Strings like this...*

*...we can call this method to get a Stream of these Strings.*

Now we can call the methods of the Streams API. For example, we could use **limit** to say we want a maximum of four elements.

```
stream<String> limit = stream.limit(4);
```

*The limit method returns another Stream of Strings, which we'll assign to another variable*

*Sets the maximum number of results to return to 4*

What happens if we try to print out the result of calling limit()?

```
System.out.println("limit = " + limit);
```

```
File Edit Window Help SliceAndDice
%java LimitWithStream
limit = java.util.stream.SliceOps$1@7a0ac6e3
```

*This doesn't look right at all! What's a SliceOps, and why isn't there a collection of just the first four items from the list?*

Like everything in Java, the stream variables in the example are Objects. But a stream does **not** contain the elements in the collection. It's more like the set of instructions for the operations to perform on the Collection data.

**Stream methods that return another Stream are called Intermediate Operations. These are instructions of things to do, but they don't actually perform the operation on their own.**



What's the point  
of having a method  
called limit if it doesn't  
actually limit my results?  
How am I supposed to see  
the output of the  
method?

## Streams are like recipes: nothing's going to happen until someone actually cooks them

A recipe in a book only tells someone *how* to cook or bake something. Opening the recipe doesn't automatically present you with a freshly baked chocolate cake. You need to gather the ingredients according to the recipe and follow the instructions exactly to come up with the result you want.

Collections are not ingredients, and a list limited to four entries is not a chocolate cake (sadly). But you do need to call one of the Stream's "do it" methods in order to get the result you want. These "do it" methods are called **Terminal Operations**, and these are the methods that will actually return something to you.



(These are some terminal operations on Stream.)

### java.util.stream.Stream

**boolean anyMatch(Predicate<? super T> predicate)**  
Returns true if any element matches the provided predicate.

**long count()**  
Returns the number of elements in this stream.

**<R,A> R collect(Collector<? super T,A,R> collector)**  
Performs a mutable reduction operation on the elements of this stream using a Collector.

**Optional<T> findFirst()**  
Returns an Optional describing the first element of this stream, or an empty Optional if the stream is empty.

// more

Yes, this looks even scarier than the map method! Don't panic, these generic types help the compiler, but you'll see when we actually use this method, we don't have to think about these generic types.

# Getting a result from a Stream

Yes, we've thrown a **lot** of new words at you: *streams*; *intermediate operations*; *terminal operations*... And we still haven't told you what streams can do!

To start to get a feel for what we can do with streams, we're going to show code for a simple use of the Streams API. After that, we'll step back and learn more about what we're seeing here.

```
List<String> strings = List.of("I", "am", "a", "list", "of", "Strings");
```

```
Stream<String> stream = strings.stream();
Stream<String> limit = stream.limit(4);
long result = limit.count();
```

Call the count terminal operator, and store the output in a variable called result

```
File Edit Window Help WellDuh
%java LimitWithStream

result = 4
```

This works, but it's not very useful. One of the most common things to do with Streams is put the results into another type of collection. The API documentation for this method might seem intimidating with all the generic types, but the simplest case is straightforward:

The stream contained Strings, so the output object will also contain Strings.

Terminal operation that will collect the output into some sort of Object.

This method returns a Collector that will output the results of the stream into a List.

```
List<String> result = limit.collect(Collectors.toList());
```

The `toList` Collector will output the results as a List.

A helpful class that contains methods to return common Collector implementations.

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

```
File Edit Window Help FinallyAResult
%java LimitWithStream

result = [I, am, a, list]
```



We'll see `collect()` and the `Collectors` in more detail later.

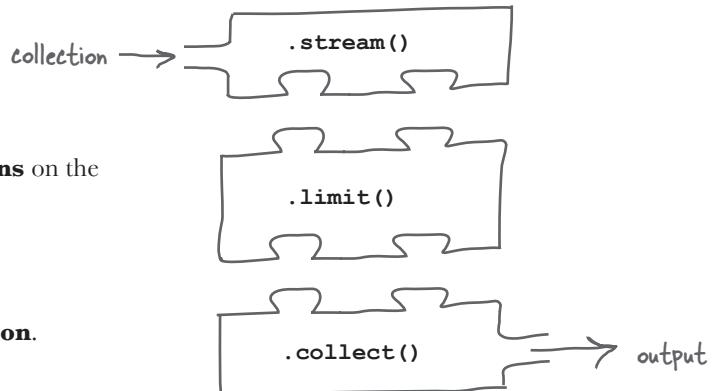
For now, `collect(Collectors.toList())` is a magic incantation to get the output of the stream pipeline in a List.

Finally, we have a result that looks like something we would have expected: we had a List of Strings, and we asked to **limit** that list to the first four items and then **collect** those four items into a new List.

# Stream operations are building blocks

We wrote a lot of code just to output the first four elements in the list. We also introduced a lot of new terminology: streams, intermediate operations, and terminal operations. Let's put all this together: you create a **stream pipeline** from three different types of building blocks.

- ① Get the Stream from a **source collection**.



- ② Call zero or more **intermediate operations** on the Stream.

- ③ Output the results with a **terminal operation**.

You need at least the **first** and **last** pieces of the puzzle to use the Streams API. However, you don't need to assign each step to its own variable (which we were doing on the last page). In fact, the operations are designed to be **chained**, so you can call one stage straight after the previous one, without putting each stage in its own variable.

On the last page, all the building blocks for the stream were highlighted (stream, limit, count, collect). We can take these building blocks and rewrite the limit-and-collect operation in this way:

```
List<String> strings = List.of("I", "am", "a", "list", "of", "Strings");
```

```

List<String> result = strings.stream()
    .limit(4)
    .collect(Collectors.toList());
  
```

Formatted to align each operation directly underneath the one above, to clearly show each stage.

Get the stream for the collection

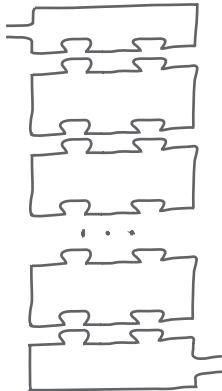
Set a limit to return a maximum of 4 results from the stream

Returns the results of the operation as a List

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

## Building blocks can be stacked and combined

Every intermediate operation acts on a Stream and returns a Stream. That means you can stack together as many of these operations as you want, before calling a terminal operation to output the results.



The source, the intermediate operation(s), and the terminal operation all combine to form a Stream Pipeline. This pipeline represents a query on the original collection.

This is where the Streams API becomes really useful. In the earlier example, we needed three building blocks (stream, limit, collect) to create a shorter version of the original List, which may seem like a lot of work for a simple operation.

But to do something more complicated, we can stack together multiple operations in a single **stream pipeline**.

For example, we can sort the elements in the stream before we apply the limit:

```
List<String> strings = List.of("I", "am", "a", "list", "of", "Strings");

List<String> result = strings.stream()
    .sorted()           ← Sort what's in the stream (not the
                        original collection), using natural
                        order, before limiting the results.
    .limit(4)          ← Limit the stream to just
                        four elements.
    .collect(Collectors.toList());

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

```
File Edit Window Help InChains
%java ChainedStream
result = [I, Strings, a, am]
```

← Natural ordering of Strings will place capitalized Strings ahead of lowercase Strings.

## Customizing the building blocks

We can stack together operations to create a more advanced query on our collection. We can also customize what the blocks do too. For example, we customized the `limit` method by passing in the maximum number of items to return (four).

If we didn't want to use the natural ordering to sort our Strings, we could define a specific way to sort them. It's possible to set the sort criteria for the `sorted` method (remember, we did something similar in the previous chapter when we sorted Lou's song list).

```
List<String> result = strings.stream()
    .sorted((s1, s2) -> s1.compareToIgnoreCase(s2))
    .limit(4)
    .collect(Collectors.toList());
```

File Edit Window Help IgnoreCaps

```
%java ChainedStream
result = [a, am, I, list]
```

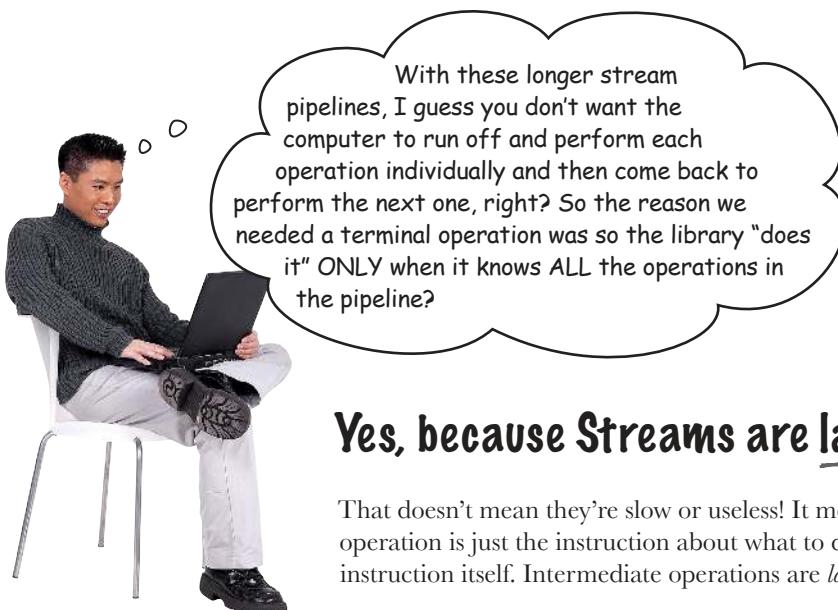
## Create complex pipelines block by block

Each new operation you add to the pipeline changes the output from the pipeline. Each operations tell the Streams API *what* it is you want to do.

```
List<String> result = strings.stream()
    .sorted((s1, s2) -> s1.compareToIgnoreCase(s2))
    .skip(2)
    .limit(4)
    .collect(Collectors.toList());
```

```
File Edit Window Help BoxersDolt
%java ChainedStream
result = [I, list, of, Strings]
```

The stream skipped over the first two elements.



With these longer stream pipelines, I guess you don't want the computer to run off and perform each operation individually and then come back to perform the next one, right? So the reason we needed a terminal operation was so the library "does it" ONLY when it knows ALL the operations in the pipeline?

## Yes, because Streams are lazy

That doesn't mean they're slow or useless! It means that each intermediate operation is just the instruction about what to do; it doesn't perform the instruction itself. Intermediate operations are *lazily evaluated*.

The terminal operation is responsible for looking at the whole list of instructions, all those intermediate operations in the pipeline, and then running the whole set together in one go. Terminal operations are *eager*; they are run as soon as they're called.

This means that in theory it's possible to run the combination of instructions in the most efficient way. Instead of having to iterate over the original collection for each and every intermediate operation, it may be possible to do all the operations while only going through the data once.



I only start my day once I know exactly what I'm going to do, and exactly how to do it.

# Terminal operations do all the work

Since intermediate operations are *lazy*, it's up to the terminal operation to do everything.

- 1** Perform all the intermediate operations as efficiently as possible. Ideally, just going through the original data once.
- 2** Work out the result of the operation, which is defined by the terminal operation itself. For example, this could be a list of values, a single value, or a boolean (true/false).
- 3** Return the result.

## Collecting to a List

Now that we know more about what's going on in a terminal operation, let's take a closer look at the "magic incantation" that returns a list of results.

```
List<String> result = strings.stream()
    .sorted()
    .skip(2)
    .limit(4)
    .collect(Collectors.toList());
```

Terminal operation:

1. performs all intermediate operations, in this case: sort; skip; limit.
2. collects the results according to the instructions passed into it
3. returns those results

Collectors is a class that has static methods that provide different implementations of Collector. Look at the Collectors class to find the most common ways to collect up the results.

The collect method takes a Collector, the recipe for how to put together the results. In this case, it's using a helpful predefined Collector that puts the results into a List.

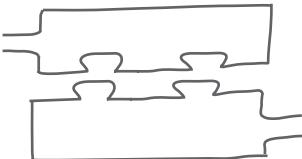
We will look at more Collectors, and other terminal operations, later in the chapter. For now, you know enough to get going with Streams.

## Guidelines for working with streams

Like any puzzle or game, there are rules for getting the stream building blocks to work properly.

### ① You need at least the first and last pieces to create a stream pipeline.

Without the `stream()` piece, you don't get a Stream at all, and without the terminal operation, you're not going to get any results.



### ② You can't reuse Streams.

It might seem useful to store a Stream representing a query, and reuse it in multiple places, either because the query itself is useful or because you want to build on it and add to it. But once a terminal operation has been called on a stream, you can't reuse any parts of that stream; you have to create a new one. Once a pipeline has executed, that stream is closed and can't be used in another pipeline, even if you stored part of it in a variable for reusing elsewhere. If you try to reuse a stream in any way, you'll get an Exception.

```
Stream<String> limit = strings.stream()
                                .limit(4);
List<String> result = limit.collect(Collectors.toList());
List<String> result2 = limit.collect(Collectors.toList());
```

```
File Edit Window Help ClosingTime
%java LimitWithStream

Exception in thread "main" java.lang.IllegalStateException: stream has
already been operated upon or closed
        at java.base/java.util.stream.AbstractPipeline.
evaluate(AbstractPipeline.java:229)
```

### ③ You can't change the underlying collection while the stream is operating.

If you do this, you'll see strange results, or exceptions. Think about it—if someone asked you a question about what was in a shopping list and then someone else was scribbling on that shopping list at the same time, you'd give confusing answers too.



I'm so confused!  
I'm just trying to  
read this list, but  
it keeps changing!



So if you shouldn't change the underlying collection while you're querying it, the stream operations don't change the collection either, right?

## Correct! Stream operations don't change the original collection.

The Streams API is a way to query a collection, but it **doesn't make changes** to the collection itself. You can use the Streams API to look through that collection and return results based on the contents of the collection, but your original collection will remain the same as it was.

This is actually very helpful. It means you can query collections and output the results from anywhere in your program and know that the data in your original collection is safe; it will not be changed ("mutated") by any of these queries.

You can see this in action by printing out the contents of the original collection after using the Streams API to query it.

```
List<String> strings = List.of("I", "am", "a", "list", "of", "Strings");

Stream<String> limit = strings.stream()
    .limit(4)
    .collect(Collectors.toList());

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

```
File Edit Window Help Untouchable
%java LimitWithStream

strings = [I, am, a, list, of, Strings]
result = [I, am, a, list]
```

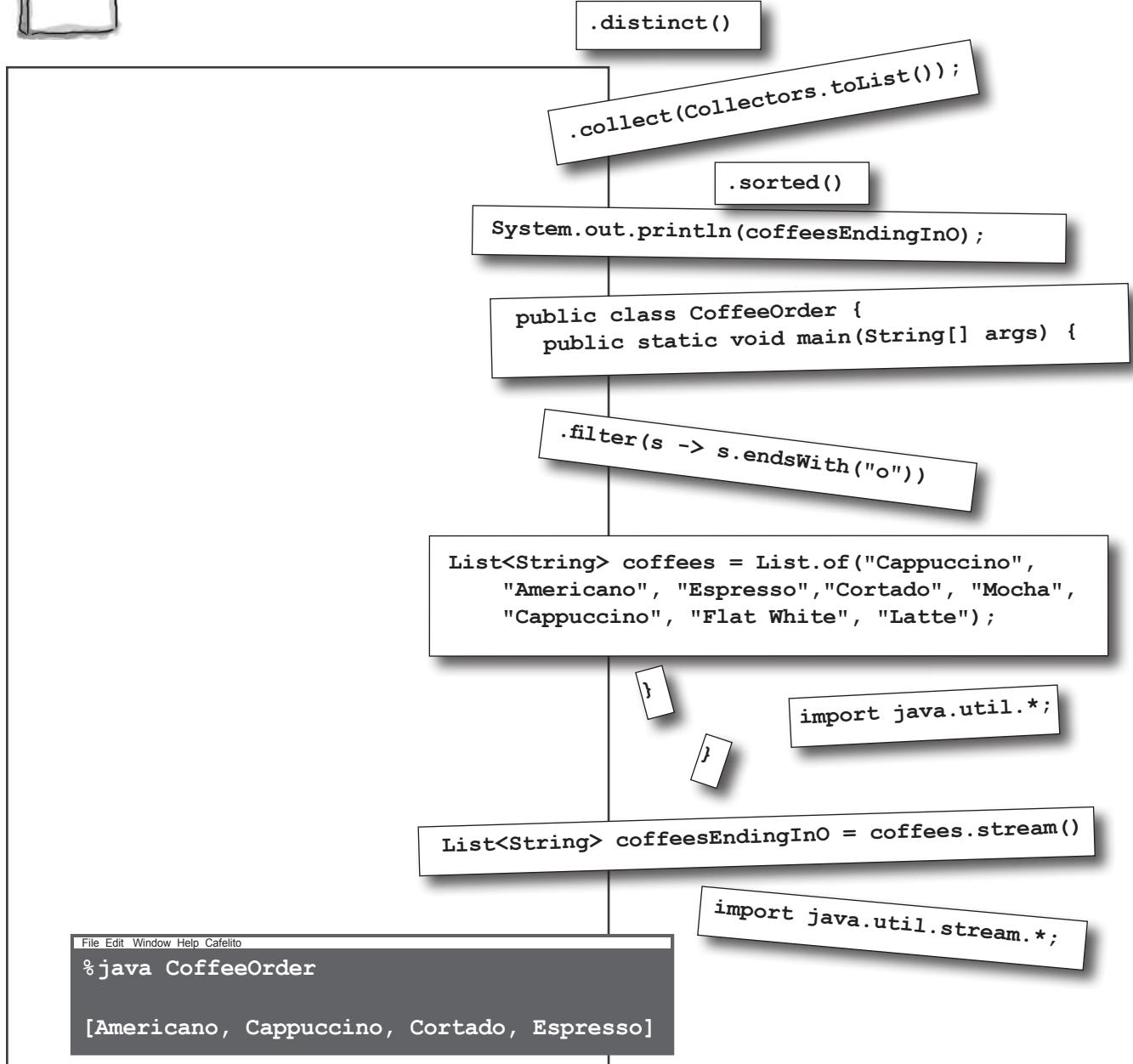
No changes to original collection after the stream operations are run.

Only the output object has the results of the query. This is a brand new List.



# Code Magnets

A Java program is all scrambled up on the fridge. Can you reconstruct the code snippets to make a working Java program that produces the output listed below?



there are no  
Dumb Questions

**Q:** Is there a limit to the number of intermediate operations I can put in a stream pipeline?

**A:** No, you can keep chaining these operations as much as you like. But do remember that it's not just computers that have to read and understand this code; it's humans too! If the stream pipeline is really long, it might be too complicated to understand. That's when you might want to split it up and assign sections to variables, so you can give these variables useful names.

**Q:** Is there any point in having a stream pipeline *without* intermediate operations?

**A:** Yes, you might find that there's a terminal operation that outputs the original collection in some new shape, which is just right for what you need. Be aware, however, that some of the terminal operations are similar to methods that exist on the collection; you don't always need to use streams. For example, if you're just using `count` on a Stream, you could probably use `size` instead, if your original collection is a List. Similarly, anything that is Iterable (like List) already has a `forEach` method; you don't need to use `stream().forEach()`.

**Q:** You said not to change the source collection while the stream operation is in progress. How is it possible to change the collection from my code, if my code is doing a stream operation?

**A:** Great question! It's possible to write programs that run different bits of code at the same time. We'll learn about this in Chapters 17 and 18, which cover concurrency. To be safe, it's usually best (not just for Streams, but in general) to create collections that can't be changed if you know they don't need to be changed.

**Q:** How can I output a List that can't be changed from the `collect` terminal operation?

**A:** If you're using Java 10 or higher, you can use `Collectors.toUnmodifiableList`, instead of using `Collectors.toList`, when you call `collect`.

**Q:** Can I get the results of the stream pipeline in a collection that isn't a List?

**A:** Yes! In the previous chapter we learned that there are a few different kinds of collections for different purposes. The `Collectors` class has convenience methods for collecting `toList`, `toSet`, and `toMap`, as well as (since Java 10) `toUnmodifiableList`, `toUnmodifiableSet`, and `toUnmodifiableMap`.

### BULLET POINTS

- You don't have to write detailed code telling the JVM exactly what to do and how to do it. You can use library methods, including the Streams API, to query collections and output the results.
- Use `forEach` on a collection instead of creating a `for` loop. Pass the method a lambda expression of the operation to perform on each element of the collection.
- Create a stream from a collection (a **source**) by calling the **stream** method.
- Configure the query you want to run on the collection by calling one or more **intermediate operations** on the stream.
- You won't get any results until you call a terminal operation. There are a number of different **terminal operations** depending upon what you want your query to output.
- To output the results into a new List, use `collect(Collectors.toList)` as the terminal operation.
- The combination of the source collection, intermediate operations, and terminal operations is a **stream pipeline**.
- Stream operations do not change the original collection; they are a way to query the collection and return a different Object, which is a result of the query.

## Hello Lambda, my (not so) old friend

Lambda expressions have cropped up in the streams examples so far, and you can bet your bottom dollar (or euro, or currency of your choice) that you're going to see more of them before this chapter is done.

Having a better understanding of what lambda expressions are will make it easier to work with the Streams API, so let's take a closer look at lambdas.

### Passing behavior around

If you wrote a **forEach** method, it might look something like this:

```
void forEach( ?????? ) {  
    for (Element element : list) {  
        }  
    }  
}
```

This is the space where the block of code to run for every list element would go.

What would you put in the place where “?????” is? It would need to somehow **be** the block of code that's going to go into that nice, blank square.

Then you want someone calling the method to be able to say:

```
forEach(do this: System.out.println( item ));
```

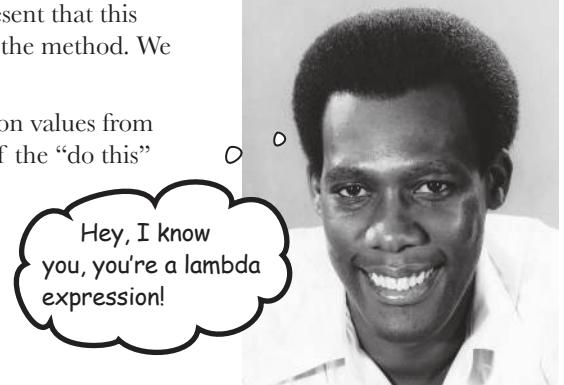
You can't just write this code here, because it will be run straightaway. Instead, we need a way to hand this block of code over to the `forEach` method so that method can call it when it's ready.

This code needs to somehow get hold of the element to print it, but how can it get an element when that code is INSIDE the `forEach` method?

Now, we need to replace the *do this* with some sort of symbol to represent that this code isn't to be run straightaway, but instead needs to be passed into the method. We could use, oh, let's see... “->” as this symbol.

Then we need a way to say “look, this code is going to need to work on values from elsewhere.” We could put the things the code needs on the left side of the “do this” symbol....

```
forEach( item -> System.out.println(item) );
```





OK, so now I get that the lambda I pass in as a method argument is somehow used in the body of that method. But what is the lambda? How can the method USE this chunk of code I just passed it?

## Lambda expressions are objects, and you run them by calling their Single Abstract Method

Remember, everything in Java is an Object (well, except for the primitive types), and lambdas are no exception.

## A lambda expression implements a Functional Interface.

This means the reference to the lambda expression is going to be a Functional Interface. So, if you want your method to accept a lambda expression, you need to have a parameter whose type is a functional interface. That functional interface needs to be the right “shape” for your lambda.

Back to our imaginary **forEach** example; our parameter needs to implement a Functional Interface. We also need to call that lambda expression somehow, passing in the list element.

