O'REILLY®

Third Edition
Covers Java 8–17

Head First Java

A Learner's Guide to Real-World Programming

Kathy Sierra,
Bert Bates &
Trisha Gee

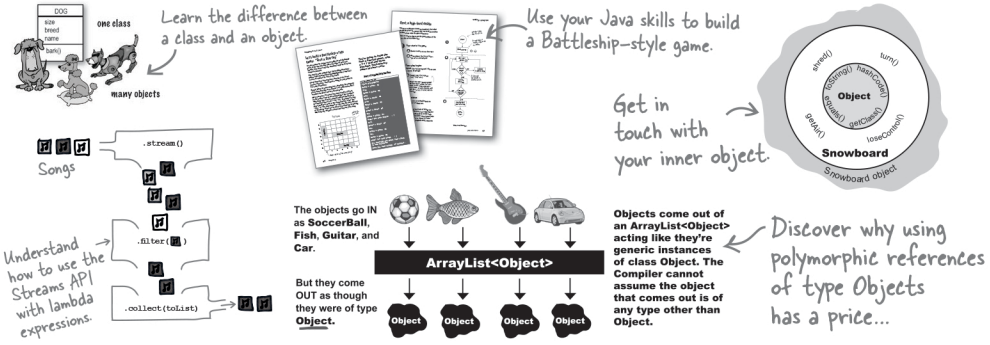
A Brain-Friendly Guide

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.

"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

JAVA

US $69.99 **CAN $87.99**

ISBN: 978-1-491-91077-1

9 781491 910771 56999

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)

"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 :-)."

— Maritjn 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 Buch* 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

"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

"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."

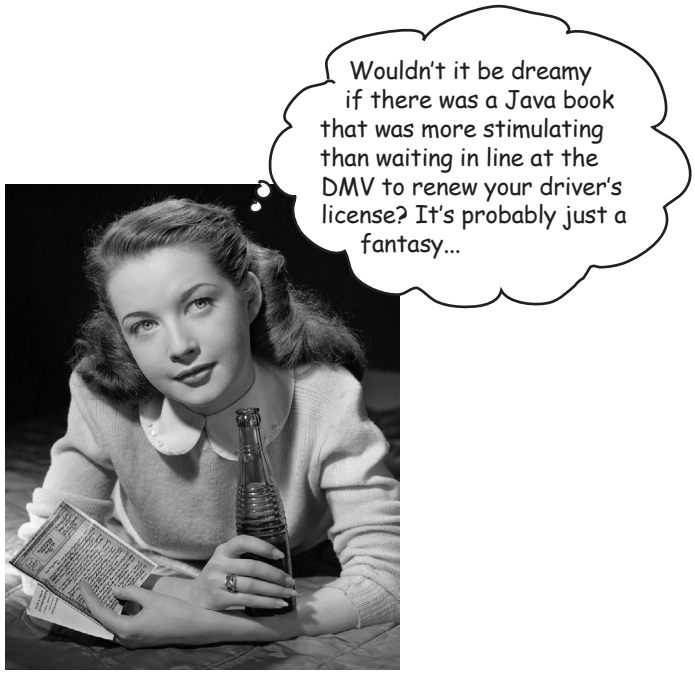
— Dr. Timothy A. Budd, Associate Professor of Computer Science at Oregon State University; author of more than a dozen books including *C++ for Java Programmers*

"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



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

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*). For more information, contact our corporate/institutional sales department: (800) 998-9938 or *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™)

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)

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 knowledgable 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.

Table of Contents (summary)

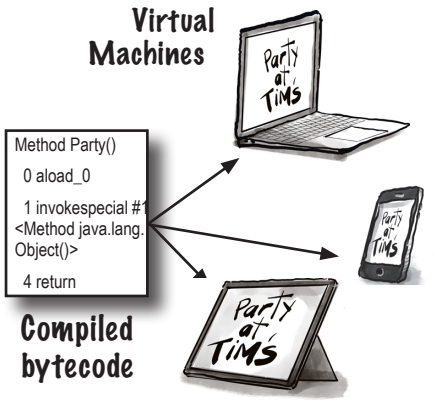
Table of Contents (the real thing)

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?

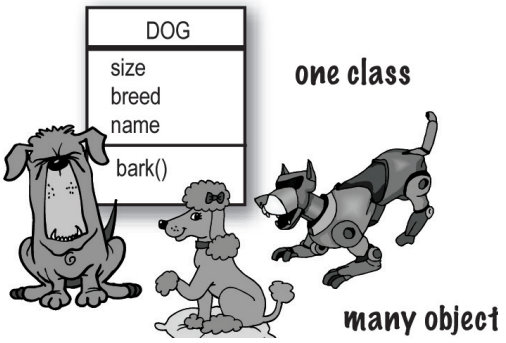
1 Breaking the Surface

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.



2 A Trip to Objectville

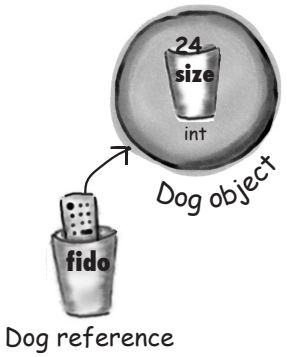
I was told there would be objects. In Chapter 1, we put all of our code in the main() method. That's not exactly object-oriented. So now we've got to leave that procedural world behind and start making some objects of our own. We'll look at what makes object-oriented (OO) development in Java so much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



3 Know Your Variables

Variables come in two flavors: primitive and reference.

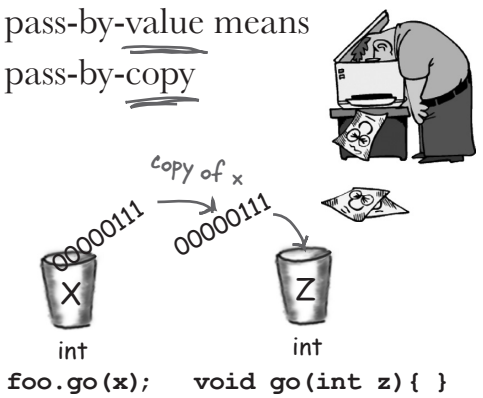
There's gotta be more to life than integers, Strings, and arrays. What if you have a PetOwner object with a Dog instance variable? Or a Car with an Engine? In this chapter we'll unwrap the mysteries of Java types and look at what you can *declare* as a variable, what you can *put* in a variable, and what you can *do* with a variable. And we'll finally see what life is truly like on the garbage-collectible heap.



4 How Objects Behave

State affects behavior, behavior affects state.

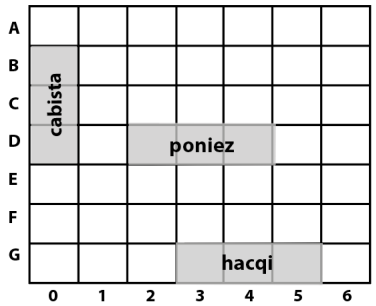
We know that objects have **state** and **behavior**, represented by **instance variables** and **methods**. Now we'll look at how state and behavior are *related*. An object's behavior uses an object's unique state. In other words, ***methods use instance variable values***. Like, "if dog weight is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



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



6 Using the Java Library