Remember, Functional Interfaces have a Single Abstract Method (SAM) . It's this method, whatever its name is, that gets called when we want to run the lambda code.

```
void forEach( SomeFunctionalInterface lambda ) {
    for (Element element : list) {
        lambda.singleAbstractMethodName(element);
    }
}
```

*This would be the name of whatever is the Single functional interface.*

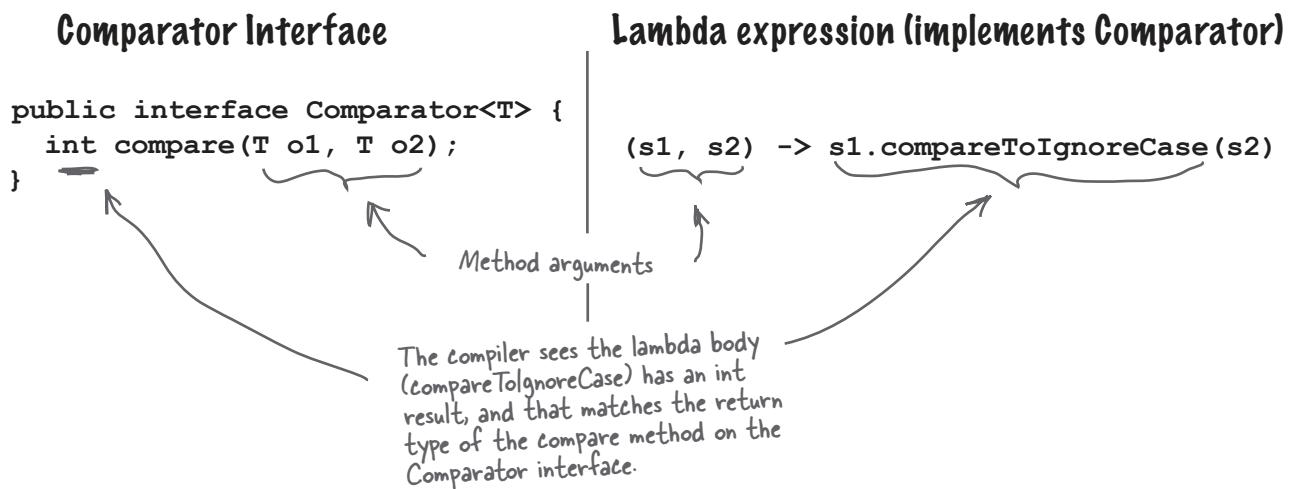
*This is a placeholder type to give you an idea of what the method would look like. We'll look at specific Functional Interfaces throughout this chapter.*

*“element” is the lambda’s parameter, the “item” in the lambda expression on the last page.*

Lambdas aren't magic;  
they're just classes  
like everything else.

# The shape of lambda expressions

We've seen two lambda expressions that implement the Comparator interface: the example for sorting Lou's songs in the previous chapter, and the lambda expression we passed into the `sorted()` stream operation on page 381. Look at this last example side by side with the Comparator Functional Interface.



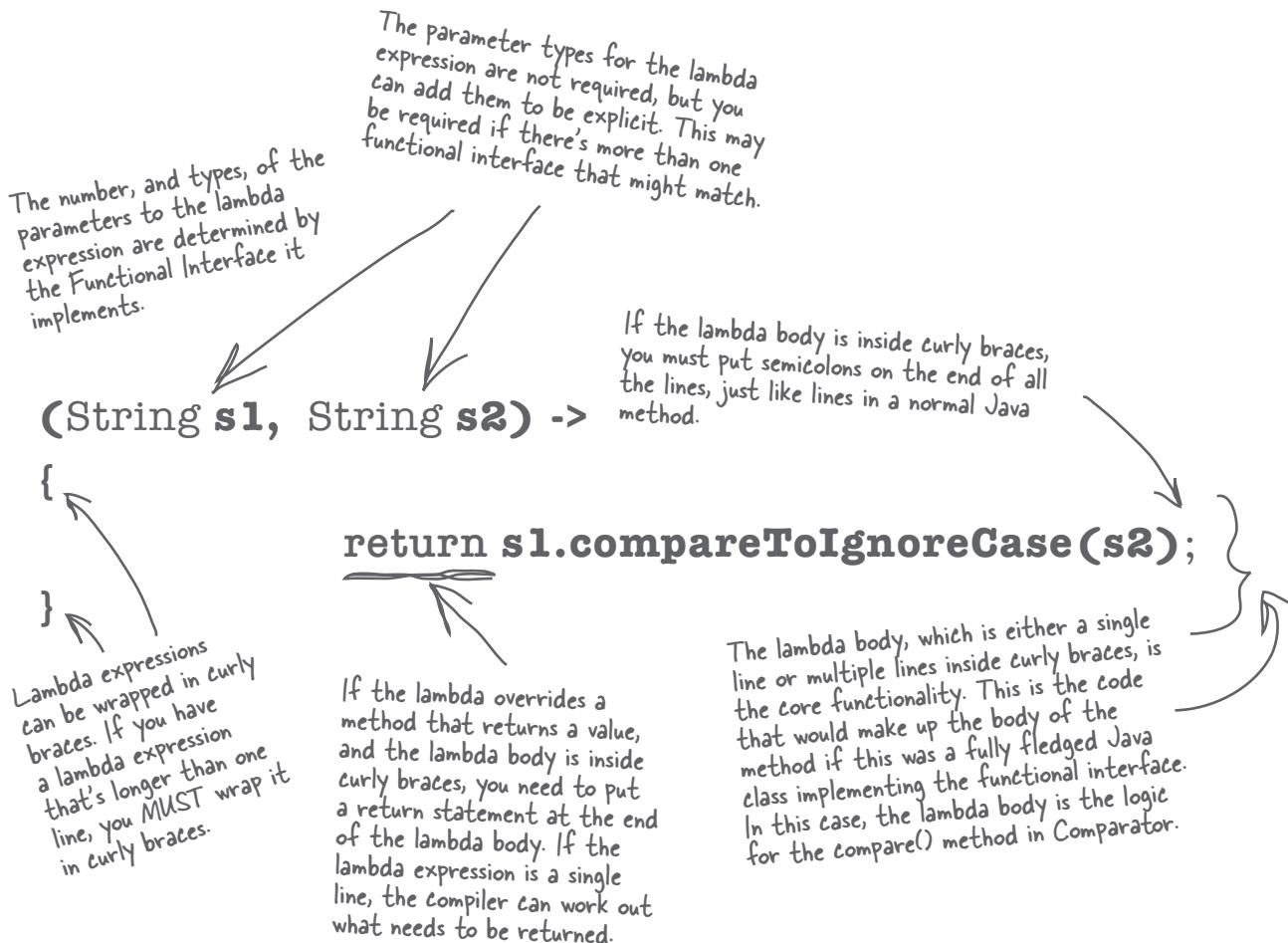
You might be wondering where the **return** keyword is in the lambda expression. The short version is: you don't need it. The longer version is, if the lambda expression is a single line, and if the functional interface's method signature requires a returned value, the compiler just assumes that your one line of code will generate the value that is to be returned.

The lambda expression can also be written like this, if you want to add all the parts a lambda expression can have:

```
(String s1, String s2) -> {
    return s1.compareToIgnoreCase(s2);
}
```

# Anatomy of a lambda expression

If you take a closer look at this expanded version of the lambda expression that implements `Comparator<String>`, you'll see it's not so different from a standard Java method.



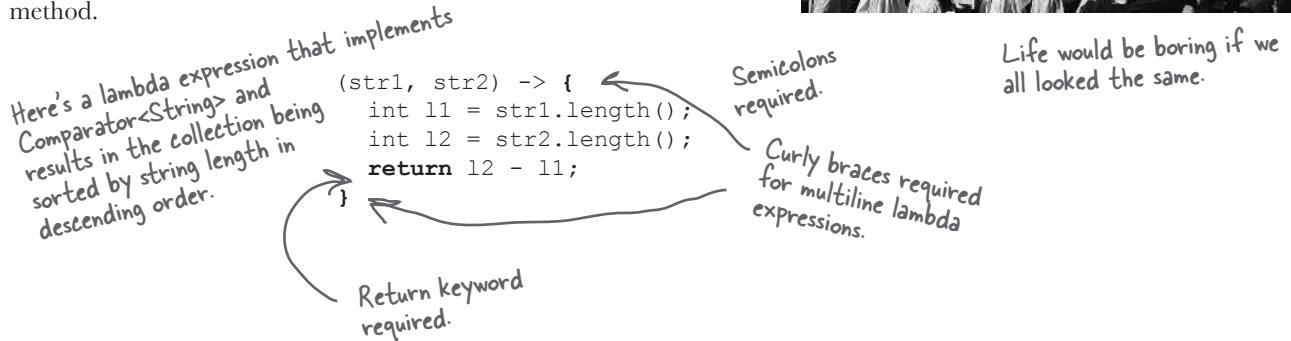
The shape of the lambda (its parameters, return type, and what it can reasonably be expected to do) is dictated by the Functional Interface it implements.

## Variety is the spice of life

Lambda expressions can come in all shapes and sizes, and still conform to the same basic rules that we've seen.

### A lambda might have more than one line

A lambda expression is effectively a method, and can have as many lines as any other method. Multiline lambda expressions **must** be inside curly braces. Then, like any other method code, every line **must** end in a semicolon, and if the method is supposed to return something, the lambda body **must** include the word "return" like any normal method.



### Single-line lambdas don't need ceremony

If your lambda expression is a single line, it makes it much easier for the compiler to guess what's going on. Therefore, we can leave out a lot of the "boilerplate" syntax. If we shrink the lambda expression from the last example into a single line, it looks like this:

No need for curly braces →

```
(str1, str2) -> str2.length() - str1.length()
```

No need for "return"

No semicolons

This is the same Functional Interface (Comparator) and performs the same operation. Whether you use multiline lambdas or single-line lambdas is completely up to you. It will probably depend upon how complicated the logic in the lambda expression is, and how easy you think it is to read—sometimes longer code can be more descriptive.

Later, we'll see another approach for handling long lambda expressions.

## A lambda might not return anything

The Functional Interface's method might be declared void; i.e., it doesn't return anything. In these cases, the code inside the lambda is simply run, and you don't need to return any values from the lambda body.

This is the case for lambda expressions in a **forEach** method.

Multiline lambda

```

Look! No round brackets! We'll
see this again in a minute.
str -> {
    String output = "str = " + str;
    System.out.println(output);
}
No return value
  
```

@FunctionalInterface  
public interface Consumer<T> {  
 void accept(T t);  
}

Method is void on the Functional Interface

## A lambda might have zero, one, or many parameters

The number of parameters the lambda expression needs is dependent upon the number of parameters the Functional Interface's method takes. The parameter types (e.g., the name “String”) are not usually required, but you can add them if you think it makes it easier to understand the code. You may need to add the types if the compiler can't automatically work out which Functional Interface your lambda implements.

If a lambda expression doesn't take any parameters, you need to use empty brackets to show this.

```
(() -> System.out.println("Hello!"))
```

No method parameters

```
@FunctionalInterface
public interface Runnable {
    void run();
}
```

No need for round brackets if it's a single parameter without a type (remember param types are optional)

```
str -> System.out.println(str)
```

One method parameter

```
@FunctionalInterface
public interface Consumer<T> {
    void accept(T t);
}
```

```
(str1, str2) -> str1.compareToIgnoreCase(str2)
```

Two method parameters

```
@FunctionalInterface
public interface Comparator<T> {
    int compare(T o1, T o2);
}
```

## How can I tell if a method takes a lambda?

By now you've seen that lambda expressions are implementations of a functional interface—that is, an Interface with a Single Abstract Method. That means the **type** of a lambda expression is this interface.

Go ahead and create a lambda expression. Instead of passing this into some method, as we have been doing so far, assign it to a variable. You'll see it can be treated just like any other Object in Java, because everything in Java is an Object. The variable's type is the Functional Interface.

```
Comparator<String> comparator = (s1, s2) -> s1.compareToIgnoreCase(s2);  
  
Runnable runnable = () -> System.out.println("Hello!");  
  
Consumer<String> consumer = str -> System.out.println(str);
```

How does this help us see if a method takes a lambda expression? Well, the method's parameter type will be a Functional Interface. Take a look at some examples from the Streams API:

Stream<T> filter(**Predicate**<? super T> predicate)

boolean allMatch(**Predicate**<? super T> predicate)

```
@FunctionalInterface  
public interface Predicate<T>
```

<R> Stream<R> map(**Function**<? super T,? extends R> mapper)

```
@FunctionalInterface  
public interface Function<T,R>
```

void forEach(**Consumer**<? super T> action)

```
@FunctionalInterface  
public interface Consumer<T>
```



Exercise

## BE the Compiler, advanced

Your job is to play compiler and determine which of these statements would compile. But some of this code wasn't covered in the chapter, so you need to work out the answers based on what you DID learn, applying the "rules" to these new situations.

The signatures of the functional interfaces are on the right, for your convenience.

Check the box if the statement would compile.

- Runnable r = () -> System.out.println("Hi!");
- Consumer<String> c = s -> System.out.println(s);
- Supplier<String> s = () -> System.out.println("Some string");
- Consumer<String> c = (s1, s2) -> System.out.println(s1 + s2);
- Runnable r = (String str) -> System.out.println(str);
- Function<String, Integer> f = s -> s.length();
- Supplier<String> s = () -> "Some string";
- Consumer<String> c = s -> "String" + s;
- Function<String, Integer> f = (int i) -> "i = " + i;
- Supplier<String> s = s -> "Some string: " + s;
- Function<String, Integer> f = (String s) -> s.length();

```
public interface Runnable {
    void run();
}
```

```
public interface Consumer<T> {
    void accept(T t);
}
```

```
public interface Supplier<T> {
    T get();
}
```

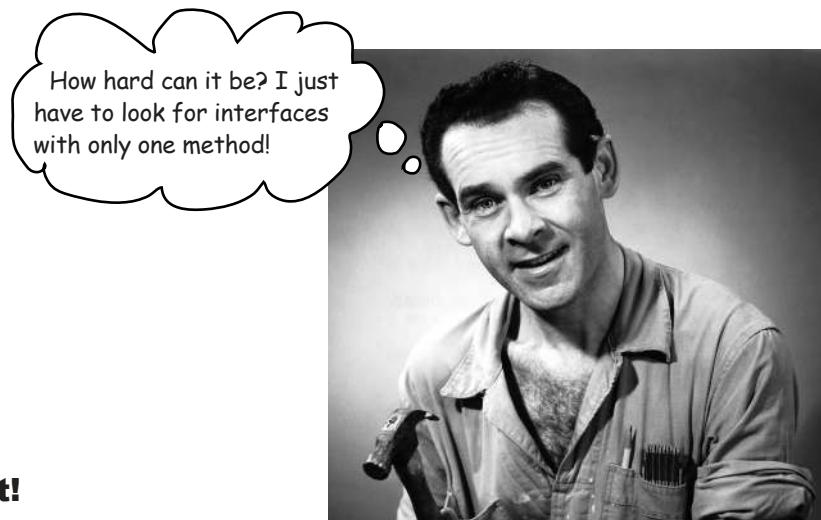
```
public interface Function<T, R> {
    R apply(T t);
}
```

—————> Answers on page 418.

## Spotting Functional Interfaces

So far we've seen Functional Interfaces that are marked with a `@FunctionalInterface` annotation (we'll cover annotations in Appendix B), which conveniently tells us this interface has a Single Abstract Method and can be implemented with a lambda expression.

Not all functional interfaces are tagged this way, particularly in older code, so it's useful to understand how to spot a functional interface for yourself.



### Not so fast!

Originally, the only kind of methods allowed in interfaces were **abstract** methods, methods that need to be *overridden* by any class that *implements* this interface. But as of Java 8, interfaces can also contain **default** and **static** methods.

You saw static methods in Chapter 10, *Numbers Matter*, and you'll see them later in this chapter too. These are methods that don't need to belong to an instance, and are often used as helper methods.

Default methods are slightly different. Remember abstract classes from Chapter 8, *Serious Polymorphism*? They had abstract methods that need to be overridden, and standard methods with a body. On an interface, a default method works a bit like a standard method in an abstract class—they have a body, and will be inherited by subclasses.

Both default and static methods have a method body, with defined behavior. With interfaces, any method that is not defined as **default** or **static** is an abstract method that *must* be overridden.

# Functional interfaces in the wild

Now that we know interfaces can have **non**-abstract methods, we can see there's a bit more of a trick to identifying interfaces with just one abstract method. Take a look at our old friend, Comparator. It has a **lot** of methods! And yet it's still a SAM-type; it has only one Single Abstract Method. It's a Functional Interface we can implement as a lambda expression.

Here it is! This is our Single Abstract Method.

Don't be misled by this method! It's not static or default, but it's not actually abstract either—it's inherited from Object. It does have a method body, defined by the Object class.

| Modifier and Type                                           | Method                                                                    |
|-------------------------------------------------------------|---------------------------------------------------------------------------|
| int                                                         | compare(T o1, T o2)                                                       |
| static <T,U extends Comparable<? super U>><br>Comparator<T> | comparing(Function<? super T,> keyExtractor)                              |
| static <T,U><br>Comparator<T>                               | comparing(Function<? super T,> keyExtractor, Comparator<U> keyComparator) |
| static <T> Comparator<T>                                    | comparingDouble(ToDoubleFunction<? super T> keyExtractor)                 |
| static <T> Comparator<T>                                    | comparingInt(ToIntFunction<? super T> keyExtractor)                       |
| static <T> Comparator<T>                                    | comparingLong(ToLongFunction<? super T> keyExtractor)                     |
| boolean                                                     | equals(Object obj)                                                        |
| static <T extends Comparable<? super T>><br>Comparator<T>   | naturalOrder()                                                            |
| static <T> Comparator<T>                                    | nullsFirst(Comparator<? super T> comparator)                              |
| static <T> Comparator<T>                                    | nullsLast(Comparator<? super T> comparator)                               |
| default Comparator<T>                                       | reversed()                                                                |



Which of these interfaces has a Single Abstract Method and can therefore be implemented as a lambda expression?

| BiPredicate              |                                    |
|--------------------------|------------------------------------|
| Modifier and Type        | Method                             |
| default BiPredicate<T,U> | and(BiPredicate<? super T,> other) |
| default BiPredicate<T,U> | negate()                           |
| default BiPredicate<T,U> | or(BiPredicate<? super T,> other)  |
| boolean                  | test(T t, U u)                     |

| ActionListener    |                                |
|-------------------|--------------------------------|
| Modifier and Type | Method                         |
| void              | actionPerformed(ActionEvent e) |

## Iterator

| Modifier and Type | Method                                       |
|-------------------|----------------------------------------------|
| default void      | forEachRemaining(Consumer<? super E> action) |
| boolean           | hasNext()                                    |
| E                 | next()                                       |
| default void      | remove()                                     |

## Function

| Modifier and Type         | Method                               |
|---------------------------|--------------------------------------|
| default <V> Function<T,V> | andThen(Function<? super R,> after)  |
| R                         | apply(T t)                           |
| default <V> Function<V,R> | compose(Function<? super V,> before) |
| static <T> Function<T,T>  | identity()                           |

## SocketOption

| Modifier and Type | Method |
|-------------------|--------|
| String            | name() |
| Class<T>          | type() |

→ Answers on page 419.

## Lou's back!

Lou's been running his new jukebox management software from the last chapter for some time now, and he wants to learn so much more about the songs played on the diner's jukebox. Now that he has the data, he wants to slice-and-dice it and put it together in a new shape, just as he does with the ingredients of his famous Special Omelette!

He's thinking there are all kinds of information he could learn about the songs that are played, like:

- What are the top five most-played songs?
- What sort of genres are played?
- Are there any songs with the same name by different artists?

We *could* find these things out writing a for loop to look at our song data, performing checks using if statements, and perhaps putting songs, titles, or artists into different collections to find the answers to these questions.

**But now that we know about the Streams API,  
we know there's an easier way....**

Now that I have data about what's been played on my jukebox, I want to know more!



The code on the next page is your **mock** code; calling `Songs.getSongs()` will give you a List of Song objects that you can assume looks just like the real data from Lou's jukebox.



Type in the Ready-Bake Code on the next page, including filling out the rest of the Song class. When you've done that, create a main method that prints out all the songs.

What do you expect the output to look like?



## Ready-Bake Code

Here's an updated "mock" method. It will return some test data that we can use on to try out some of the reports Lou wants to create for the jukebox system. There's also an updated Song class.

```

class Songs {
    public List<Song> getSongs() {
        return List.of(
            new Song("$10", "Hitchhiker", "Electronic", 2016, 183),
            new Song("Havana", "Camila Cabello", "R&B", 2017, 324),
            new Song("Cassidy", "Grateful Dead", "Rock", 1972, 123),
            new Song("50 ways", "Paul Simon", "Soft Rock", 1975, 199),
            new Song("Hurt", "Nine Inch Nails", "Industrial Rock", 1995, 257),
            new Song("Silence", "Delerium", "Electronic", 1999, 134),
            new Song("Hurt", "Johnny Cash", "Soft Rock", 2002, 392),
            new Song("Watercolour", "Pendulum", "Electronic", 2010, 155),
            new Song("The Outsider", "A Perfect Circle", "Alternative Rock", 2004, 312),
            new Song("With a Little Help from My Friends", "The Beatles", "Rock", 1967, 168),
            new Song("Come Together", "The Beatles", "Blues rock", 1968, 173),
            new Song("Come Together", "Ike & Tina Turner", "Rock", 1970, 165),
            new Song("With a Little Help from My Friends", "Joe Cocker", "Rock", 1968, 46),
            new Song("Immigrant Song", "Karen O", "Industrial Rock", 2011, 12),
            new Song("Breathe", "The Prodigy", "Electronic", 1996, 337),
            new Song("What's Going On", "Gaye", "R&B", 1971, 420),
            new Song("Hallucinate", "Dua Lipa", "Pop", 2020, 75),
            new Song("Walk Me Home", "P!nk", "Pop", 2019, 459),
            new Song("I am not a woman, I'm a god", "Halsey", "Alternative Rock", 2021, 384),
            new Song("Pasos de cero", "Pablo Alborán", "Latin", 2014, 117),
            new Song("Smooth", "Santana", "Latin", 1999, 244),
            new Song("Immigrant song", "Led Zeppelin", "Rock", 1970, 484));
    }
}

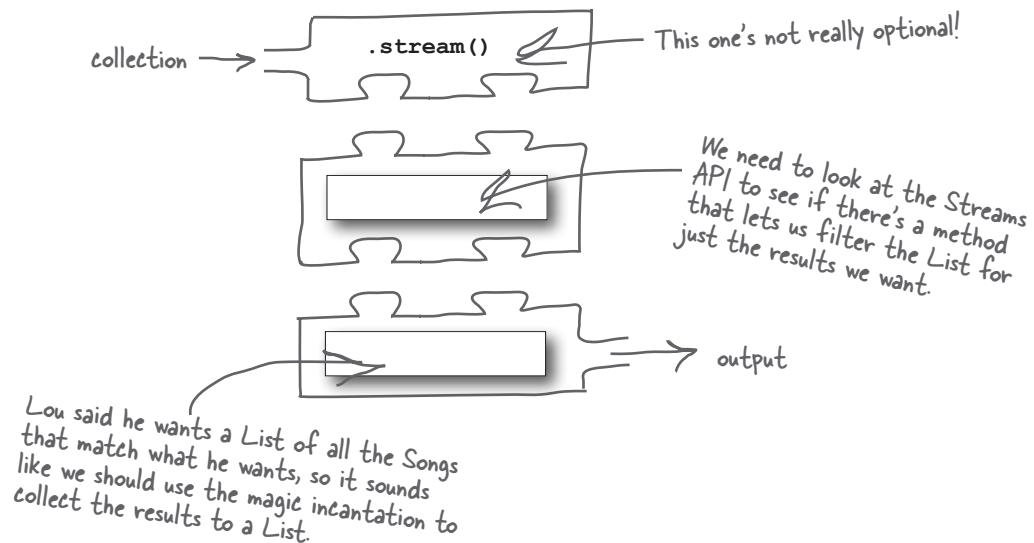
public class Song {
    private final String title;
    private final String artist;
    private final String genre;
    private final int year;
    private final int timesPlayed;
    // Practice for you! Create a constructor, all the getters and a toString()
}

```

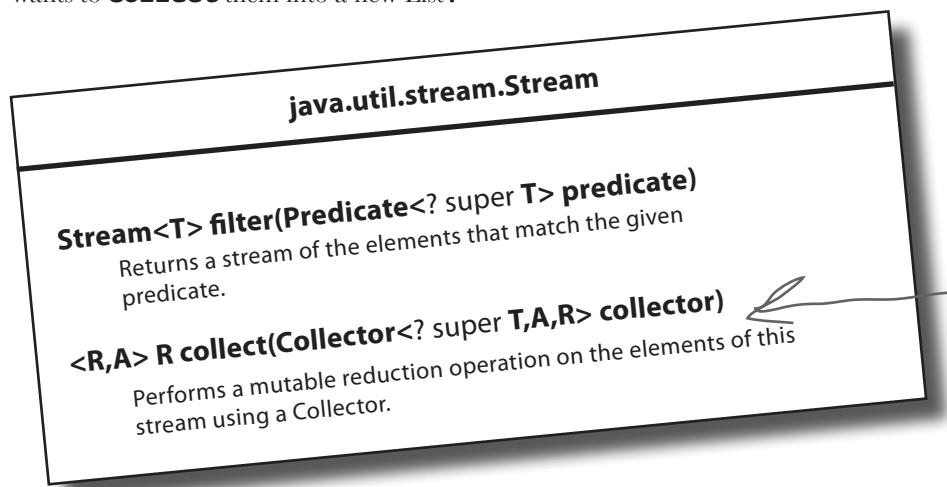
## Lou's Challenge #1: Find all the "rock" songs

The data in the updated song list contains the *genre* of the song. Lou's noticed that the diner's clientele seem to prefer variations on rock music, and he wants to see a list of all the songs that fall under some genre of "rock."

This is the Streams chapter, so clearly the solution is going to involve the Streams API. Remember, there are three types of pieces we can put together to form a solution.



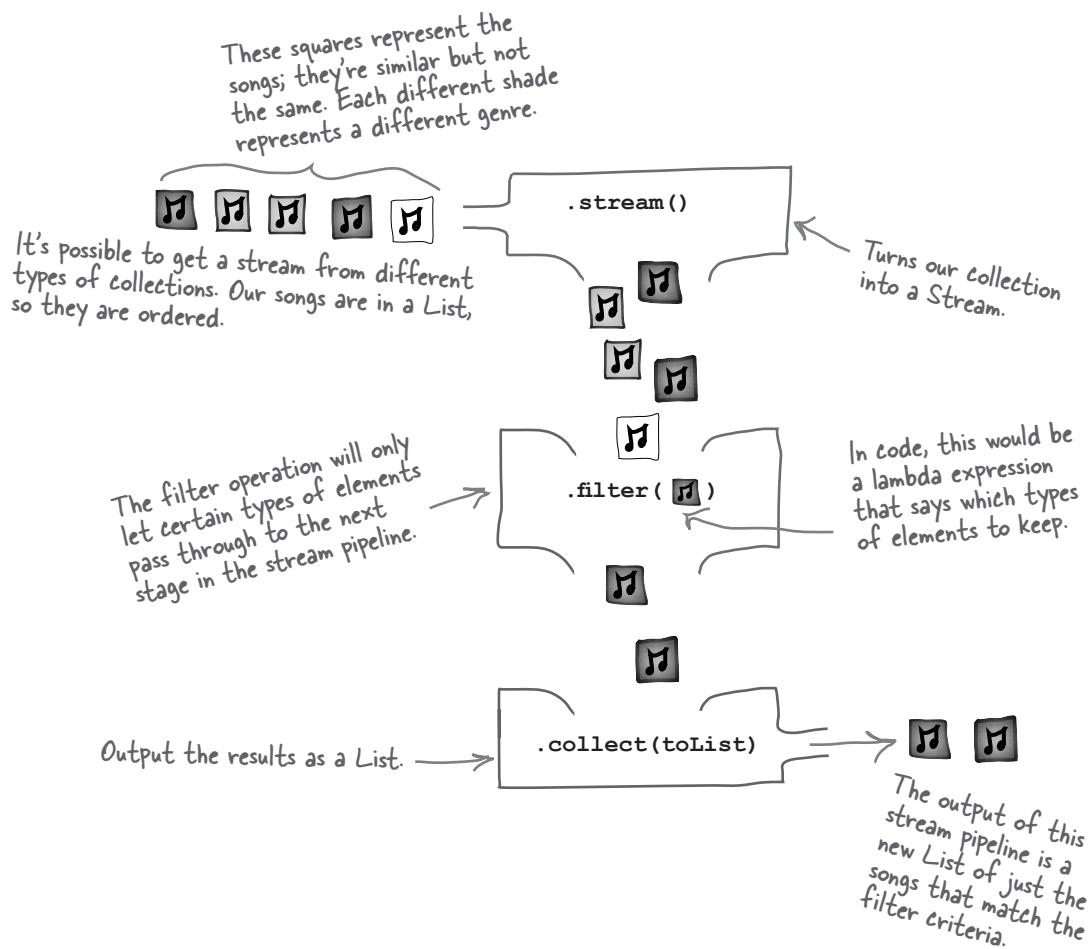
Fortunately, there are hints about how to create a Streams API call based on the requirements Lou gave us: he wants to **filter** for just the Songs with a particular genre, and he wants to **collect** them into a new List.



Remember that for now we're just going to use the "magic incantation" to collect into a List.

# Filter a stream to keep certain elements

Let's see how a filter operation might work on the list of songs.

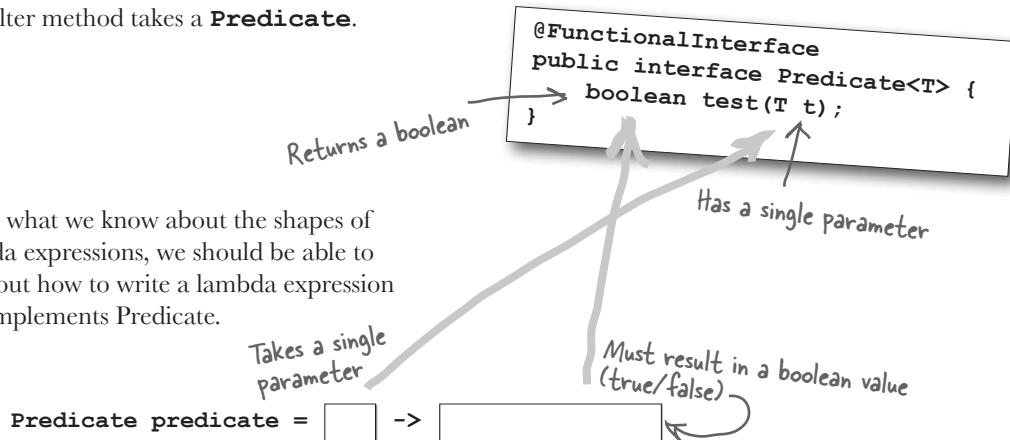


# Let's Rock!

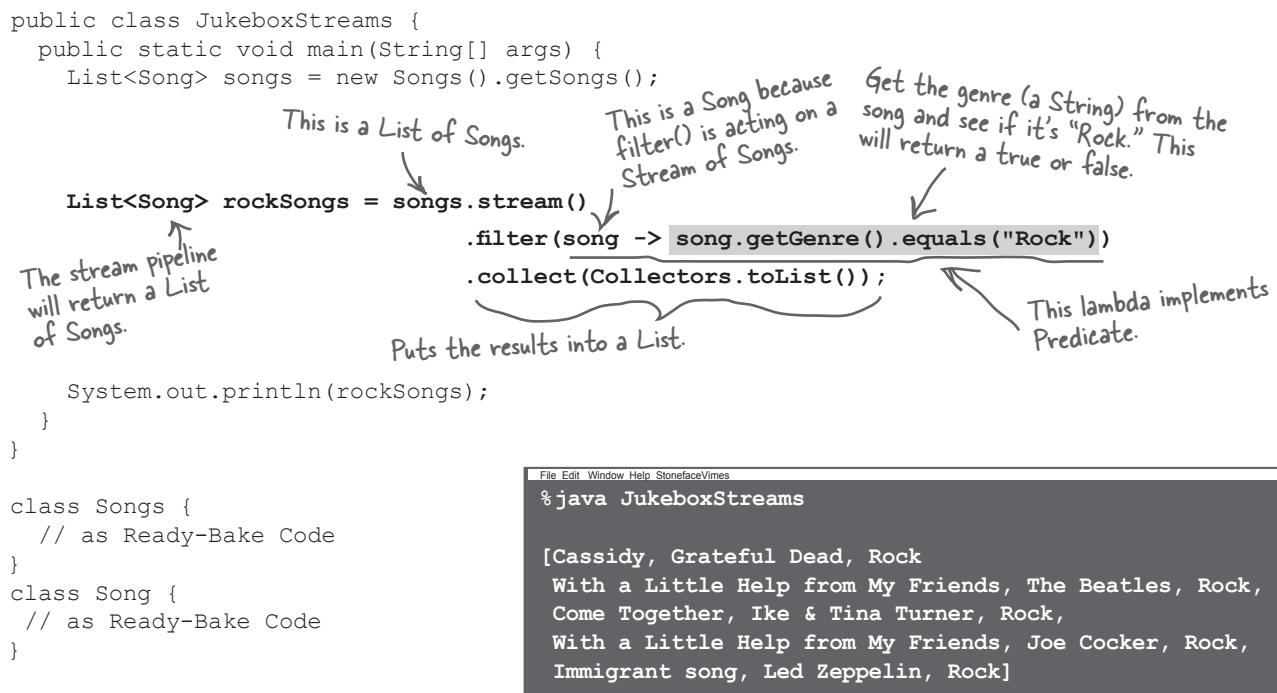
So adding a **filter** operation filters out elements that we don't want, and the stream continues with just the elements that meet our criteria. It should come as no surprise to find that you can use a lambda expression to state which elements we want to keep in the stream.

The filter method takes a **Predicate**.

Given what we know about the shapes of lambda expressions, we should be able to work out how to write a lambda expression that implements Predicate.



We'll know what the type of the single parameter is when we plug it into the Stream operation, since the input type to the lambda will be determined by the types in the stream.



# Getting clever with filters

The **filter** method, with its “simple” true or false return value, can contain sophisticated logic to filter elements in, or out, of the stream. Let’s take our filter one step further and actually do what Lou asked:

*He wants to see a list of all the songs that fall under **some genre** of “rock.”*

He doesn’t want to see just the songs that are classed as “Rock,” but any genre that is kinda Rock-like. We should search for any genre that has the word “Rock” in it somewhere.

There’s a method in String that can help us with this, it’s called **contains**.

```
List<Song> rockSongs = songs.stream()
    .filter(song -> song.getGenre().contains("Rock"))
    .collect(Collectors.toList());
```

Returns true if the genre  
has the word “Rock” in it  
anywhere

```
File Edit Window Help YouRock
%java JukeboxStreams

[Cassidy, Grateful Dead, Rock
 50 ways, Paul Simon, Soft Rock
 Hurt, Nine Inch Nails, Industrial Rock
 Hurt, Johnny Cash, Soft Rock
 ...

```

Now the stream returns different  
types of rock songs.

*Output chopped down to save space  
in the book—save the trees!*



## Brain Barbell

Can you write a filter operation that can select songs:

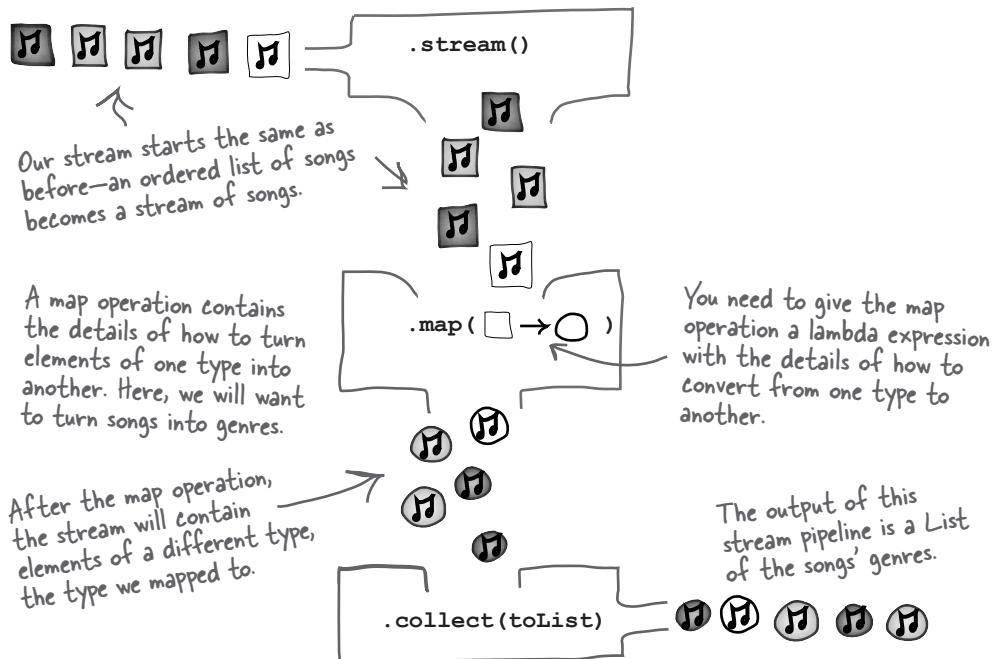
- By The Beatles
- That start with “H”
- More recent than 1995

## Lou's Challenge #2: List all the genres

Lou now senses that the genres of music that the diners are listening to are more complicated than he thought. He wants a list of all the genres of the songs that have been played.

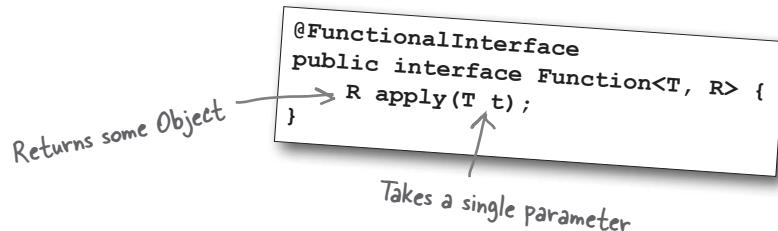
So far, all of our streams have returned the same types that they started with. The earlier examples were Streams of Strings, and returned Lists of Strings. Lou's previous challenge started with a List of Songs and ended up with a (smaller) List of Songs.

Lou now wants a list of genres, which means we need to somehow turn the song elements in the stream into genre (String) elements. This is what **map** is for. The map operation states how to map *from* one type *to* another type.



# Mapping from one type to another

The map method takes a **Function**. The generics by definition are a bit vague, which makes it a little tricky to understand, but Functions do one thing: they take something of one type and return something of a different type. Exactly what's needed for mapping varies from one type to another.



Let's see what it looks like when we use `map` in a stream pipeline.

```

List<String> genres = songs.stream()
    .map(song -> song.getGenre())
    .collect(toList());
  
```

Annotations on the code:

- `The result will be a List of Strings, because genre is a String.`
- `This is a List of Song objects.`
- `A single parameter, a Song because this map() is on a Stream of Songs.`
- `Puts the results into a List`
- `The lambda body can return an object of any type. By calling getGenre on the song, the stream after this point will be a stream of (genre) Strings.`

The map's lambda expression is similar to the one for filter; it takes a song and turns it into something else. Instead of returning a boolean, it returns some other object, in this case a String containing the song's genre.

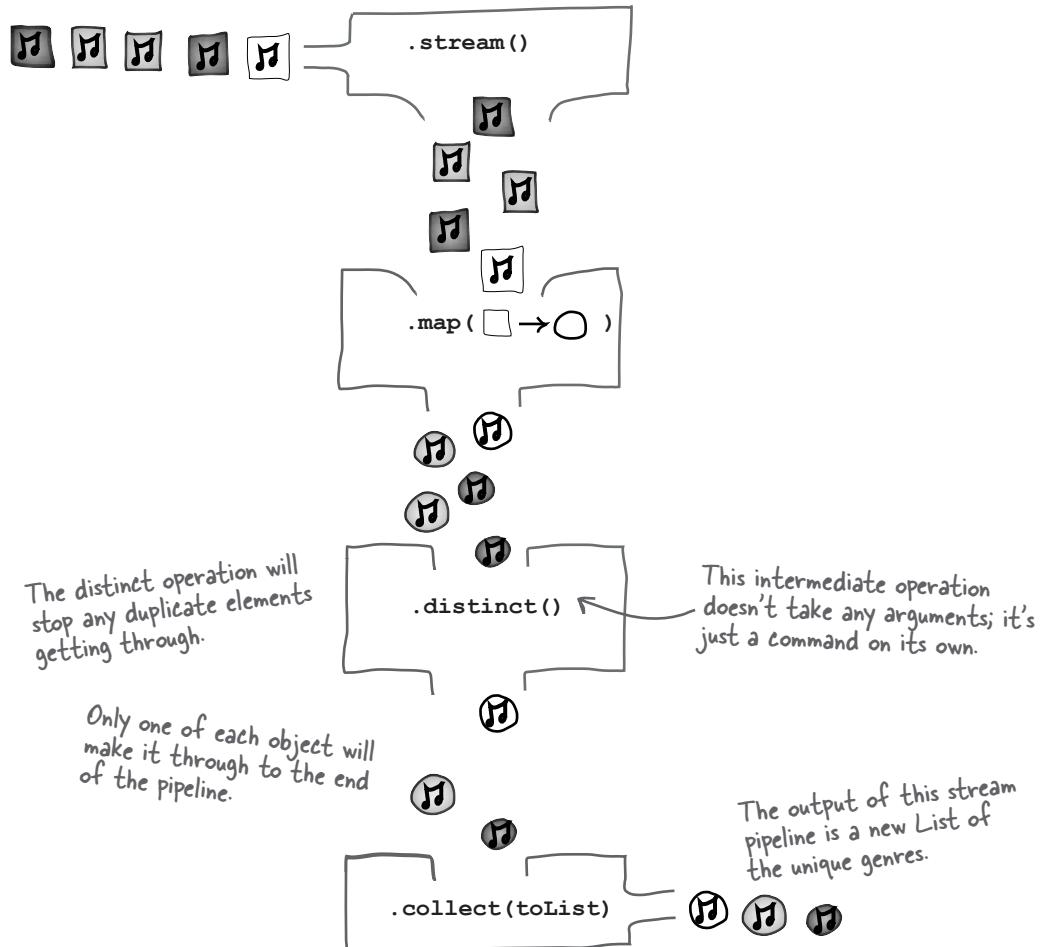
```

File Edit Window Help RoadToNowhere
%java JukeboxStreams

[Electronic, R&B, Rock, Soft Rock, Industrial Rock, Electronic, Soft Rock, Electronic, Alternative Rock, Rock, Blues rock, Rock, Rock, Industrial Rock, Electronic, R&B, Pop, Pop, Alternative Rock, Latin, Latin, Rock]
  
```

## Removing duplicates

We've got a list of all the genres in our test data, but Lou probably doesn't want to wade through all these duplicate genres. The `map` operation on its own will result in an output List that's the same size as the input List. Since stream operations are designed to be stacked together, perhaps there's another operation we can use to get just one of every element in the stream?



# Only one of every genre

All we need to do is to add a distinct operation to the stream pipeline, and we'll get just one of each genre.

```
List<String> genres = songs.stream()
    .map(song -> song.getGenre())
    ↗ .distinct()
    .collect(Collectors.toList());
```

Having this in the stream pipeline means there will be no duplicates after this point

Outputs a much more readable list of all the genres

```
File Edit Window Help UniqueIsGood
% java JukeboxStreams
[Electronic, R&B, Rock, Soft Rock,
Industrial Rock, Alternative Rock,
Blues rock, Pop, Latin]
```

# Just keep building!

A stream pipeline can have any number of intermediate operations. The power of the Streams API is that we can build up complex queries with understandable building blocks. The library will take care of running this in a way that is as efficient as possible. For example, we could create a query that returns a list of all the artists that have covered a specific song, excluding the original artists, by using a map operation and multiple filters.

```
String songTitle = "With a Little Help from My Friends";
List<String> result = allSongs.stream()
    .filter(song -> song.getTitle().equals(songTitle))
    .map(song -> song.getArtist())
    .filter(artist -> !artist.equals("The Beatles"))
    .collect(Collectors.toList());
```



## Sharpen your pencil

Try annotating this code yourself.  
What do each of the filters do?  
What does the map do?

→ Yours to solve.

# Sometimes you don't even need a lambda expression

Some lambda expressions do something simple and predictable, given the type of the parameter or the shape of the functional interface. Look again at the lambda expression for the `map` operation.

```
Function<Song, String> getGenre = song -> song.getGenre();
```

Instead of spelling this whole thing out, you can point the compiler to a method that does the operation we want, using a **method reference**.

```
Function<Song, String> getGenre = song -> song.getGenre();
```

The input parameter to this Function is a Song.

The output of this Function needs to be a String.

The output of getGenre() is a String, just like the Function needs.

This is the method call in the lambda body.

A method reference—instead of using a “.” that would cause the compiler to call the method, use a “::” to point the compiler in the direction of the method.

Method references can replace lambda expressions in a number of different cases. Generally, we might use a method reference if it makes the code easier to read.

Take our old friend the `Comparator`, for example. There are a lot of helper methods on the `Comparator` interface that, when combined with a method reference, let you see which value is being used for sorting and in which direction. Instead of doing this, to order the songs from oldest to newest:

```
List<Song> result = allSongs.stream()
    .sorted(o1, o2) -> o1.getYear() - o2.getYear())
    .collect(toList());
```

Use a method reference combined with a `static` helper method from `Comparator` to state what the comparison should be:

```
List<Song> result = allSongs.stream()
    .sorted(Comparator.comparingInt(Song::getYear))
    .collect(toList());
```



You don't need to use method references if you don't feel comfortable with them. What's important is to be able to recognize the “::” syntax, especially in a stream pipeline.

Method references can replace lambda expressions, but you don't have to use them.

Sometimes method references make the code easier to understand.

# Collecting results in different ways

While `Collectors.toList` is the most commonly used Collector, there are other useful Collectors. For example, instead of using `distinct` to solve the last challenge, we could collect the results into a Set, which does not allow duplicates. The advantage of using this approach is that anything else that uses the results knows that because it's a Set, *by definition* there will be no duplicates.

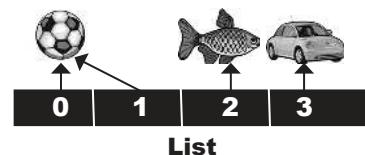
```
Set<String> genres = songs.stream()
    .map(song -> song.getGenre())
    .collect(Collectors.toSet());
```

Store the results in a Set of Strings, not a List. Sets cannot contain duplicates.

Put the results into a Set, which will automatically only have unique entries.

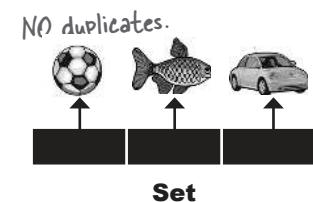
## Collectors.toList and Collectors.toUnmodifiableList

You've already seen `toList`. Alternatively, you can get a List that can't be changed (no elements can be added, replaced or removed) by using `Collectors.toUnmodifiableList` instead. This is only available from Java 10 onward.



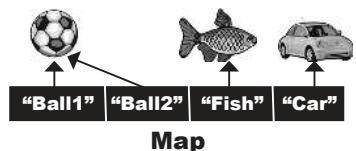
## Collectors.toSet and Collectors.toUnmodifiableSet

Use these to put the results into a Set, rather than a List. Remember that a Set cannot contain duplicates, and is not usually ordered. If you're using Java 10 or higher, you can use `Collectors.toUnmodifiableSet` if you want to make sure your results aren't changed by anything.



## Collectors.toMap and Collectors.toUnmodifiableMap

You can collect your stream into a Map of key/value pairs. You will need to provide some functions to tell the collector what will be the key and what will be the value. You can use `Collectors.toUnmodifiableMap` to create a map that can't be changed, from Java 10 onward.



## Collectors.joining

You can create a String result from the stream. It will join together all the stream elements into a single String. You can optionally define the *delimiter*, the character to use to separate each element. This can be very useful if you want to turn your stream into a String of Comma Separated Values (CSV).

## But wait, there's more!

Collecting the results is not the only game in town; `collect` is just one of many terminal operations.

### Checking if something exists

You can use terminal operations that return a boolean value to look for certain things in the stream. For example, we can see if any R&B songs have been played in the diner.

```
boolean result =
    songs.stream()
        .anyMatch(s -> s.getGenre().equals("R&B"));
```

```
boolean anyMatch(Predicate p);
boolean allMatch(Predicate p);
boolean noneMatch(Predicate p);
```

### Find a specific thing

Terminal operations that return an `Optional` value look for certain things in the stream. For example, we can find the first song played that was released in 1995.

```
Optional<Song> result =
    songs.stream()
        .filter(s -> s.getYear() == 1995)
        .findFirst();
```

```
Optional<T> findAny();
Optional<T> findFirst();
Optional<T> max(Comparator c);
Optional<T> min(Comparator c);
Optional<T> reduce(BinaryOperator a);
```

### Count the items

There's a count operation that you can use to find out the number of elements in your stream. We could find the number of unique artists, for example.

```
long result =
    songs.stream()
        .map(Song::getArtist)
        .distinct()
        .count();
```

```
long count();
```

There are even more terminal operations, and some of them depend upon the type of Stream you're working with.

Remember, the API documentation can help you figure out if there's a built-in operation that does what you want.



Wait a minute. How can a result be "Optional"? What does that even mean?

## Well, some operations may return something, or may not return anything at all

It might seem weird that a method *may* or *may not* return a value, but it happens all the time in real life.

Imagine you're at an ice-cream stand, and you ask for strawberry ice cream.

Strawberry ice cream, please!

Here you go!

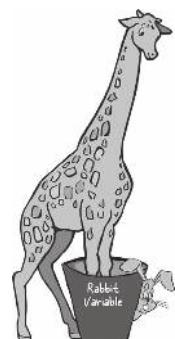
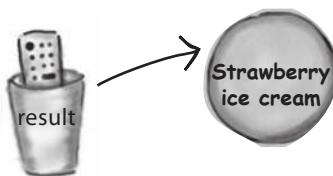
```
IceCream iceCream =  
    getIceCream("Strawberry");
```

Easy, right? But what if they don't have any strawberry? The ice-cream person is likely to tell you "we don't have that flavor."

We don't have any, sorry.

It's then up to you what you do next—perhaps order chocolate instead, find another ice-cream place, or maybe just go home and sulk about your lack of ice cream.

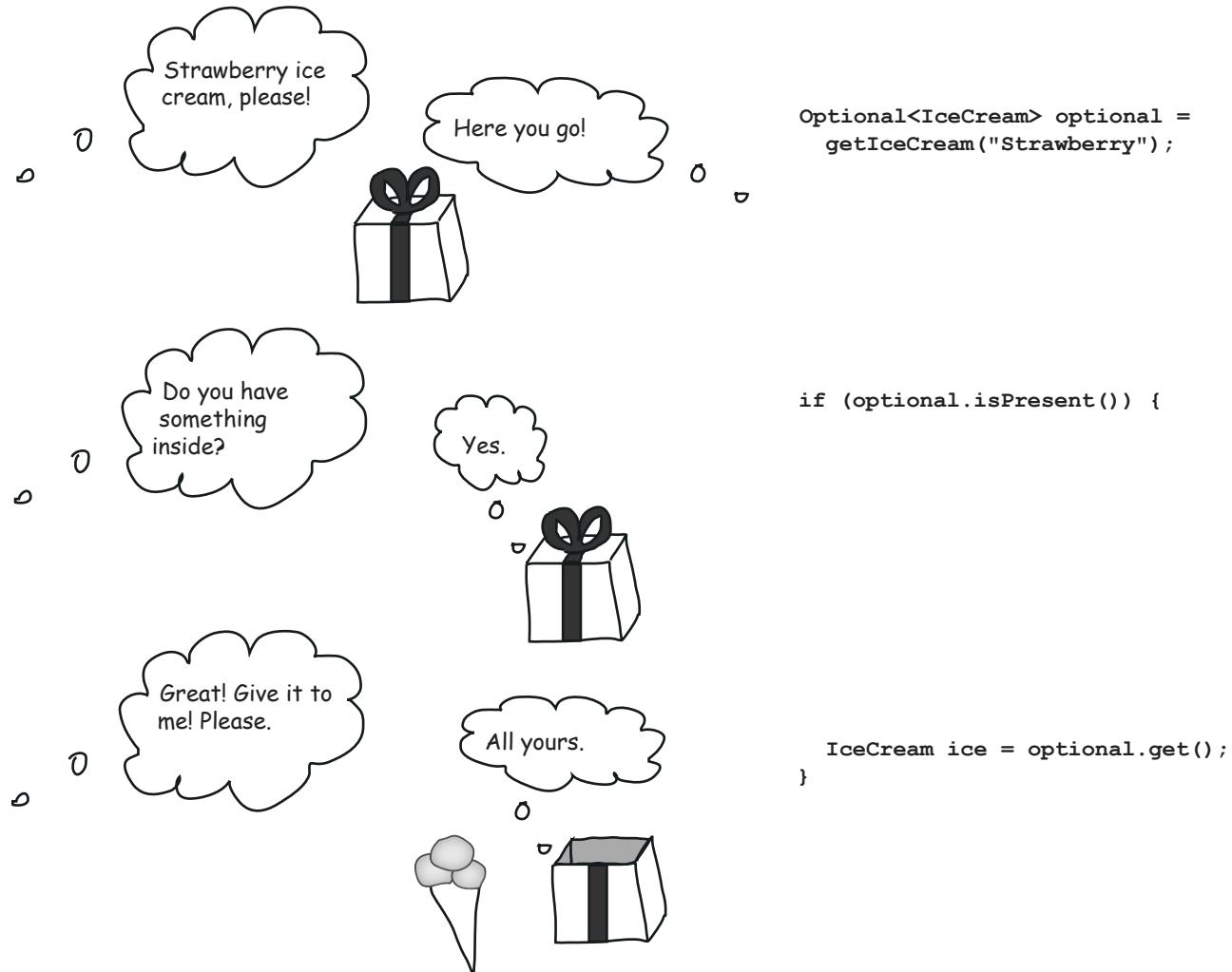
Imagine trying to do this in the Java world. In the first example, you get an ice-cream instance. In the second, you get...a String message? But a message doesn't fit into an ice-cream-shaped variable. A null? But what does null really mean?



optional

## Optional is a wrapper

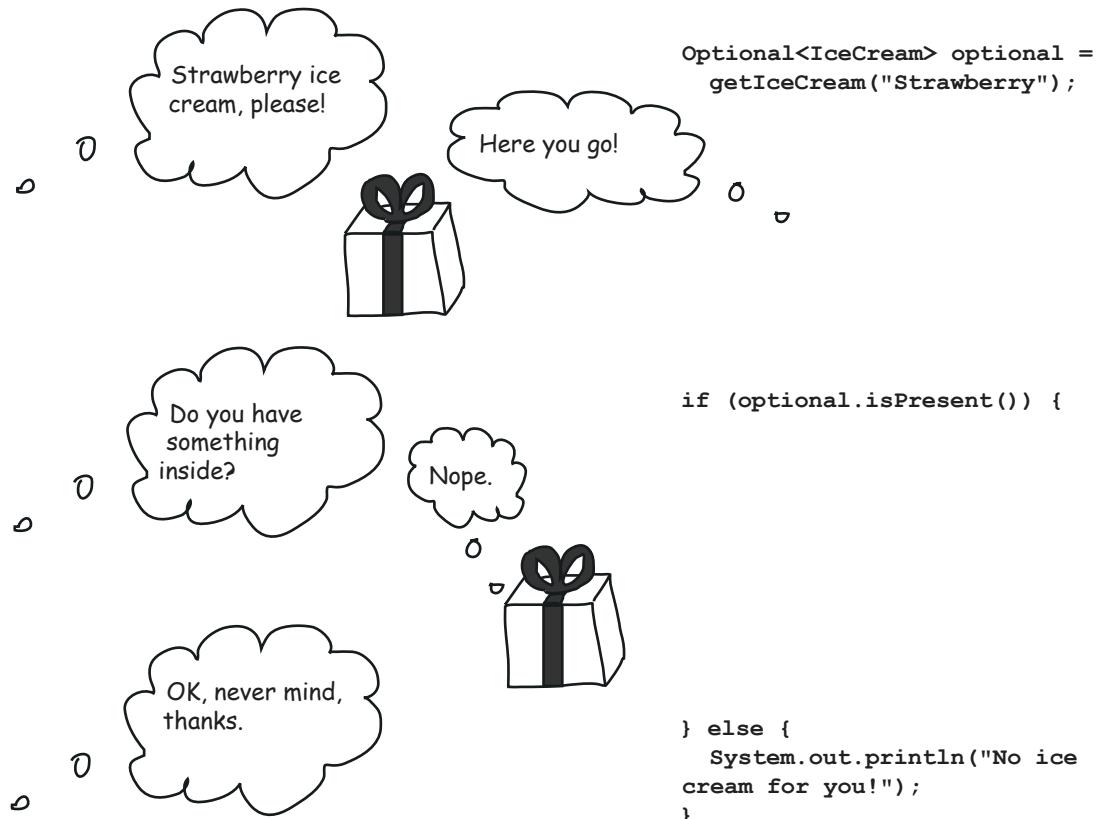
Since Java 8, the normal way for a method to declare that *sometimes it might not return a result* is to return an **Optional**. This is an object that *wraps* the result, so you can ask “Did I get a result? Or is it empty?” Then you can make a decision about what to do next.





## Yes, but now we have a way to ask if we have a result

Optional gives us a way to find out about, and deal with, the times when you don't get an ice cream.

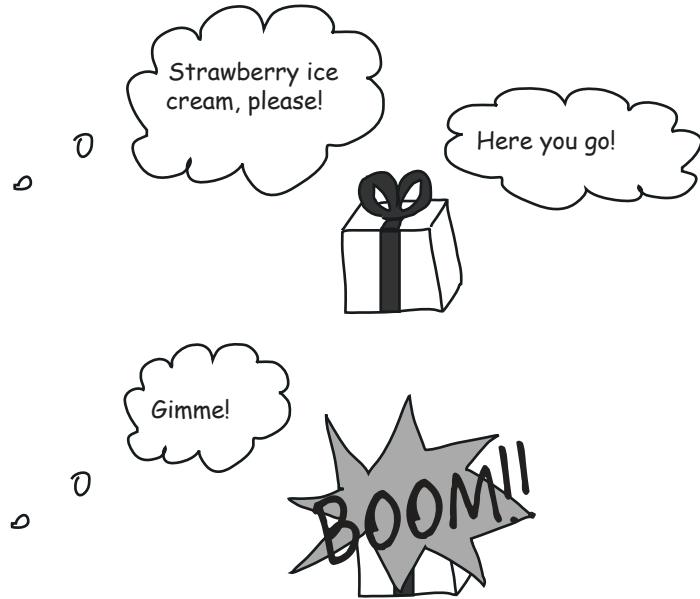


In the past, methods might have thrown Exceptions for this case, or return “null”, or a special type of “Not Found” ice-cream instance. Returning an *Optional* from a method makes it really clear that anything calling the method **needs** to check if there’s a result first, and **then** make their own decision about what to do if there isn’t one.

optional

## Don't forget to talk to the Optional wrapper

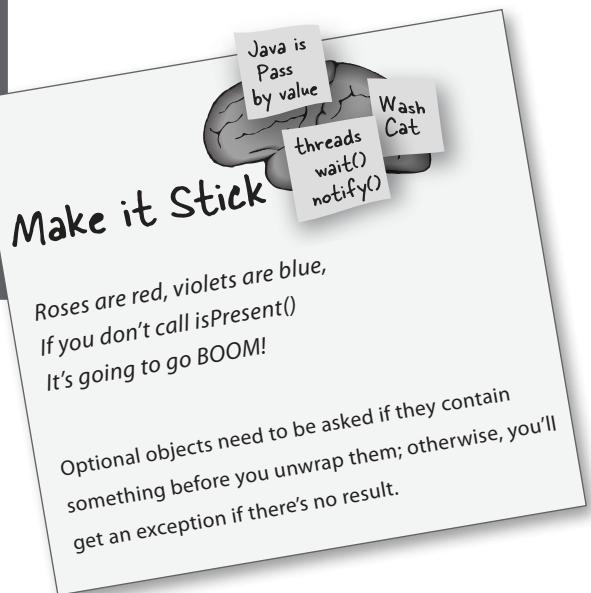
The important thing about Optional results is that **they can be empty**. If you don't check first to see if there's a value present and the result is empty, you will get an exception.



```
Optional<IceCream> optional =  
    getIceCream("Strawberry");
```

```
IceCream ice = optional.get();
```

```
File Edit Window Help Boom  
%java OptionalExamples  
  
Exception in thread "main" java.util.No-  
SuchElementException: No value present  
    at java.base/java.util.Optional.  
get(Optional.java:148)  
    at ch10c.OptionalExamples.  
main(OptionalExamples.java:11)
```





## The Unexpected Coffee

Alex was programming her mega-ultra-clever (Java-powered) coffee machine to give her the types of coffee that suited her best at different times of day.

### Five-Minute Mystery

In the afternoons, Alex wanted the machine to give her the weakest coffee it had available (she had enough to keep her up at night; she didn't need caffeine adding to her problems!). As an experienced software developer, she knew the Streams API would give her the best stream of coffee at the right time.



The coffees would automatically be sorted from the weakest to the strongest using natural ordering, so she gave the coffee machine these instructions:

```
Optional<String> afternoonCoffee = coffees.stream()
    .map(Coffee::getName)
    .sorted()
    .findFirst();
```

The very next day, she asked for an afternoon coffee. To her horror, the machine presented her with an Americano, not the Decaf Cappuccino she was expecting.

“I can’t drink that!! I’ll be up all night worrying about my latest software project!”

*What happened? Why did the coffee machine give Alex an Americano?*

→ Answers on page 419.

## puzzle: Pool Puzzle



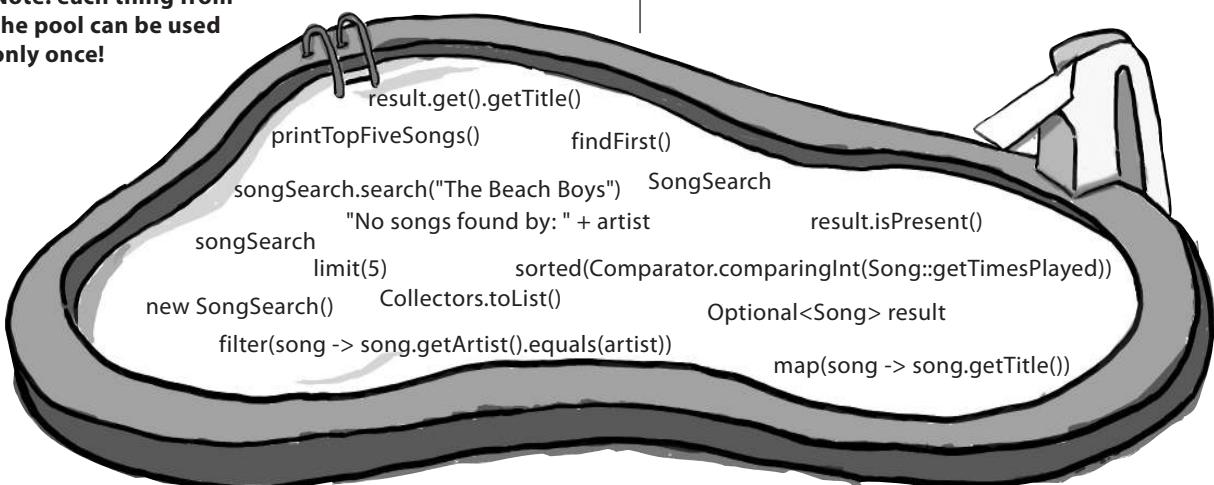
### Pool Puzzle

Your **job** is to take code snippets from the pool and place them into the blank lines in the code. You may **not** use the same snippet more than once, and you won't need to use all the snippets. Your **goal** is to make a class that will compile and run and produce the output listed.

#### Output

```
File Edit Window Help DiveIn
%java StreamPuzzle
[Immigrant Song, With a Little
Help from My Friends, Hallucinate,
Pasos de cero, Cassidy]
With a Little Help from My Friends
No songs found by: The Beach Boys
```

Note: each thing from the pool can be used only once!



```
public class StreamPuzzle {
    public static void main(String[] args) {
        SongSearch songSearch = _____;
        songSearch._____.
            _____ .search("The Beatles");
        _____;
    }
    class _____ {
        private final List<Song> songs =
            new JukeboxData.Songs() .getSongs();

        void printTopFiveSongs() {
            List<String> topFive = songs.stream()
                .
                .
                .
                .
                .
                collect(_____);
            System.out.println(topFive);
        }

        void search(String artist) {
            _____ = songs.stream()
                .
                .
                .
                if (_____ ) {
                    System.out.println(_____ );
                } else {
                    System.out.println(_____ );
                }
            }
        }
    }
}
```



## Mixed Messages (from page 372)

**Candidates:**

```
for (int i = 1; i < nums.size(); i++)
    output += nums.get(i) + " ";
```

```
for (Integer num : nums)
    output += num + " ";
```

```
for (int i = 0; i <= nums.length; i++)
    output += nums.get(i) + " ";
```

```
for (int i = 0; i <= nums.size(); i++)
    output += nums.get(i) + " ";
```

**Possible output:**

1 2 3 4 5

Compiler error

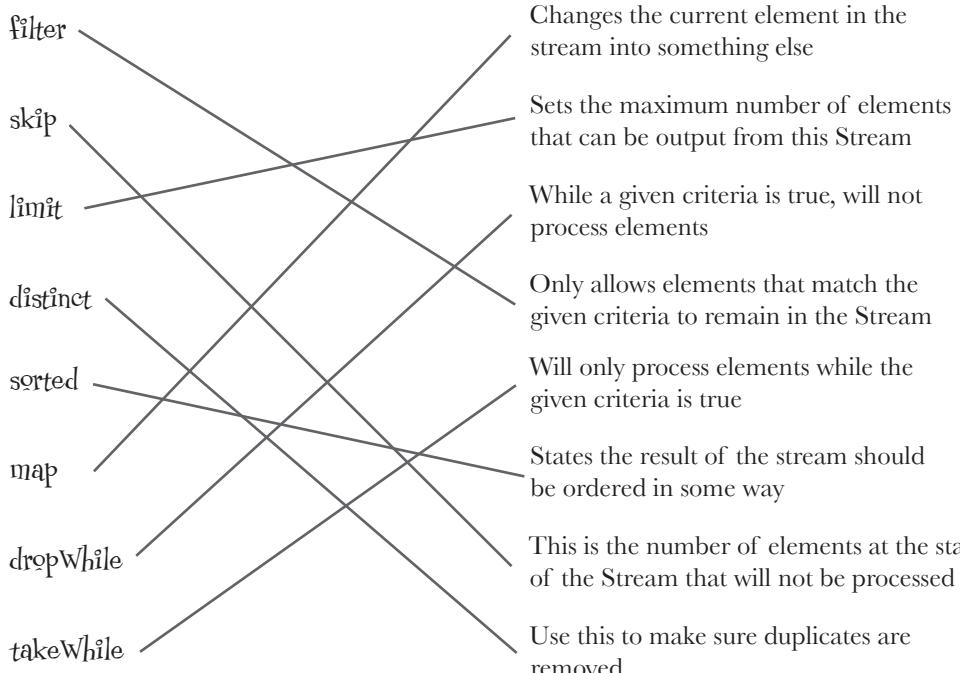
2 3 4 5

Exception thrown

[1, 2, 3, 4, 5]  
[1, 2, 3, 4, 5]  
[1, 2, 3, 4, 5]  
[1, 2, 3, 4, 5]  
[1, 2, 3, 4, 5]



## WHO DOES WHAT? (from page 374)





## Code Magnets (from page 386)

```

import java.util.*;
import java.util.stream.*;

public class CoffeeOrder {
    public static void main(String[] args) {
        List<String> coffees = List.of("Cappuccino",
   "Americano", "Espresso", "Cortado", "Mocha",
   "Cappuccino", "Flat White", "Latte");

        List<String> coffeesEndingInO = coffees.stream()
            .filter(s -> s.endsWith("o"))
            .sorted()
            .distinct()
            .collect(Collectors.toList());

        System.out.println(coffeesEndingInO);
    }
}

```

What would happen if the stream operations were in a different order? Does it matter?

File Edit Window Help Cafelito  
% java CoffeeOrder  
[Americano, Cappuccino,  
Cortado, Espresso]

## BE the Compiler (from page 395)

- Runnable r = () -> System.out.println("Hi!");
- Consumer<String> c = s -> System.out.println(s); *Should return a String but doesn't*
- Supplier<String> s = () -> System.out.println("Some string"); *Should take only one parameter but has two*
- Consumer<String> c = (s1, s2) -> System.out.println(s1 + s2); *Should not have parameters*
- Runnable r = (String str) -> System.out.println(str); *Should not have parameters*
- Function<String, Integer> f = s -> s.length(); *This single-line lambda effectively returns a String*
- Supplier<String> s = () -> "Some string"; *when a consumer method should return nothing. Even though there's no "return," this calculated String value is assumed to be the returned value.*
- Consumer<String> c = s -> "String" + s; *Should have a String parameter and return an int, but instead it has an int param and returns a String*
- Function<String, Integer> f = (int i) -> "i = " + i; *Should not have any parameters*
- Supplier<String> s = s -> "Some string: " + s; *Should take a String parameter. Should return an int, but actually returns nothing.*
- Function<String, Integer> f = () -> System.out.println("Some string"); *Should return an int, but actually returns nothing.*



| BiPredicate       |                                                                      |
|-------------------|----------------------------------------------------------------------|
| Modifier and Type | Method                                                               |
|                   | default BiPredicate<T,U> and(BiPredicate<? super T,? super U> other) |
|                   | default BiPredicate<T,U> negate()                                    |
|                   | default BiPredicate<T,U> or(BiPredicate<? super T,? super U> other)  |
| boolean           | test(T t, U u)                                                       |

Has a Single Abstract Method, test(). The others are all default methods.

| ActionListener    |                                |
|-------------------|--------------------------------|
| Modifier and Type | Method                         |
| void              | actionPerformed(ActionEvent e) |

Has a Single Abstract Method

| Iterator          |                                              |
|-------------------|----------------------------------------------|
| Modifier and Type | Method                                       |
| default void      | forEachRemaining(Consumer<? super E> action) |
| boolean           | hasNext()                                    |
| E                 | next()                                       |
| default void      | remove()                                     |

Has TWO abstract methods, hasNext() and next()

| Function                  |                                                 |
|---------------------------|-------------------------------------------------|
| Modifier and Type         | Method                                          |
| default <V> Function<T,V> | andThen(Function<? super R,? extends V> after)  |
| R                         | apply(T t)                                      |
| default <V> Function<V,R> | compose(Function<? super V,? extends T> before) |
| static <T> Function<T,T>  | identity()                                      |

Has a Single Abstract Method, apply(). The others are default and static methods.

| SocketOption      |        |
|-------------------|--------|
| Modifier and Type | Method |
| String            | name() |
| Class<T>          | type() |

Has two abstract methods

## Five-Minute Mystery (from page 415)



Alex didn't pay attention to the order of the stream operations. She first mapped the coffee objects to a stream of Strings, and then ordered that. Strings are naturally ordered alphabetically, so when the coffee machine got the "first" of these results for Alex's afternoon coffee, it was brewing a fresh "Americano."

If Alex wanted to order the coffees by strength, with the weakest (1 out of 5) first, she needed to order the stream of coffees first, before mapping it to a String name,

```
afternoonCoffee = coffees.stream()
    .sorted()
    .map(Coffee::getName)
    .findFirst();
```

Then the coffee machine will brew her a decaf instead of an Americano.



## Poōl Puzzle (from page 416)

```
public class StreamPuzzle {  
    public static void main(String[] args) {  
        SongSearch songSearch = new SongSearch();  
        songSearch.printTopFiveSongs();  
        songSearch.search("The Beatles");  
        songSearch.search("The Beach Boys");  
    }  
}  
  
class SongSearch {  
    private final List<Song> songs =  
        new JukeboxData.Songs().getSongs();  
  
    void printTopFiveSongs() {  
        List<String> topFive = songs.stream()  
            .sorted(Comparator.comparingInt(Song::getTimesPlayed))  
            .map(song -> song.getTitle())  
            .limit(5)  
            .collect(Collectors.toList());  
        System.out.println(topFive);  
    }  
    void search(String artist) {  
        Optional<Song> result = songs.stream()  
            .filter(song -> song.getArtist().equals(artist))  
            .findFirst();  
        if (result.isPresent()) {  
            System.out.println(result.get().getTitle());  
        } else {  
            System.out.println("No songs found by: " + artist);  
        }  
    }  
}
```

# Risky Behavior



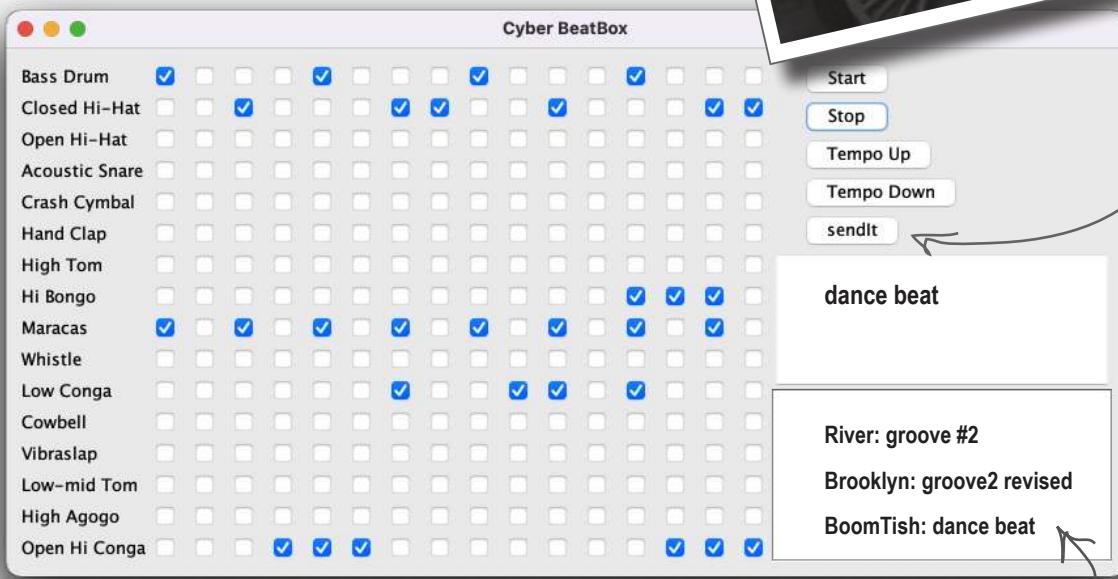
**Stuff happens. The file isn't there. The server is down.** No matter how good a programmer you are, you can't control everything. Things can go wrong. *Very* wrong. When you write a risky method, you need code to handle the bad things that might happen. But how do you *know* when a method is risky? And where do you put the code to *handle* the **exceptional** situation? So far in this book, we haven't *really* taken any risks. We've certainly had things go wrong at runtime, but the problems were mostly flaws in our own code. Bugs. And those we should fix at development time. No, the problem-handling code we're talking about here is for code that you *can't* guarantee will work at runtime. Code that expects the file to be in the right directory, the server to be running, or the Thread to stay asleep. And we have to do this *now*. Because in *this* chapter, we're going to build something that uses the risky JavaSound API. We're going to build a MIDI Music Player.

# Let's make a Music Machine

Over the next three chapters, we'll build a few different sound applications, including a BeatBox Drum Machine. In fact, before the book is done, we'll have a multiplayer version so you can send your drum loops to another player, kind of like sharing over social media. You're going to write the whole thing, although you can choose to use Ready-Bake Code for the GUI parts. OK, so not every IT department is looking for a new BeatBox server, but we're doing this to learn more about Java. Building a BeatBox is just a way to have fun *while* we're learning Java.

The finished BeatBox looks something like this:

You make a beatbox loop (a 16-beat drum pattern) by putting check marks in the boxes.



Your message gets sent to the other players, along with your current beat pattern, when you hit "sendIt."

Incoming messages from players. Click one to load the pattern that goes with it, and then click Start to play it.

Notice the check marks in the boxes for each of the 16 “beats.” For example, on beat 1 (of 16) the Bass drum and the Maracas will play, on beat 2 nothing, and on beat 3 the Maracas and Closed Hi-Hat...you get the idea. When you hit Start, it plays your pattern in a loop until you hit Stop. At any time, you can “capture” one of your own patterns by sending it to the BeatBox server (which means any other players can listen to it). You can also load any of the incoming patterns by clicking on the message that goes with it.

# We'll start with the basics

Obviously we've got a few things to learn before the whole program is finished, including how to build a GUI, how to *connect* to another machine via networking, and a little I/O so we can *send* something to the other machine.

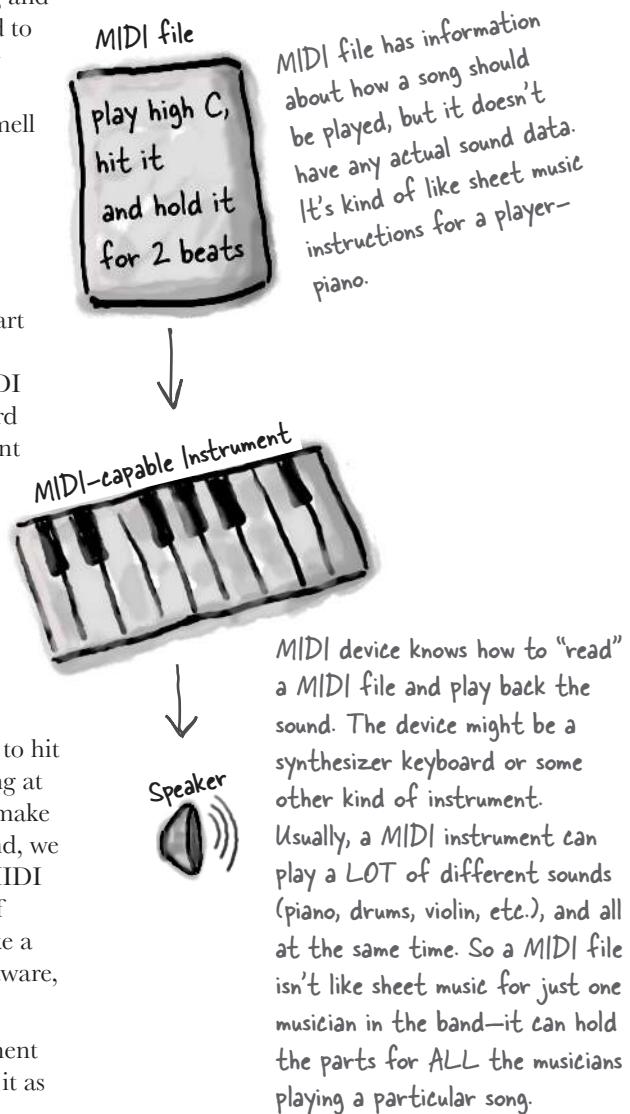
Oh yeah, and the JavaSound API. *That's* where we'll start in this chapter. For now, you can forget the GUI, forget the networking and the I/O, and focus only on getting some MIDI-generated sound to come out of your computer. And don't worry if you don't know a thing about MIDI or a thing about reading or making music. Everything you need to learn is covered here. You can almost smell the record deal.

## The JavaSound API

JavaSound is a collection of classes and interfaces added to Java way back in version 1.3. These aren't special add-ons; they're part of the standard Java SE class library. JavaSound is split into two parts: MIDI and Sampled. We use only MIDI in this book. MIDI stands for Musical Instrument Digital Interface, and is a standard protocol for getting different kinds of electronic sound equipment to communicate. But for our BeatBox app, you can think of MIDI as *a kind of sheet music* that you feed into some device like a high-tech "player piano." In other words, MIDI data doesn't actually include any *sound*, but it does include the *instructions* that a MIDI-reading instrument can play back. Or for another analogy, you can think of a MIDI file like an HTML document, and the instrument that renders the MIDI file (i.e., *plays* it) is like the web browser.

MIDI data says *what* to do (play middle C, and here's how hard to hit it, and here's how long to hold it, etc.), but it doesn't say anything at all about the actual *sound* you hear. MIDI doesn't know how to make a flute, piano, or Jimi Hendrix guitar sound. For the actual sound, we need an instrument (a MIDI device) that can read and play a MIDI file. But the device is usually more like an *entire band or orchestra* of instruments. And that instrument might be a physical device, like a keyboard, or it could even be an instrument built entirely in software, living in your computer.

For our BeatBox, we use only the built-in, software-only instrument that you get with Java. It's called a *synthesizer* (some folks refer to it as a *software synth*) because it *creates* sound. Sound that you *hear*.



but it looked so simple

## First we need a Sequencer

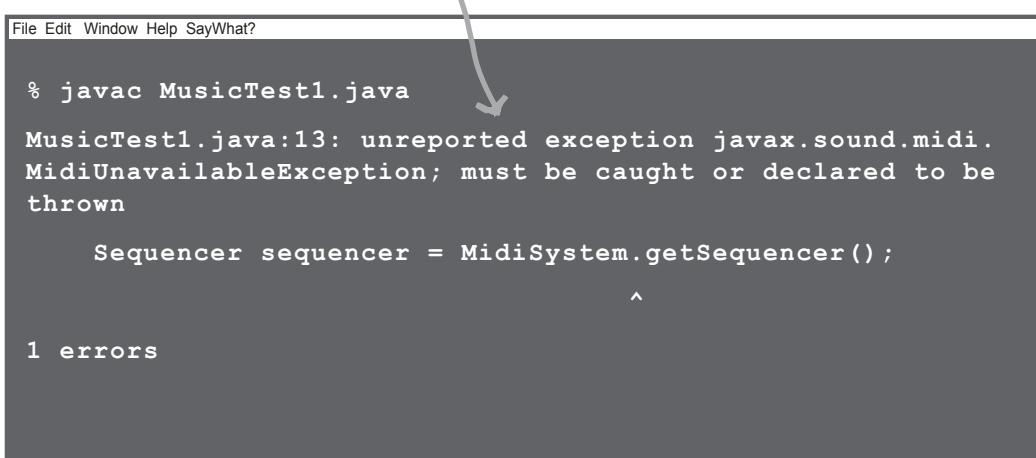
Before we can get any sound to play, we need a Sequencer object. The sequencer is the object that takes all the MIDI data and sends it to the right instruments. It's the thing that *plays* the music. A sequencer can do a lot of different things, but in this book, we're using it strictly as a playback device. It's like a device that streams music, but with a few added features. The Sequencer class is in the javax.sound.midi package. So let's start by making sure we can make (or get) a Sequencer object.

```
import javax.sound.midi.*;  
    ← import the javax.sound.midi package  
  
public class MusicTest1 {  
    public void play() {  
        try {  
            Sequencer sequencer = MidiSystem.getSequencer();  
            System.out.println("Successfully got a sequencer");  
        }  
    }  
  
    public static void main(String[] args) {  
        MusicTest1 mt = new MusicTest1();  
        mt.play();  
    }  
}
```

We need a Sequencer object. It's the main part of the MIDI device/instrument we're using. It's the thing that, well, sequences all the MIDI information into a "song." But we don't make a brand new one ourselves—we have to ask the MidiSystem to give us one.

### Something's wrong!

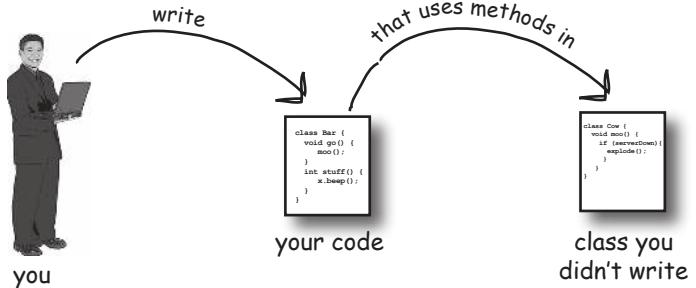
This code won't compile! The compiler says there's an "unreported exception" that must be caught or declared.



```
File Edit Window Help SayWhat?  
  
% javac MusicTest1.java  
MusicTest1.java:13: unreported exception javax.sound.midi.  
MidiUnavailableException; must be caught or declared to be  
thrown  
  
    Sequencer sequencer = MidiSystem.getSequencer();  
  
^  
  
1 errors
```

# What happens when a method you want to call (probably in a class you didn't write) is risky?

- ① Let's say you want to call a method in a class that you didn't write.**



- ② That method does something risky, something that might not work at runtime.**

A code editor window labeled "class you didn't write" contains the following Java code:

```
class Cow {
    void moo() {
        if (serverDown) {
            explode();
        }
    }
}
```

An arrow points from this code to a larger version of the same code in a separate window, labeled "void moo() { if (serverDown) { explode(); } }".

- ③ You need to know that the method you're calling is risky.**

A person labeled "you" is shown thinking, with a thought bubble containing the text: "I wonder if that method could blow up..."

Another thought bubble contains the text: "My moo() method will explode if the server is down."

A code editor window labeled "class you didn't write" shows the same Java code as before: `class Cow { void moo() { if (serverDown) { explode(); } } }`.

- ④ You then write code that can handle the failure if it does happen. You need to be prepared, just in case.**

A person labeled "you" is shown thinking, with a thought bubble containing the text: "Now that I know, I can take precautions."

From the person, an arrow labeled "write safely" points down to a code editor window labeled "your code". Inside is the following Java code:

```
class Bar {
    void goo() {
        try {
            moo();
        } catch (Exception e) {
            cry();
        }
    }
}
```

when things might go wrong

## Methods in Java use exceptions to tell the calling code, “Something Bad Happened. I failed.”

Java’s exception-handling mechanism is a clean, well-lighted way to handle “exceptional situations” that pop up at runtime; it lets you put all your error-handling code in one easy-to-read place. It’s based on the method you’re calling *telling you* it’s risky (i.e., that the method *might* generate an exception), so that you can write code to deal with that possibility. If you *know* you might get an exception when you call a particular method, you can be *prepared* for—possibly even *recover* from—the problem that caused the exception.

So, how does a method tell you it might throw an exception? You find a **throws** clause in the risky method’s declaration.

**The `getSequencer()` method takes a risk. It can fail at runtime. So it must “declare” the risk you take when you call it.**

### getSequencer

```
public static Sequencer getSequencer()  
    throws MidiUnavailableException
```

Obtains the default Sequencer, connected to a default device. The returned Sequencer instance is connected to the default Synthesizer, as returned by `getSynthesizer()`. If there is no Synthesizer available, or the default Synthesizer cannot be opened, the sequencer is connected to the default Receiver, as returned by `getReceiver()`. The connection is made by retrieving a Transmitter instance from the Sequencer and setting its Receiver. Closing and re-opening the sequencer will restore the connection to the default device.

This method is equivalent to calling `getSequencer(true)`.

If the system property `javax.sound.midi.Sequencer` is defined or it is defined in the file "sound.properties", it is used to identify the default sequencer. For details, refer to the class description.

**Returns:**

the default sequencer, connected to a default Receiver

**Throws:**

`MidiUnavailableException` - if the sequencer is not available due to resource restrictions, or there is no Receiver available by any installed `MidiDevice`, or no sequencer is installed in the system

**See Also:**

`getSequencer(boolean)`, `getSynthesizer()`, `getReceiver()`

This part tells you WHEN you might get that exception—in this case, because of resource restrictions (which could mean the sequencer is already being used).

The API does tell you that `getSequencer()` can throw an exception: `MidiUnavailableException`. A method has to declare the exceptions it might throw.

Risky methods that could fail at runtime declare the exceptions that might happen using “throws `SomeKindOfException`” on their method declaration.

# The compiler needs to know that YOU know you're calling a risky method

If you wrap the risky code in something called a **try/catch**, the compiler will relax.

A try/catch block tells the compiler that you *know* an exceptional thing could happen in the method you're calling, and that you're prepared to handle it. That compiler doesn't care *how* you handle it; it cares only that you say you're taking care of it.

```
import javax.sound.midi.*;

public class MusicTest1 {

    public void play() {
        try {
            Sequencer sequencer = MidiSystem.getSequencer();
            System.out.println("Successfully got a sequencer");
        } catch(MidiUnavailableException e) {
            System.out.println("Bummer");
        }
    }

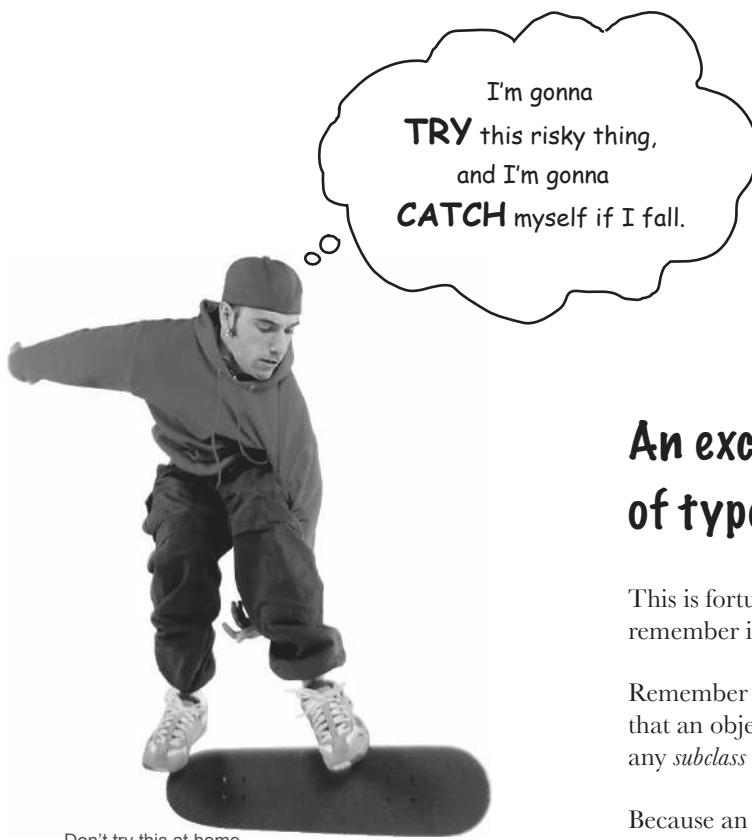
    public static void main(String[] args) {
        MusicTest1 mt = new MusicTest1();
        mt.play();
    }
}
```

Dear Compiler,  
*I know I'm taking a risk here, but  
 don't you think it's worth it? What  
 should I do?*  
 Signed, Geeky in Waikiki

Dear Geeky,  
*Life is short (especially on the  
 heap). Take the risk. Try it. But  
 just in case things don't work out, be  
 sure to catch any problems before all  
 hell breaks loose.*

} Put the risky thing in  
 a "try" block. It's the  
 "risky" getSequencer  
 method that might  
 throw an exception.

Make a "catch" block for what  
 to do if the exceptional situation  
 happens—in other words, a  
 MidiUnavailableException is thrown  
 by the call to getSequencer().



## An exception is an object... of type Exception

This is fortunate, because it would be much harder to remember if exceptions were of type Broccoli.

Remember from the polymorphism chapters (7 and 8) that an object of type Exception *can* be an instance of any *subclass* of Exception.

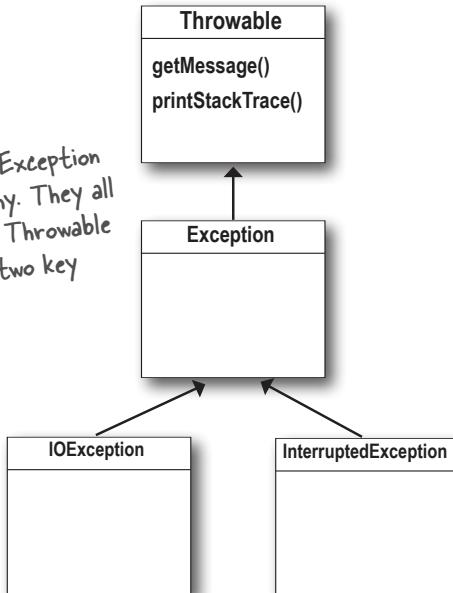
Because an *Exception* is an object, what you *catch* is an object. In the following code, the **catch** argument is declared as type Exception, and the parameter reference variable is *ex*.

```
try {  
    // do risky thing  
}  
} catch (Exception e) {  
    // try to recover  
}
```

It's just like declaring a method argument.

This code runs only if an Exception is thrown.

Part of the Exception class hierarchy. They all extend class Throwable and inherit two key methods.



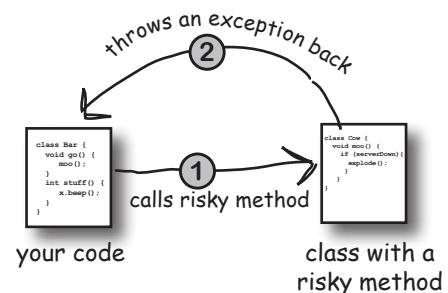
What you write in a catch block depends on the exception that was thrown. For example, if a server is down, you might use the catch block to try another server. If the file isn't there, you might ask the user for help finding it.

# If it's your code that catches the exception, then whose code throws it?

You'll spend much more of your Java coding time *handling* exceptions than you'll spend *creating* and *throwing* them yourself. For now, just know that when your code *calls* a risky method—a method that declares an exception—it's the risky method that *throws* the exception back to *you*, the caller.

In reality, it might be you who wrote both classes. It really doesn't matter who writes the code...what matters is knowing which method *throws* the exception and which method *catches* it.

When somebody writes code that could throw an exception, they must *declare* the exception.



## ① Risky, exception-throwing code:

```
public void takeRisk() throws BadException {
    if (abandonAllHope) {
        throw new BadException();
    }
}
```

*Create a new Exception object and throw it.*

This method MUST tell the world (by declaring) that it throws a BadException.

One method will catch what another method throws. An exception is always thrown back to the caller.

The method that throws has to declare that it might throw the exception.

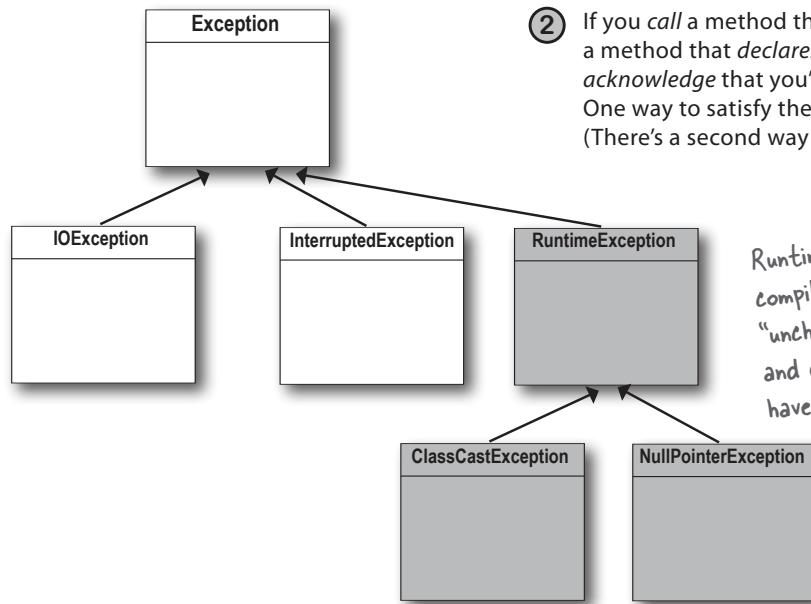
## ② Your code that *calls* the risky method:

```
public void crossFingers() {
    try {
        anObject.takeRisk();
    } catch (BadException e) {
        System.out.println("Aaargh!");
        e.printStackTrace();
    }
}
```

If you can't recover from the exception, at LEAST get a stack trace using the `printStackTrace()` method that all exceptions inherit.

## The compiler checks for everything except **RuntimeExceptions**.

Exceptions that are NOT subclasses of `RuntimeException` are checked for by the compiler. They're called "checked exceptions."



The compiler guarantees:

- ① If you *throw* an exception in your code, you *must* declare it using the `throws` keyword in your method declaration.
- ② If you *call* a method that throws an exception (in other words, a method that *declares* it throws an exception), you must *acknowledge* that you're aware of the exception possibility. One way to satisfy the compiler is to wrap the call in a try/catch. (There's a second way we'll look at a little later in this chapter.)

`RuntimeExceptions` are NOT checked by the compiler. They're known as (big surprise here) "unchecked exceptions." You can throw, catch, and declare `RuntimeExceptions`, but you don't have to, and the compiler won't check.

### there are no Dumb Questions

**Q:** Wait just a minute! How come this is the FIRST time we've had to try/catch an Exception? What about the exceptions I've already gotten like `NullPointerException` and the exception for `DivideByZero`? I even got a `NumberFormatException` from the `Integer.parseInt()` method. How come we didn't have to catch those?

**A:** The compiler cares about all subclasses of `Exception`, *unless* they are a special type, `RuntimeException`. Any exception class that extends `RuntimeException` gets a free pass. `RuntimeExceptions` can be thrown anywhere, with or without `throws` declarations or try/catch blocks. The compiler doesn't bother checking whether a method declares that it throws a `RuntimeException`, or whether the caller acknowledges that they might get that exception at runtime.

**Q:** I'll bite. WHY doesn't the compiler care about those runtime exceptions? Aren't they just as likely to bring the whole show to a stop?

**A:** Most `RuntimeExceptions` come from a problem in your code logic, rather than a condition that fails at runtime in ways that you cannot predict or prevent. You *cannot* guarantee the file is there. You *cannot* guarantee the server is up. But you *can* make sure your code doesn't index off the end of an array (that's what the `.length` attribute is for).

You WANT `RuntimeExceptions` to happen at development and testing time. You don't want to code in a try/catch, for example, and have the overhead that goes with it, to catch something that shouldn't happen in the first place.

A try/catch is for handling exceptional situations, not flaws in your code. Use your catch blocks to try to recover from situations you can't guarantee will succeed. Or at the very least, print out a message to the user and a stack trace so somebody can figure out what happened.

**BULLET POINTS**

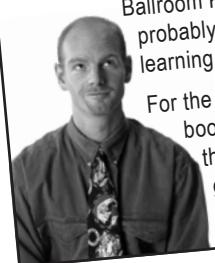
- A method can throw an exception when something fails at runtime.
- An exception is always an object of type `Exception`. (This, as you remember from the polymorphism chapters (7 and 8), means the object is from a class that has `Exception` somewhere up its inheritance tree.)
- The compiler does NOT pay attention to exceptions that are of type `RuntimeException`. A `RuntimeException` does not have to be declared or wrapped in a try/catch (although you're free to do either or both of those things).
- All Exceptions the compiler cares about are called “checked exceptions,” which really means *compiler*-checked exceptions. Only `RuntimeExceptions` are excluded from compiler checking. All other exceptions must be acknowledged in your code.
- A method throws an exception with the keyword `throw`, followed by a new exception object:
 

```
throw new NoCaffeineException();
```
- Methods that *might* throw a checked exception **must** announce it with a `throws SomeException` declaration.
- If your code calls a checked-exception-throwing method, it must reassure the compiler that precautions have been taken.
- If you’re prepared to handle the exception, wrap the call in a try/catch, and put your exception handling/recovery code in the catch block.
- If you’re not prepared to handle the exception, you can still make the compiler happy by officially “ducking” the exception. We’ll talk about ducking a little later in this chapter.

**metacognitive tip**

If you’re trying to learn something new, make that the *last* thing you try to learn before going to sleep. So, once you put this book down (assuming you can tear yourself away from it), don’t read anything else more challenging than the back of a Cheerios™ box. Your brain needs time to process what you’ve read and learned. That could take a few hours. If you try to shove something new in right on top of your Java, some of the Java might not “stick.”

Of course, this doesn’t rule out learning a physical skill. Working on your latest Ballroom KickBoxing routine probably won’t affect your Java learning.



For the best results, read this book (or at least look at the pictures) right before going to sleep.

**Sharpen your pencil**

**Which of these do you think might throw an exception that the compiler should care about? These are things that you CAN’T control in your code. We did the first one.**

(Because it was the easiest.)

→ **Yours to solve.**

**Things you want to do**

- ✓ Connect to a remote server
- Access an array beyond its length
- Display a window on the screen
- Retrieve data from a database
- See if a text file is where you *think* it is
- Create a new file
- Read a character from the command line

**What might go wrong**

**The server is down**

---



---



---



---



---



---



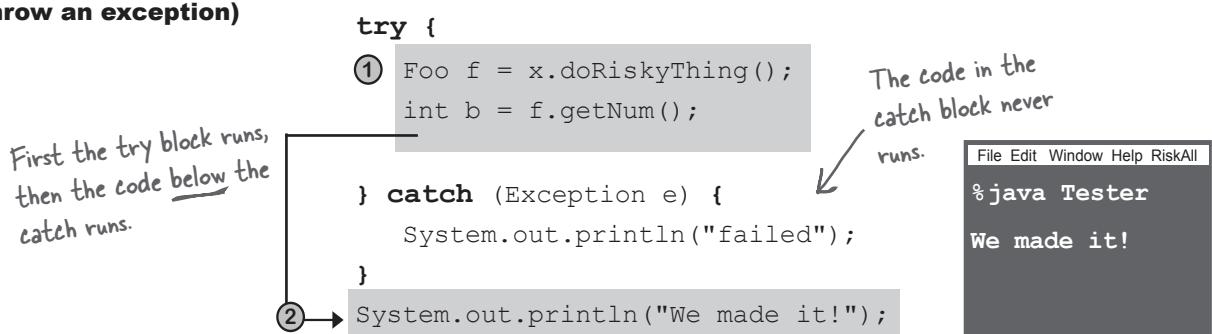
---

# Flow control in try/catch blocks

When you call a risky method, one of two things can happen. The risky method either succeeds, and the try block completes, or the risky method throws an exception back to your calling method.

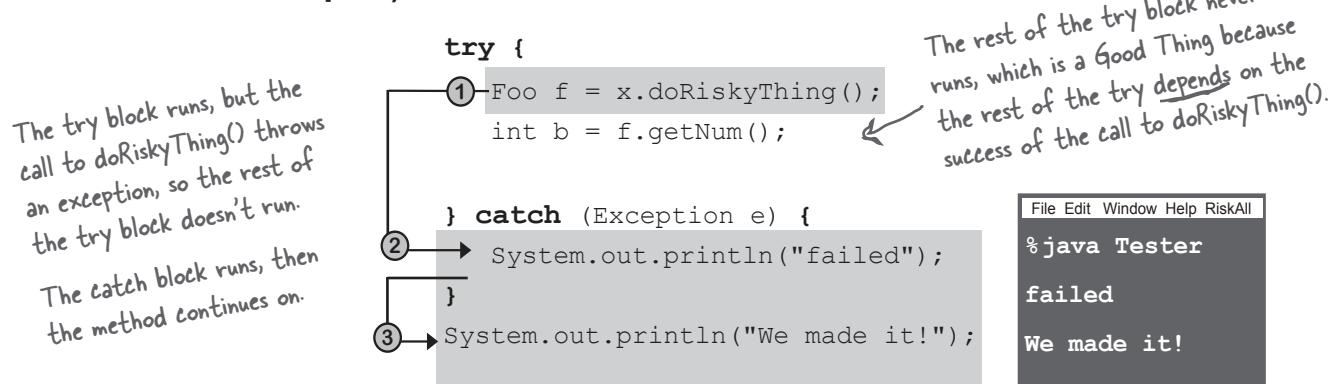
## If the try succeeds

(`doRiskyThing()` does *not* throw an exception)



## If the try fails

(because `doRiskyThing()` does throw an exception)



# Finally: for the things you want to do no matter what

If you try to cook something, you start by turning on the oven.

If the thing you try is a complete **failure**, ***you have to turn off the oven.***

If the thing you try **succeeds**, ***you have to turn off the oven.***

***You have to turn off the oven no matter what!***

**A finally block is where you put code that must run regardless of an exception.**

```
try {
    turnOvenOn();
    x.bake();
} catch (BakingException e) {
    e.printStackTrace();
} finally {
    turnOvenOff();
}
```

Without finally, you have to put the turnOvenOff() in *both* the try and the catch because ***you have to turn off the oven no matter what.*** A finally block lets you put all your important cleanup code in *one* place instead of duplicating it like this:

```
try {
    turnOvenOn();
    x.bake();
    turnOvenOff();
} catch (BakingException e) {
    e.printStackTrace();
    turnOvenOff();
}
```



**If the try block fails (an exception),** flow control immediately moves to the catch block. When the catch block completes, the finally block runs. When the finally block completes, the rest of the method continues on.

**If the try block succeeds (no exception),** flow control skips over the catch block and moves to the finally block. When the finally block completes, the rest of the method continues on.

**If the try or catch block has a return statement, finally will still run!** Flow jumps to the finally, then back to the return.



Sharpen your pencil

→ Yours to solve.

# Flow Control

Look at the code to the left. What do you think the output of this program would be? What do you think it would be if the third line of the program were changed to `String test = "yes";?`  
Assume `ScaryException` extends `Exception`.

```
public class TestExceptions {

    public static void main(String[] args) {
        String test = "no";
        try {
            System.out.println("start try");
            doRisky(test);
            System.out.println("end try");
        } catch (ScaryException se) {
            System.out.println("scary exception");
        } finally {
            System.out.println("finally");
        }
        System.out.println("end of main");
    }

    static void doRisky(String test) throws ScaryException {
        System.out.println("start risky");
        if ("yes".equals(test)) {
            throw new ScaryException();
        }
        System.out.println("end risky");
    }

    class ScaryException extends Exception {
    }
}
```

Output when `test = "no"`
Output when `test = "yes"`

When `test = "yes"`: start try - start risky - scary exception - finally - end of main  
 When `test = "no"`: start try - start risky - end risky - finally - end of main

# Did we mention that a method can throw more than one exception?

A method can throw multiple exceptions if it darn well needs to. But a method's declaration must declare *all* the checked exceptions it can throw (although if two or more exceptions have a common superclass, the method can declare just the superclass).

## Catching multiple exceptions

The compiler will make sure that you've handled *all* the checked exceptions thrown by the method you're calling. Stack the *catch* blocks under the *try*, one after the other. Sometimes the order in which you stack the catch blocks matters, but we'll get to that a little later.

```
public class Laundry {
    public void doLaundry() throws PantsException, LingerieException {
        // code that could throw either exception
    }
}
```



This method declares two, count 'em, TWO exceptions.

```
public class WashingMachine {
    public void go() {
        Laundry laundry = new Laundry();
        try {
            laundry.doLaundry();
        } catch (PantsException pex) {
            // recovery code
        } catch (LingerieException lex) {
            // recovery code
        }
    }
}
```



If doLaundry() throws a PantsException, it lands in the PantsException catch block.

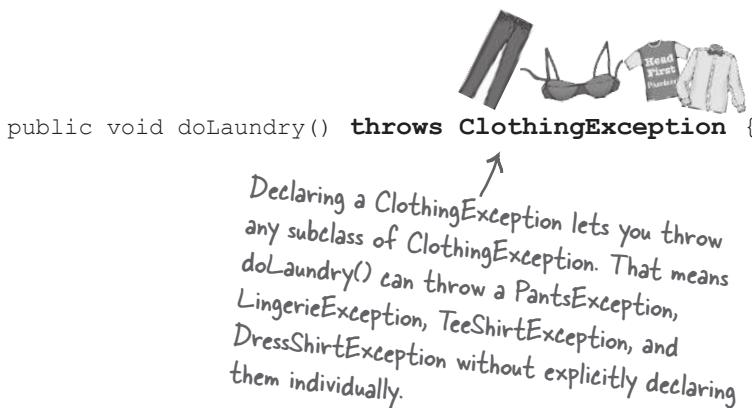


If doLaundry() throws a LingerieException, it lands in the LingerieException catch block.

# Exceptions are polymorphic

Exceptions are objects, remember. There's nothing all that special about one, except that it is *a thing that can be thrown*. So like all good objects, Exceptions can be referred to polymorphically. A LingerieException *object*, for example, could be assigned to a ClothingException *reference*. A PantsException could be assigned to an Exception reference. You get the idea. The benefit for exceptions is that a method doesn't have to explicitly declare every possible exception it might throw; it can declare a superclass of the exceptions. Same thing with catch blocks—you don't have to write a catch for each possible exception as long as the catch (or catches) you have can handle any exception thrown.

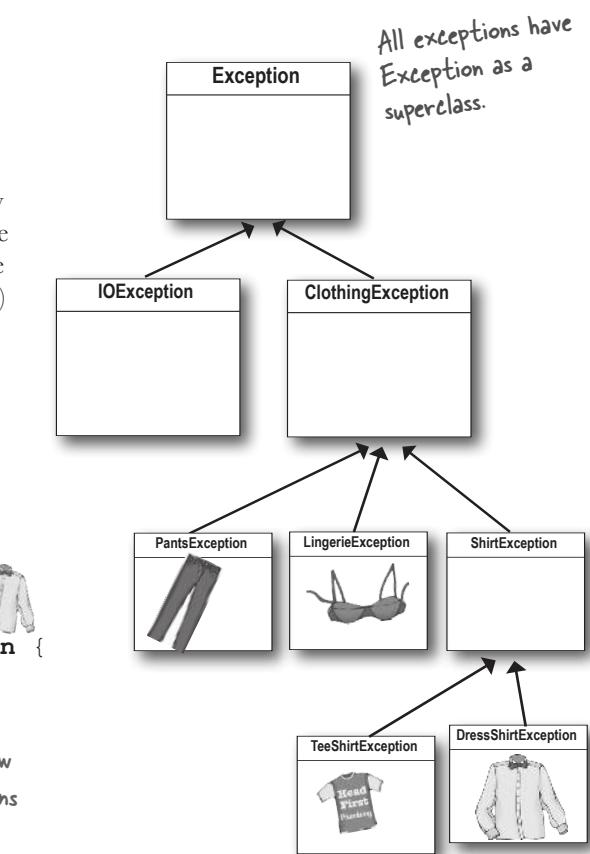
## ① You can DECLARE exceptions using a superclass of the exceptions you throw.



## ② You can CATCH exceptions using a superclass of the exception thrown.

```
try {
    laundry.doLaundry();
    // code
} catch(ClothingException cex) {
    // recovery code
}
```

Can catch any ClothingException subclass



```
try {
    laundry.doLaundry();
    // code
} catch(ClothingException cex) {
    // recovery code
} catch(ShirtException shex) {
    // recovery code
}
```

Can catch only TeeShirtException and DressShirtException

**Just because you CAN catch everything  
with one big super polymorphic catch,  
doesn't always mean you SHOULD.**

You *could* write your exception-handling code so that you specify only *one* catch block, using the superclass Exception in the catch clause, so that you'll be able to catch *any* exception that might be thrown.

```
try {
    laundry.doLaundry();
} catch(Exception ex) {
    // recovery code...
}
```

Recovery from WHAT? This catch block will catch ANY and all exceptions, so you won't automatically know what went wrong.

**Write a different catch block for each exception that you need to handle uniquely.**

For example, if your code deals with (or recovers from) a TeeShirtException differently than it handles a LingerieException, write a catch block for each. But if you treat all other types of ClothingException in the same way, then add a ClothingException catch to handle the rest.

```
try {
    laundry.doLaundry();

} catch (TeeShirtException tex) {
    // recovery from TeeShirtException
    
}

} catch (LingerieException lex) {
    // recovery from LingerieException
    
}

} catch (ClothingException cex) {
    // recovery from all others
    
}
```

TeeShirtExceptions and LingerieExceptions need different recovery code, so you should use different catch blocks.

All other ClothingExceptions are caught here.

## Multiple catch blocks must be ordered from smallest to biggest



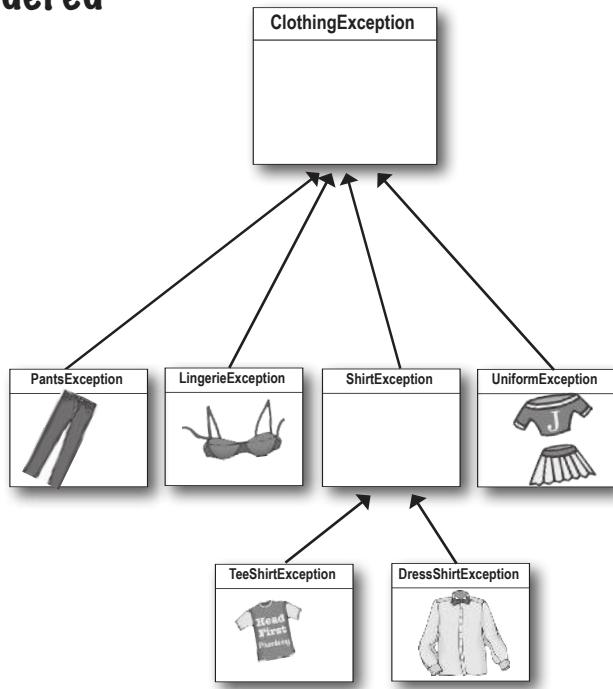
```
catch(TeeShirtException tex)
```



```
catch(ShirtException sex)
```



```
catch(ClothingException cex)
```



The higher up the inheritance tree, the bigger the catch “basket.” As you move down the inheritance tree, toward more and more specialized Exception classes, the catch “basket” is smaller. It’s just plain old polymorphism.

A ShirtException catch is big enough to take a TeeShirtException or a DressShirtException (and any future subclass of anything that extends ShirtException). A ClothingException is even bigger (i.e., there are more things that can be referenced using a ClothingException type). It can take an exception of type ClothingException (duh) and any ClothingException subclasses: PantsException, UniformException, LingerieException, and ShirtException. The mother of all catch arguments is type **Exception**; it will catch *any* exception, including runtime (unchecked) exceptions, so you probably won’t use it outside of testing.

# You can't put bigger baskets above smaller baskets

Well, you *can*, but it won't compile. Catch blocks are not like overloaded methods where the best match is picked. With catch blocks, the JVM simply starts at the first one and works its way down until it finds a catch that's broad enough (in other words, high enough on the inheritance tree) to handle the exception. If your first catch block is `catch (Exception ex)`, the compiler knows there's no point in adding any others—they'll never be reached.

*Don't do this!*

```
try {
    laundry.doLaundry();

} catch(ClothingException cex) {
    // recovery from ClothingException

} catch(LingerieException lex) {
    // recovery from LingerieException

} catch(ShirtException shex) {
    // recovery from ShirtException
}
```



Size matters when you have multiple catch blocks. The one with the biggest basket has to be on the bottom. Otherwise, the ones with smaller baskets are useless.



Siblings (exceptions at the same level of the hierarchy tree, like `PantsException` and `LingerieException`) can be in any order, because they can't catch one another's exceptions.

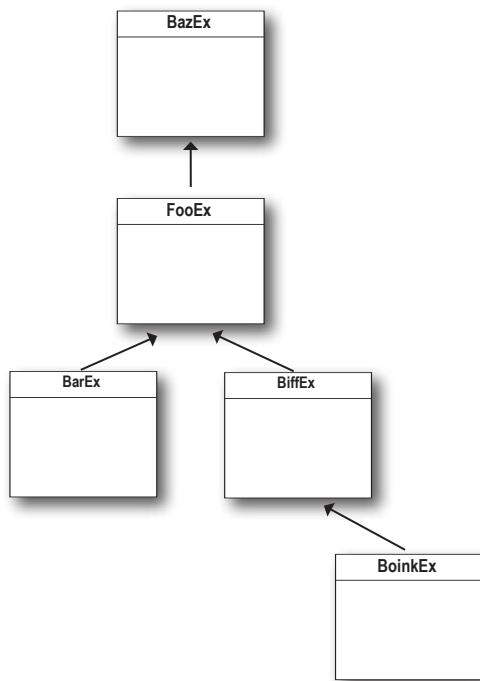
You could put `ShirtException` above `LingerieException`, and nobody would mind. Because even though `ShirtException` is a bigger (broader) type because it can catch other classes (its own subclasses), `ShirtException` can't catch a `LingerieException`, so there's no problem.

## polymorphic puzzle



Assume the try/catch block here is legally coded. Your task is to draw two different class diagrams that can accurately reflect the Exception classes. In other words, what class inheritance structures would make the try/catch blocks in the sample code legal?

```
try {
    x.doRisky();
} catch(AlphaEx a) {
    // recovery from AlphaEx
} catch(BetaEx b) {
    // recovery from BetaEx
} catch(GammaEx c) {
    // recovery from GammaEx
} catch(DeltaEx d) {
    // recovery from DeltaEx
}
```



Your task is to create two different *legal* try/catch structures (similar to the one above left) to accurately represent the class diagram shown on the left. Assume ALL of these exceptions might be thrown by the method with the try block.

# When you don't want to handle an exception...

**just duck it**

**If you don't want to handle an exception, you can duck it by declaring it.**

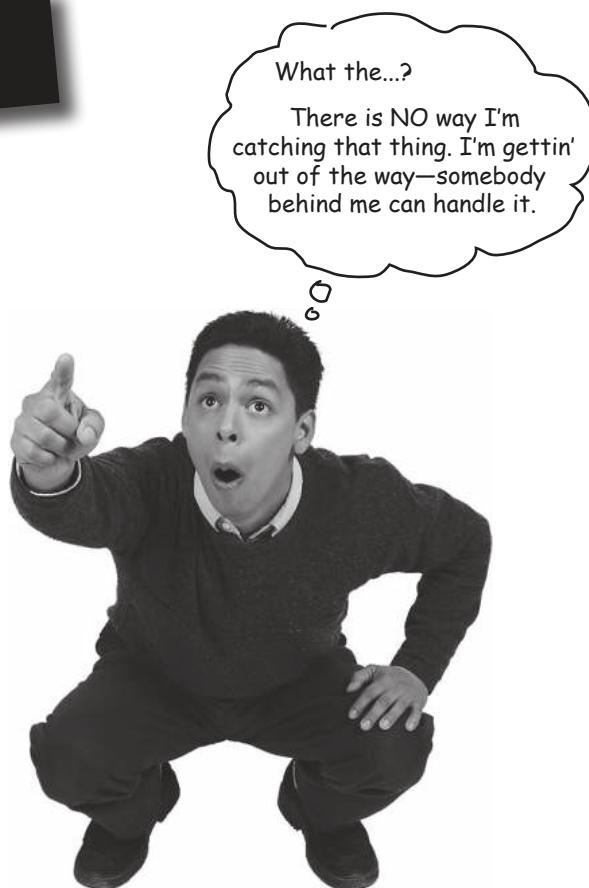
When you call a risky method, the compiler needs you to acknowledge it. Most of the time, that means wrapping the risky call in a try/catch. But you have another alternative: simply duck it and let the method that called you catch the exception.

It's easy—all you have to do is *declare* that *you* throw the exceptions. Even though, technically, *you* aren't the one doing the throwing, it doesn't matter. You're still the one letting the exception whiz right on by.

But if you duck an exception, then you don't have a try/catch, so what happens when the risky method (`doLaundry()`) *does* throw the exception?

When a method throws an exception, that method is popped off the stack immediately, and the exception is thrown to the next method down the stack—the *caller*. But if the *caller* is a *ducker*, then there's no catch for it, so the *caller* pops off the stack immediately, and the exception is thrown to the next method and so on...where does it end? You'll see a little later.

```
public void foo() throws ReallyBadException {
    // call risky method without a try/catch
    laundry.doLaundry();
}
```



You don't REALLY throw it, but since you don't have a try/catch for the risky method you call, YOU are now the "risky method." Because now, whoever calls YOU has to deal with the exception.

# Ducking (by declaring) only delays the inevitable

**Sooner or later, somebody has to deal with it. But what if main() ducks the exception?**

```
public class Washer {  
    Laundry laundry = new Laundry();  
  
    public void foo() throws ClothingException {  
        laundry.doLaundry();  
    }  
  
    public static void main (String[] args) throws ClothingException {  
        Washer a = new Washer();  
        a.foo();  
    }  
}
```

Both methods duck the exception (by declaring it), so there's nobody to handle it! This compiles just fine.

1 doLaundry() throws a ClothingException



main() calls foo()  
foo() calls doLaundry()  
doLaundry() is running and throws a ClothingException

2 foo() ducks the exception



doLaundry() pops off the stack immediately, and the exception is thrown back to foo().  
But foo() doesn't have a try/catch, so...

3 main() ducks the exception



foo() pops off the stack, and the exception is thrown back to main(). But main() doesn't have a try/catch, so the exception is thrown back to... who? What? There's nobody left but the JVM, and it's thinking, "Don't expect ME to get you out of this."

4 The JVM shuts down



We're using the T-shirt to represent a Clothing Exception. We know, we know...you would have preferred the blue jeans.

## Handle or Declare. It's the law.

So now we've seen both ways to satisfy the compiler when you call a risky (exception-throwing) method.

### ① HANDLE

Wrap the risky call in a try/catch

```
try {
    laundry.doLaundry();
} catch(ClothingException cex) {
    // recovery code
}
```

This had better be a big enough catch to handle all exceptions that doLaundry() might throw. Or else the compiler will still complain that you're not catching all of the exceptions.

### ② DECLARE (duck it)

Declare that YOUR method throws the same exceptions as the risky method you're calling.

```
void foo() throws ClothingException {
    laundry.doLaundry();
}
```

The doLaundry() method throws a ClothingException, but by declaring the exception, the foo() method gets to duck the exception. No try/catch.

But now this means that whoever calls the foo() method has to follow the Handle or Declare law. If foo() ducks the exception (by declaring it) and main() calls foo(), then main() has to deal with the exception.

```
public class Washer {
    Laundry laundry = new Laundry();

    public void foo() throws ClothingException {
        laundry.doLaundry();
    }

    public static void main (String[] args) {
        Washer a = new Washer();
        a.foo();
    }
}
```

Because the foo() method ducks the ClothingException thrown by doLaundry(), main() has to wrap a.foo() in a try/catch, or main() has to declare that it, too, throws ClothingException!

**TROUBLE!!**  
Now main() won't compile, and we get an "unreported exception" error. As far as the compiler's concerned, the foo() method throws an exception.

## Getting back to our music code...

Now that you've completely forgotten, we started this chapter with a first look at some JavaSound code. We created a Sequencer object, but it wouldn't compile because the method Midi.getSequencer() declares a checked exception (MidiUnavailableException). But we can fix that now by wrapping the call in a try/catch.

```
public void play() {
    try {
        Sequencer sequencer = MidiSystem.getSequencer();
        System.out.println("Successfully got a sequencer");

    } catch(MidiUnavailableException e) {
        System.out.println("Bummer");
    }
}
```

No problem calling getSequencer(), now that we've wrapped it in a try/catch block.

The catch parameter has to be the "right" exception. If we said catch(FileNotFoundException fnf), the code would not compile, because polymorphically a MidiUnavailableException won't fit into a FileNotFoundException.

Remember, it's not enough to have a catch block...you have to catch the thing being thrown!

## Exception Rules

---

- ① You cannot have a catch or finally without a try.**

```
void go() {
    Foo f = new Foo();
    f.foof();
    catch(FooException ex) { }
}
```

NOT LEGAL!  
Where's the try?

- ② You cannot put code between the try and the catch.**

```
try {
    x.doStuff();
}
int y = 43;
} catch(Exception ex) { }
```

NOT LEGAL! You can't put code between the try and the catch.

- ③ A try MUST be followed by either a catch or a finally.**

```
try {
    x.doStuff();
} finally {
    // cleanup
}
```

LEGAL because you have a finally, even though there's no catch. But you cannot have a try by itself.

- ④ A try with only a finally (no catch) must still declare the exception.**

```
void go() throws FooException {
    try {
        x.doStuff();
    } finally { }
}
```

A try without a catch doesn't satisfy the handle or declare law.

# Code Kitchen



You don't have to do it yourself, but it's a lot more fun if you do.

The rest of this chapter is optional; you can use Ready-Bake Code for all the music apps.

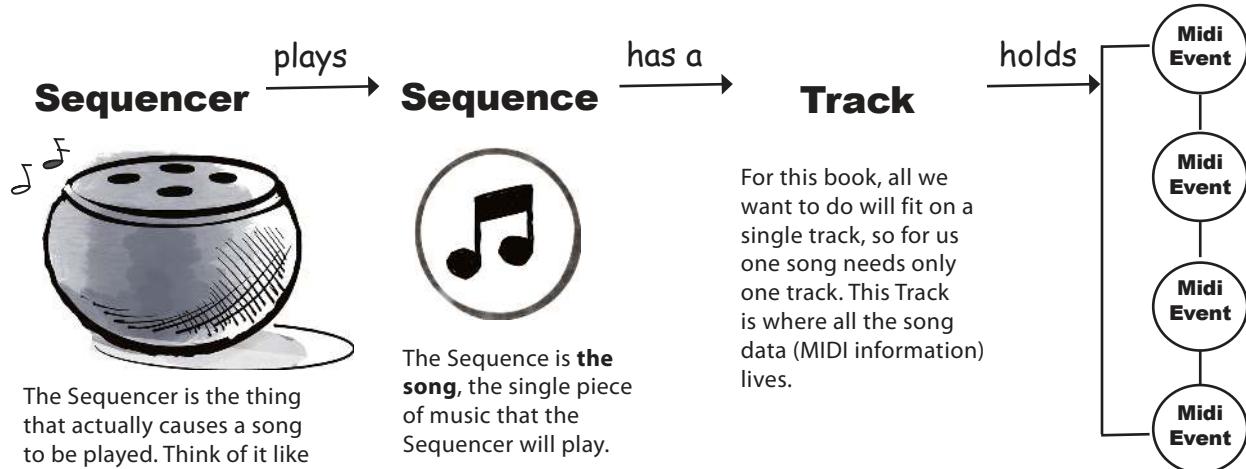
But if you want to learn more about JavaSound, turn the page.

# Making actual sound

Remember near the beginning of the chapter, we looked at how MIDI data holds the instructions for *what* should be played (and *how* it should be played) and we also said that MIDI data doesn't actually *create any sound that you hear*. For sound to come out of the speakers, the MIDI data has to be sent through some kind of MIDI device that takes the MIDI instructions and renders them in sound, by triggering either a hardware instrument or a "virtual" instrument (software synthesizer). In this book, we're using only software devices, so here's how it works in JavaSound:

## You need FOUR things:

- ① The thing that plays the music
- ② The music to be played...a song.
- ③ The part of the Sequence that holds the actual information
- ④ The actual music information: notes to play, how long, etc.



For this book, think of the Sequence as a single-song CD (has only one Track). The information about how to play the song lives on the Track, and the Track is part of the Sequence.

## And you need **FIVE** steps:

- ① Get a **Sequencer** and open it

```
Sequencer player = MidiSystem.getSequencer();  
player.open();
```

- ② Make a new **Sequence**

```
Sequence seq = new Sequence(timing, 4);
```

- ③ Get a new **Track** from the Sequence

```
Track t = seq.createTrack();
```

- ④ Fill the Track with **MidiEvents** and give the Sequence to the Sequencer

```
t.add(myMidiEvent1);  
player.setSequence(seq);
```



## Version 1: Your very first sound player app

Type it in and run it. You'll hear the sound of someone playing a single note on a piano! (OK, maybe not *someone*, but *something*.)

```

import javax.sound.midi.*;           ← Don't forget to import the midi package.
import static javax.sound.midi.ShortMessage.*;
public class MiniMiniMusicApp {
    public static void main(String[] args) {
        MiniMiniMusicApp mini = new MiniMiniMusicApp();
        mini.play();
    }

    public void play() {
        try {
            ① Sequencer player = MidiSystem.getSequencer();
            player.open();
            ② Sequence seq = new Sequence(Sequence.PPQ, 4);
            ③ Track track = seq.createTrack();
            ④ ShortMessage msg1 = new ShortMessage();
            msg1.setMessage(NOTE_ON, 1, 44, 100);
            MidiEvent noteOn = new MidiEvent(msg1, 1);
            track.add(noteOn);

            ShortMessage msg2 = new ShortMessage();
            msg2.setMessage(NOTE_OFF, 1, 44, 100);
            MidiEvent noteOff = new MidiEvent(msg2, 16);
            track.add(noteOff);

            player.setSequence(seq);   ← Give the Sequence to the Sequencer (like
                                      selecting the song to play).
            player.start();          ← start() the Sequencer (play the song).
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

Don't forget to import the midi package.

We're using a static import here so we can use the constants in the ShortMessage class.

Get a Sequencer and open it (so we can use it...a Sequencer doesn't come already open).

Don't worry about the arguments to the Sequence constructor. Just copy these (think of 'em as Ready-bake arguments).

Ask the Sequence for a Track. Remember, the Track lives in the Sequence, and the MIDI data lives in the Track.

Put some MidiEvents into the Track. This part is mostly Ready-Bake Code. The only thing you'll have to care about are the arguments to the setMessage() method, and the arguments to the MidiEvent constructor. We'll look at those arguments on the next page.



# Making a MidiEvent (song data)

A MidiEvent is an instruction for part of a song. A series of MidiEvents is kind of like sheet music, or a player piano roll. Most of the MidiEvents we care about describe **a thing to do** and the **moment in time to do it**. The moment in time part matters, since timing is everything in music. This note follows this note and so on. And because MidiEvents are so detailed, you have to say at what moment to *start* playing the note (a NOTE ON event) and at what moment to *stop* playing the notes (NOTE OFF event). So you can imagine that firing the “stop playing note G” (NOTE OFF message) *before* the “start playing Note G” (NOTE ON) message wouldn’t work.

The MIDI instruction actually goes into a Message object; the MidiEvent is a combination of the Message plus the moment in time when that message should “fire.” In other words, the Message might say, “Start playing Middle C,” while the MidiEvent would say, “Trigger this message at beat 4.”

So we always need a Message and a MidiEvent.

The Message says *what* to do, and the MidiEvent says *when* to do it.

**A MidiEvent says  
what to do and when  
to do it.**

**Every instruction  
must include the  
timing for that  
instruction.**

**In other words,  
at which beat that  
thing should happen.**

## ① Make a Message

```
ShortMessage msg = new ShortMessage();
```

## ② Put the Instruction in the Message

```
msg.setMessage(144, 1, 44, 100);
```

This message says, “start playing note 44”  
(we’ll look at the other numbers on the  
next page).

## ③ Make a new MidiEvent using the Message

```
MidiEvent noteOn = new MidiEvent(a, 1);
```

The instructions are in the message, but the  
MidiEvent adds the moment in time when the  
instruction should be triggered. This MidiEvent says  
to trigger message ‘a’ at the first beat (beat 1).

## ④ Add the MidiEvent to the Track

```
track.add(noteOn);
```

A Track holds all the MidiEvent objects. The Sequence organizes them according to when each event is supposed to happen, and then the Sequencer plays them back in that order. You can have lots of events happening at the exact same moment in time. For example, you might want two notes played simultaneously, or even different instruments playing different sounds at the same time.

# MIDI message: the heart of a MidiEvent

A MIDI message holds the part of the event that says *what* to do. It's the actual instruction you want the sequencer to execute. The first argument of an instruction is always the type of the message. The values you pass to the other three arguments depend on the type of message. For example, a message of type 144 means "NOTE ON." But in order to carry out a NOTE ON, the sequencer needs to know a few things. Imagine the sequencer saying, "OK, I'll play a note, but *which channel?* In other words, do you want me to play a Drum note or a Piano note? And *which note?* Middle-C? D Sharp? And while we're at it, at *which velocity* should I play the note?"

To make a MIDI message, make a ShortMessage instance and invoke setMessage(), passing in the four arguments for the message. But remember, the message says only *what* to do, so you still need to stuff the message into an event that adds *when* that message should "fire."

## Anatomy of a message

The *first* argument to setMessage() always represents the message "type," while the *other* three arguments represent different things depending on the message type.

```
msg.setMessage(144, 1, 44, 100);
    message type      channel      note to play      velocity
    The last 3 args vary depending on the
    message type. This is a NOTE ON message, so
    the other args are for things the Sequencer
    needs to know in order to play a note.
```

### ① Message type

144 means  
NOTE ON



128 means  
NOTE OFF



You can use the constant  
values in ShortMessage  
instead of having to  
remember the numbers, e.g.,  
ShortMessage.NOTE\_ON.

**The Message says what to do; the  
MidiEvent says when to do it.**

### ② Channel

Think of a channel like a musician in a band. Channel 1 is musician 1 (the keyboard player), channel 9 is the drummer, etc.

### ③ Note to play

A number from 0 to 127, going from low to high notes.



### ④ Velocity

How fast and hard did you press the key? 0 is so soft you probably won't hear anything, but 100 is a good default.

# Change a message

Now that you know what's in a MIDI message, you can start experimenting. You can change the note that's played, how long the note is held, add more notes, and even change the instrument.

## ① Change the note

Try a number between 0 and 127 in the note on and note off messages.

```
msg.setMessage(144, 1, 20, 100);
```



## ② Change the duration of the note

Change the note off event (not the message) so that it happens at an earlier or later beat.

```
msg.setMessage(128, 1, 44, 100);
MidiEvent noteOff = new MidiEvent(b, 3);
```



## ③ Change the instrument

Add a new message, BEFORE the note-playing message, that sets the instrument in channel 1 to something other than the default piano. The change-instrument message is "192," and the third argument represents the actual instrument (try a number between 0 and 127).

```
first.setMessage(192, 1, 102, 0);
```

change-instrument message  
in channel 1 (musician)  
to instrument 102



## change the instrument and note

# Version 2: Using command-line args to experiment with sounds

This version still plays just a single note, but you get to use command-line arguments to change the instrument and note. Experiment by passing in two int values from 0 to 127. The first int sets the instrument; the second int sets the note to play.

```
import javax.sound.midi.*;
import static javax.sound.midi.ShortMessage.*;

public class MiniMusicCmdLine {
    public static void main(String[] args) {
        MiniMusicCmdLine mini = new MiniMusicCmdLine();
        if (args.length < 2) {
            System.out.println("Don't forget the instrument and note args");
        } else {
            int instrument = Integer.parseInt(args[0]);
            int note = Integer.parseInt(args[1]);
            mini.play(instrument, note);
        }
    }

    public void play(int instrument, int note) {
        try {
            Sequencer player = MidiSystem.getSequencer();
            player.open();
            Sequence seq = new Sequence(Sequence.PPQ, 4);
            Track track = seq.createTrack();

            ShortMessage msg1 = new ShortMessage();
            msg1.setMessage(PROGRAM_CHANGE, 1, instrument, 0);
            MidiEvent changeInstrument = new MidiEvent(msg1, 1);
            track.add(changeInstrument);

            ShortMessage msg2 = new ShortMessage();
            msg2.setMessage(NOTE_ON, 1, note, 100);
            MidiEvent noteOn = new MidiEvent(msg2, 1);
            track.add(noteOn);

            ShortMessage msg3 = new ShortMessage();
            msg3.setMessage(NOTE_OFF, 1, note, 100);
            MidiEvent noteOff = new MidiEvent(msg3, 16);
            track.add(noteOff);

            player.setSequence(seq);
            player.start();
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }
}
```

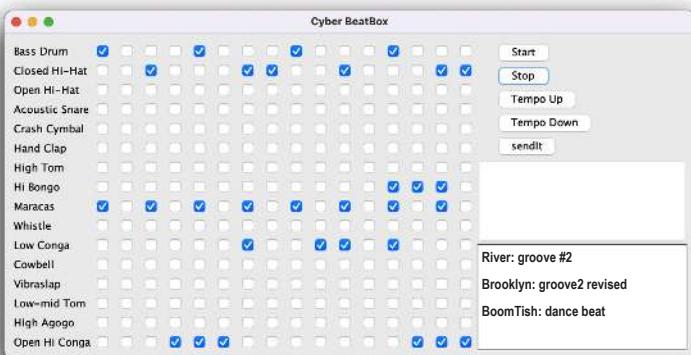
Run it with two int args from 0 to 127. Try these for starters:

```
File Edit Window Help Attenuate
%java MiniMusicCmdLine 102 30
%java MiniMusicCmdLine 80 20
%java MiniMusicCmdLine 40 70
```

# Where we're headed with the rest of the CodeKitchens

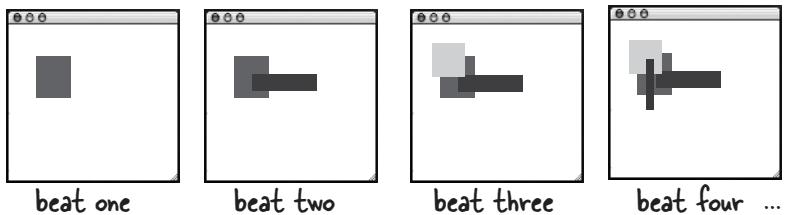
## Chapter 17: the goal

When we're done, we'll have a working BeatBox that's also a Drum Chat Client. We'll need to learn about GUIs (including event handling), I/O, networking, and threads. The next three chapters (14, 15, and 16) will get us there.



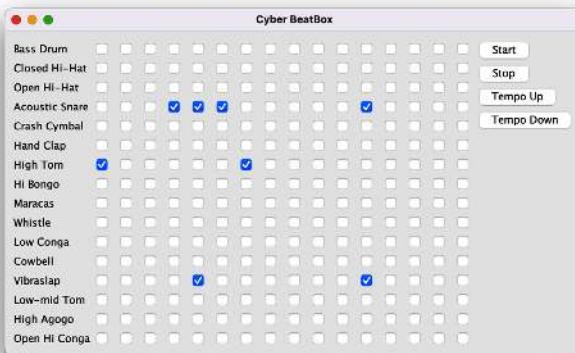
## Chapter 14: MIDI events

This CodeKitchen lets us build a little "music video" (bit of a stretch to call it that...) that draws random rectangles to the beat of the MIDI music. We'll learn how to construct and play a lot of MIDI events (instead of just a couple, as we do in the current chapter).



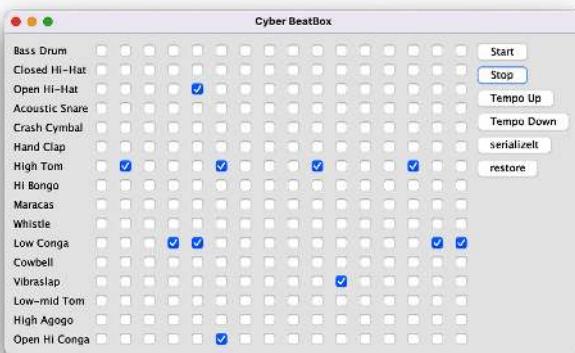
## Chapter 15: Standalone BeatBox

Now we'll actually build the real BeatBox, GUI and all. But it's limited—as soon as you change a pattern, the previous one is lost. There's no Save and Restore feature, and it doesn't communicate with the network. (But you can still use it to work on your drum pattern skills.)



## Chapter 16: Save and Restore

You've made the perfect pattern, and now you can save it to a file and reload it when you want to play it again. This gets us ready for the final version (Chapter 15), where instead of writing the pattern to a file, we send it over a network to the chat server.



## exercise: True or False



This chapter explored the wonderful world of exceptions. Your job is to decide whether each of the following exception-related statements is true or false.

## TRUE OR FALSE

1. A try block must be followed by a catch and a finally block.
2. If you write a method that might cause a compiler-checked exception, you must wrap that risky code in a try/catch block.
3. Catch blocks can be polymorphic.
4. Only “compiler checked” exceptions can be caught.
5. If you define a try/catch block, a matching finally block is optional.
6. If you define a try block, you can pair it with a matching catch or finally block, or both.
7. If you write a method that declares that it can throw a compiler-checked exception, you must also wrap the exception throwing code in a try/catch block.
8. The main() method in your program must handle all unhandled exceptions thrown to it.
9. A single try block can have many different catch blocks.
10. A method can throw only one kind of exception.
11. A finally block will run regardless of whether an exception is thrown.
12. A finally block can exist without a try block.
13. A try block can exist by itself, without a catch block or a finally block.
14. Handling an exception is sometimes referred to as “ducking.”
15. The order of catch blocks never matters.
16. A method with a try block and a finally block can optionally declare a checked exception.
17. Runtime exceptions must be handled or declared.

—————> Answers on page 457.



## Code Magnets

A working Java program is scrambled up on the fridge. Can you reconstruct all the code snippets to make a working Java program that produces the output listed below? Some of the curly braces fell on the floor and they were too small to pick up, so feel free to add as many of those as you need!

```

System.out.print("r");
try {
    System.out.print("t");
    doRisky(test);
}
System.out.println("s");
} finally {
    System.out.print("o");
}

class MyEx extends Exception { }

public class ExTestDrive {

    System.out.print("w");
    if ("yes".equals(t)) {
        System.out.print("a");
        throw new MyEx();
    } catch (MyEx e) {
}

static void doRisky(String t) throws MyEx {
    System.out.print("h");
}

public static void main(String [] args) {
    String test = args[0];
}

```

File Edit Window Help ThrowUp

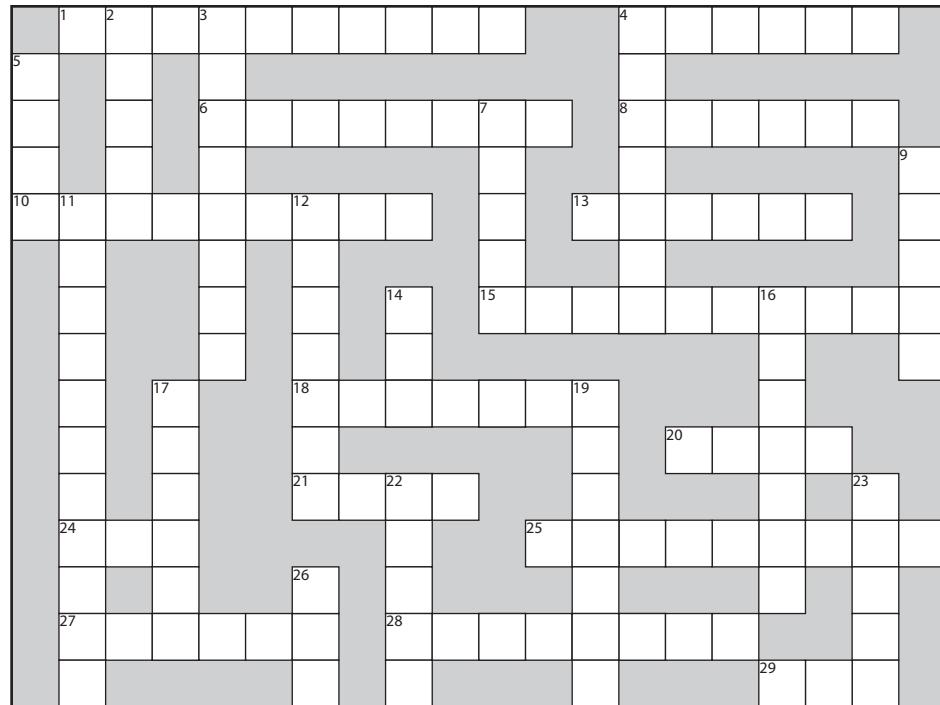
```
% java ExTestDrive yes
thaws
```

```
% java ExTestDrive no
throws
```

→ Answers on page 458.



→ Answers on page 459.



You know what  
to do!

### Across

- 1. To give value
- 4. Flew off the top
- 6. All this and more!
- 8. Start
- 10. The family tree
- 13. No ducking
- 15. Problem objects
- 18. One of Java's '49'
- 20. Class hierarchy
- 21. Too hot to handle
- 24. Common primitive
- 25. Code recipe
- 27. Unruly method action
- 28. No Picasso here
- 29. Start a chain of events

### Down

- 2. Currently usable
- 3. Template's creation
- 4. Don't show the kids
- 5. Mostly static API class
- 7. Not about behavior
- 9. The template
- 11. Roll another one off the line
- 12. Javac saw it coming
- 14. Attempt risk
- 16. Automatic acquisition
- 17. Changing method
- 19. Announce a duck
- 22. Deal with it
- 23. Create bad news
- 26. One of my roles

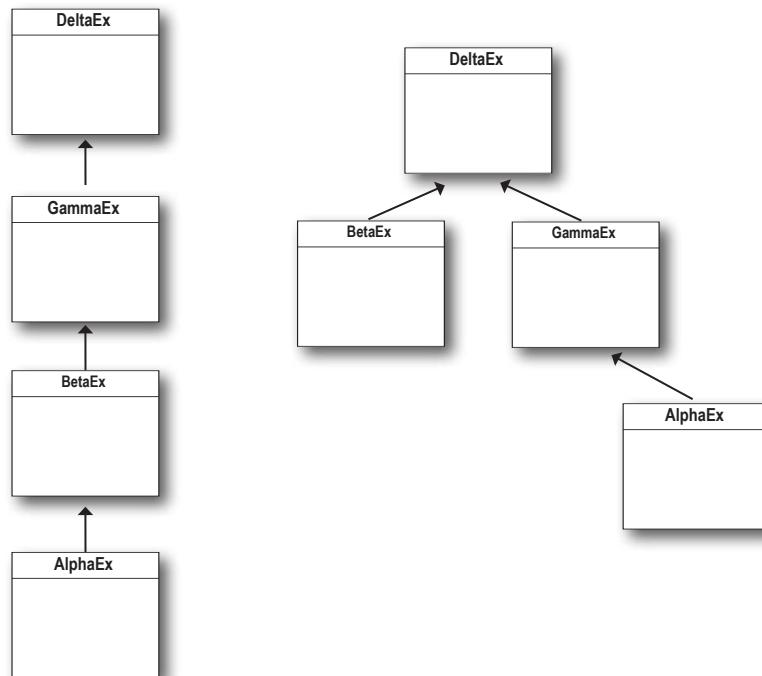
### More Hints:

- |        |                           |                               |                              |                            |                        |
|--------|---------------------------|-------------------------------|------------------------------|----------------------------|------------------------|
| Across | 6. A Java child           | 20. Also a type of collection | 21. Duck                     | 27. Starts a problem       | 28. Not Abstract       |
|        | 8. Starts a method        | 2. Oral mouthwash             | 16. _____ the family fortune | 3. For _____ (not example) | 17. Not a better       |
|        | 9. Only public or default | 15. Mouthwash                 | 5. Numbers ...               | 4. Mouthwash               | 13. Instead of declare |

# Solution



(from page 440)



## Exercise Solution

### TRUE OR FALSE (from page 454)

1. False, either or both.
2. False, you can declare the exception.
3. True.
4. False, runtime exception can be caught.
5. True.
6. True, both are acceptable.
7. False, the declaration is sufficient.
8. False, but if it doesn't, the JVM may shut down.
9. True.
10. False.
11. True. It's often used to clean up partially completed tasks.
12. False.
13. False.
14. False, ducking is synonymous with declaring.
15. False, broadest exceptions must be caught by the last catch blocks.
16. False, if you don't have a catch block, you must declare.
17. False.



## Exercise Solutions

### Code Magnets (from page 455)

```
class MyEx extends Exception { }

public class ExTestDrive {
    public static void main(String[] args) {
        String test = args[0];
        try {
            System.out.print("t");
            doRisky(test);
            System.out.print("o");
        } catch (MyEx e) {
            System.out.print("a");
        } finally {
            System.out.print("w");
        }
        System.out.println("s");
    }

    static void doRisky(String t) throws MyEx {
        System.out.print("h");

        if ("yes".equals(t)) {
            throw new MyEx();
        }

        System.out.print("r");
    }
}
```

```
File Edit Window Help Chill
% java ExTestDrive yes
throws

% java ExTestDrive no
throws
```

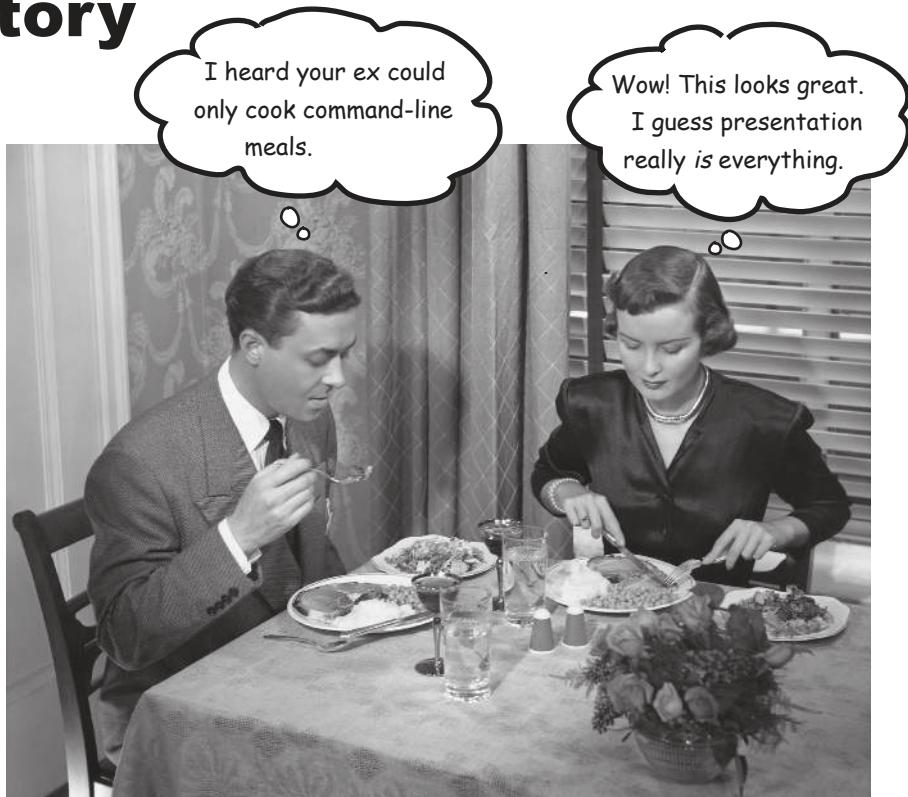


# JavaCross (from page 456)

|    |        |   |              |    |            |           |    |
|----|--------|---|--------------|----|------------|-----------|----|
|    | 1      | 2 | ASSIGNMENT   |    | 4          | POPPED    |    |
| 5  | M      | C | N            |    | R          |           |    |
| A  | O      | S | SUBCLASS     | 7  | I          | INVOKE    | 9  |
| T  | P      | T |              | 8  | V          |           | C  |
| H  | I      | E | 11 HIERARCHY | 12 | A          | 13 HANDLE | L  |
| N  | N      | N |              | 14 | T          | T         | A  |
| S  | C      | E |              | 15 | EXCEPTIONS |           | S  |
| T  | E      | C | R            |    |            | N         |    |
| A  | S      | K | KEYWORD      | 18 |            | H         |    |
| N  | E      | E |              | 19 |            |           |    |
| T  | T      | D | DUCK         | 21 | E          | 20 TREE   | 23 |
| I  | N      | A |              | 22 | C          | R         | T  |
| A  | E      | T |              | 25 | ALGORITHM  | M         |    |
| 27 | THROWS | I |              | 26 | A          | T         | R  |
| E  |        | A |              | 28 | CONCRETE   |           | O  |
|    |        | H |              | 29 | E          | NEW       |    |



# A Very Graphic Story



**Face it, you need to make GUIs.** If you're building applications that other people are going to use, you *need* a graphical interface. If you're building programs for yourself, you *want* a graphical interface. Even if you believe that the rest of your natural life will be spent writing server-side code, where the client user interface is a web page, sooner or later you'll need to write tools, and you'll want a graphical interface. Sure, command-line apps are retro, but not in a good way. They're weak, inflexible, and unfriendly. We'll spend two chapters working on GUIs and learn key Java language features along the way including **Event Handling** and **Inner Classes** and **lambdas**. In this chapter, we'll put a button on the screen, and make it do something when you click it. We'll paint on the screen, we'll display a JPEG image, and we'll even do some (crude) animation.

## It all starts with a window

A JFrame is the object that represents a window on the screen. It's where you put all the interface things like buttons, check boxes, text fields, and so on. It can have an honest-to-goodness menu bar with menu items. And it has all the little windowing icons for whatever platform you're on, for minimizing, maximizing, and closing the window.

The JFrame looks different depending on the platform you're on. This is a JFrame on an old Mac OS X:



## Put widgets in the window

Once you have a JFrame, you can put things ("widgets") in it by adding them to the JFrame. There are a ton of Swing components you can add; look for them in the javax.swing package. The most common include JButton, JRadioButton, JCheckBox, JLabel, JList, JScrollPane, JSlider, JTextArea, JTextField, and JTable. Most are really simple to use, but some (like JTable) can be a bit more complicated.

Two issues!

1. Swing? This looks like Swing code.
2. That window looks really old-fashioned.

**She's asked a couple of really good**

**questions.** In a few pages we'll address these questions with an extra-special "No Dumb Questions."



## Making a GUI is easy:

- ① Make a frame (a JFrame)

```
JFrame frame = new JFrame();
```

- ② Make a widget (button, text field, etc.)

```
JButton button = new JButton("click me");
```

- ③ Add the widget to the frame

```
frame.getContentPane().add(button);
```

You don't add things to the frame directly. Think of the frame as the trim around the window, and you add things to the window pane.

- ④ Display it (give it a size and make it visible)

```
frame.setSize(300,300);  
frame.setVisible(true);
```

## Your first GUI: a button on a frame

```

import javax.swing.*; ← don't forget to import
                      this swing package

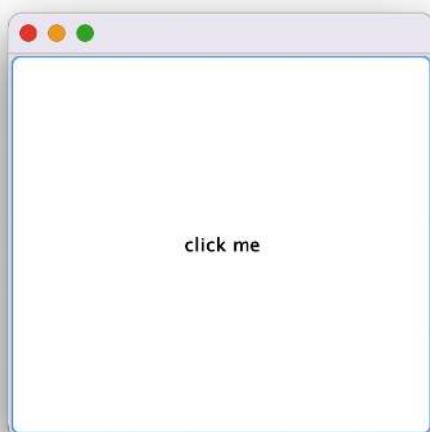
public class SimpleGuil {
    public static void main(String[] args) {
        JFrame frame = new JFrame(); ← make a frame and a button
        JButton button = new JButton("click me"); ← (you can pass the button constructor
  the text you want on the button)

        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE); ← this line makes the program quit as soon as you
  close the window (if you leave this out it will
  just sit there on the screen forever)
        frame.getContentPane().add(button); ← add the button to the frame's
   content pane
        frame.setSize(300, 300); ← give the frame a size, in pixels
        frame.setVisible(true); ← finally, make it visible!! (if you forget
                               this step, you won't see anything when
                               you run this code)
    }
}

```

### Let's see what happens when we run it:

%java SimpleGuil



Whoa! That's a  
Really Big Button.

The button fills all the  
available space in the frame.  
Later we'll learn to control  
where (and how big) the  
button is on the frame.

## But nothing happens when I click it...

That's not exactly true. When you press the button, it shows that "pressed" or "pushed in" look (which changes depending on the platform look and feel, but it always does *something* to show when it's being pressed).

The real question is, "How do I get the button to do something specific when the user clicks it?"

### We need two things:

- ① A **method** to be called when the user clicks (the thing you want to happen as a result of the button click).
- ② A way to **know** when to trigger that method. In other words, a way to know when the user clicks the button!



there are no  
Dumb Questions

**Q:** I heard that nobody uses Swing anymore.

**A:** There are other options, like JavaFX. But there are no clear winners in the endless and ongoing "Which approach should I use to make GUIs in Java?" debate. The good news is that if you learn a little Swing, that knowledge will help you whichever way you end up going. For example, if you want to do Android development, your Swing knowledge will make learning to code Android apps easier.

**Q:** Will a button look like a Windows button when you run on Windows?

**A:** If you want it to. You can choose from a few "look and feels"—classes in the core library that control what the interface looks like. In most cases you can choose between at least two different looks. The screens in this book use a number of "look and feels," including the default system look and feel (for macOS), the OS X **Aqua** look and feel, or the **Metal** (cross platform) look and feel.

**Q:** Isn't Aqua really old?

**A:** Yes, but we like it.

## Getting a user event

Imagine you want the text on the button to change from *click me* to *I've been clicked!* when the user presses the button. First we can write a method that changes the text of the button (a quick look through the API will show you the method):

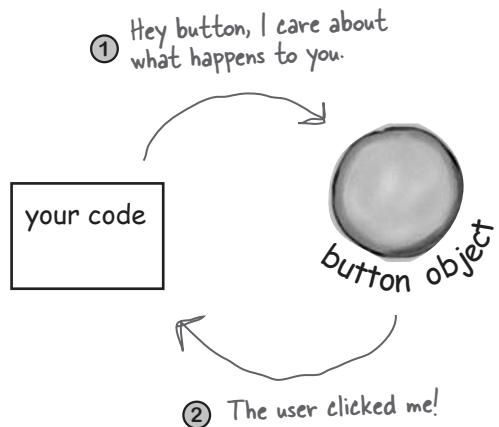
```
public void changeIt() {
    button.setText("I've been clicked!");
}
```

But *now* what? How will we *know* when this method should run? ***How will we know when the button is clicked?***

In Java, the process of getting and handling a user event is called *event-handling*. There are many different event types in Java, although most involve GUI user actions. If the user clicks a button, that's an event. An event that says “The user wants the action of this button to happen.” If it's a “Slow the Tempo” button, the user wants the slow-the-music-tempo action to occur. If it's a Send button on a chat client, the user wants the send-my-message action to happen. So the most straightforward event is when the user clicked the button, indicating they want an action to occur.

With buttons, you usually don't care about any intermediate events like button-is-being-pressed and button-is-being-released. What you want to say to the button is, “I don't care how the user plays with the button, how long they hold the mouse over it, how many times they change their mind and roll off before letting go, etc. ***Just tell me when the user means business!*** In other words, don't call me unless the user clicks in a way that indicates he wants the darn button to do what it says it'll do!”

**First, the button needs to know that we care.**



**Second, the button needs a way to call us back when a button-clicked event occurs.**



1. How could you tell a button object that you care about its events? That you're a concerned listener?

2. How will the button call you back? Assume that there's no way for you to tell the button the name of your unique method (`changeIt()`). So what else can we use to reassure the button that we have a specific method it can call when the event happens? [hint: think Pet]

If you care about the button's events,  
**implement an interface** that says,  
“I'm **listening** for your events.”

A **listener interface** is the bridge between the **listener** (you) and **event source** (the button).

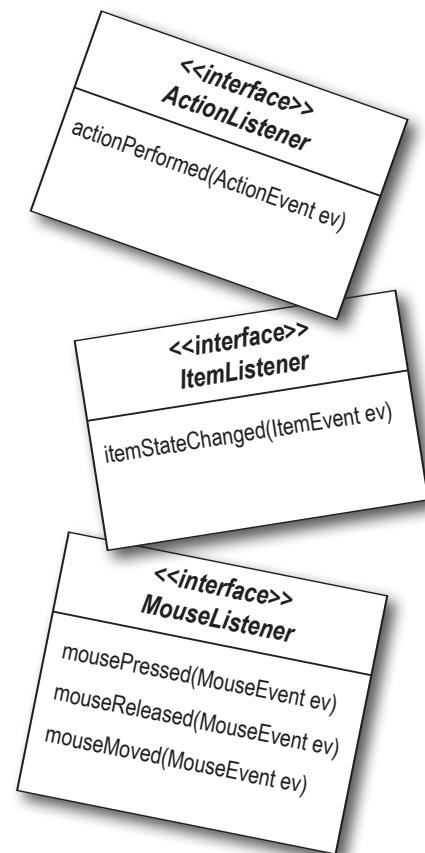
The Swing GUI components are event sources. In Java terms, an event source is an object that can turn user actions (click a mouse, type a key, close a window) into events. And like virtually everything else in Java, an event is represented as an object. An object of some event class. If you scan through the `java.awt.event` package in the API, you'll see a bunch of event classes (easy to spot—they all have **Event** in the name). You'll find `MouseEvent`, `KeyEvent`, `WindowEvent`, `ActionEvent`, and several others.

An event **source** (like a button) creates an **event object** when the user does something that matters (like *click* the button). Most of the code you write (and all the code in this book) will *receive* events rather than *create* events. In other words, you'll spend most of your time as an event *listener* rather than an event *source*.

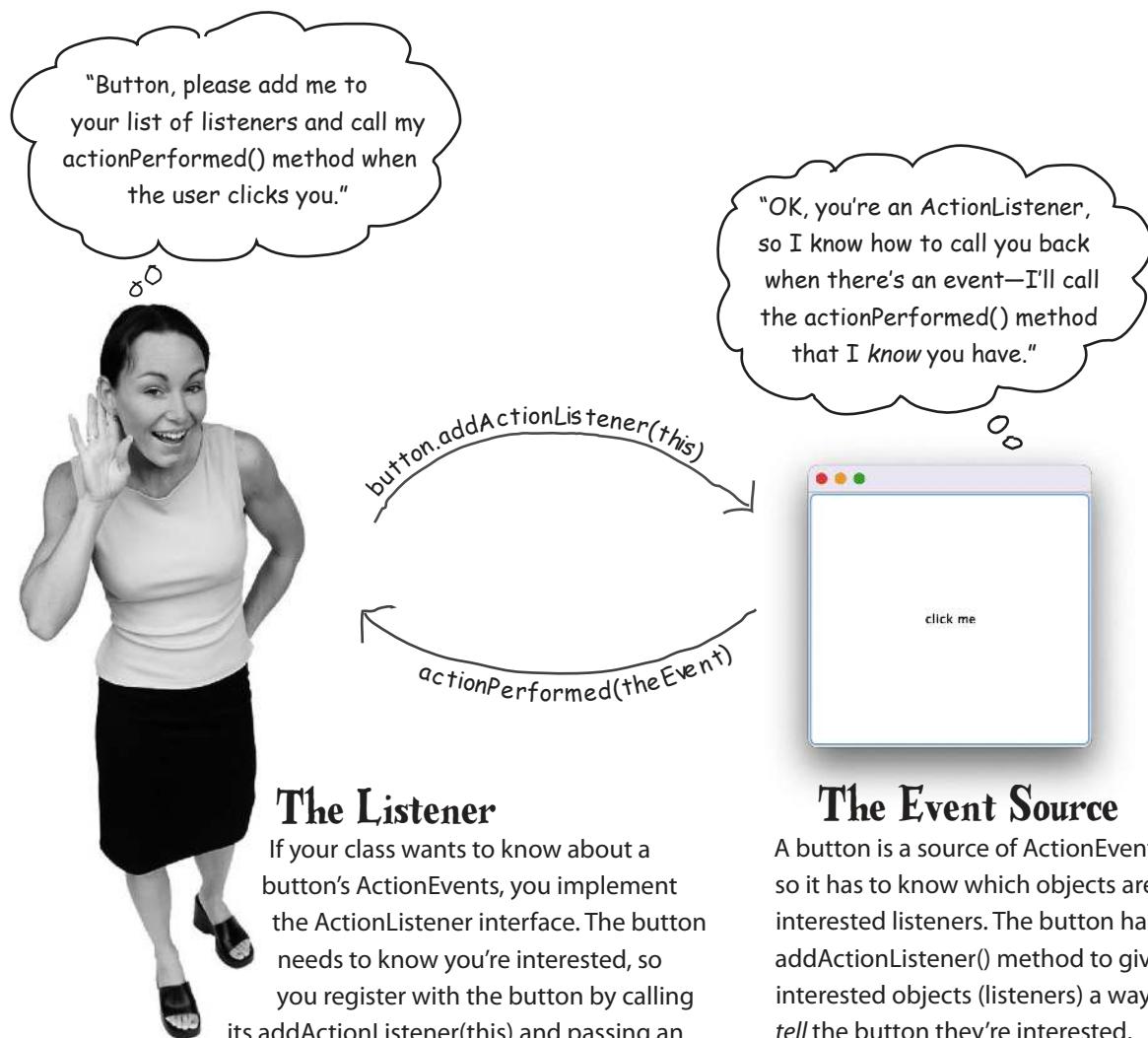
Every event type has a matching listener interface. If you want `MouseEvents`, implement the `MouseListener` interface. Want `WindowEvents`? Implement `WindowListener`. You get the idea. And remember your interface rules—to implement an interface you *declare* that you implement it (class `Dog` implements `Pet`), which means you must *write implementation methods* for every method in the interface.

Some interfaces have more than one method because the event itself comes in different flavors. If you implement `MouseListener`, for example, you can get events for `mousePressed`, `mouseReleased`, `mouseMoved`, etc. Each of those mouse events has a separate method in the interface, even though they all take a `MouseEvent`. If you implement `MouseListener`, the `mousePressed()` method is called when the user (you guessed it) presses the mouse. And when the user lets go, the `mouseReleased()` method is called. So for mouse events, there's only one event *object*, `MouseEvent`, but several different event *methods*, representing the different *types* of mouse events.

When you **implement a listener interface**, you give the button a way to call you back. The interface is where the call-back method is declared.



## How the listener and source communicate:



### The Listener

If your class wants to know about a button's ActionEvents, you implement the ActionListener interface. The button needs to know you're interested, so you register with the button by calling its `addActionListener(this)` and passing an ActionListener reference to it. In our first example, you are the ActionListener so you pass `this`, but it's more common to create a specific class to do listen to events. The button needs a way to call you back when the event happens, so it calls the method in the listener interface. As an ActionListener, you *must* implement the interface's sole method, `actionPerformed()`. The compiler guarantees it.

### The Event Source

A button is a source of ActionEvents, so it has to know which objects are interested listeners. The button has an `addActionListener()` method to give interested objects (listeners) a way to tell the button they're interested.

When the button's `addActionListener()` runs (because a potential listener invoked it), the button takes the parameter (a reference to the listener object) and stores it in a list. When the user clicks the button, the button "fires" the event by calling the `actionPerformed()` method on each listener in the list.

## Getting a button's ActionEvent

- ① Implement the ActionListener interface
- ② Register with the button (tell it you want to listen for events)
- ③ Define the event-handling method (implement the actionPerformed() method from the ActionListener interface)

```

import javax.swing.*;
import java.awt.event.*; ← A new import statement for the package
                           that ActionListener and ActionEvent are in.

①
public class SimpleGui2 implements ActionListener { ← Implement the interface. This says,
                           "an instance of SimpleGui2 IS-A
                           ActionListener."
                           (The button will give events only to
                           ActionListener implementers.)
private JButton button;

public static void main(String[] args) {
    SimpleGui2 gui = new SimpleGui2();
    gui.go();
}

public void go() {
    JFrame frame = new JFrame();
    button = new JButton("click me");
    ← button.addActionListener(this); ← Register your interest with the button. This says to
                                      the button, "Add me to your list of listeners." The
                                      argument you pass MUST be an object from a class
                                      that implements ActionListener!!
}

③
public void actionPerformed(ActionEvent event) {
    button.setText("I've been clicked!");
}

```

NOTE: You wouldn't usually make your main GUI class implement ActionListener like this; this is just the simplest way to get started. We'll see better ways of creating ActionListeners as we go through this chapter.

Implement the ActionListener interface's actionPerformed() method. This is the actual event-handling method!

The button calls this method to let you know an event happened. It sends you an ActionEvent object as the argument, but we don't need it here. Knowing the event happened is enough info for us.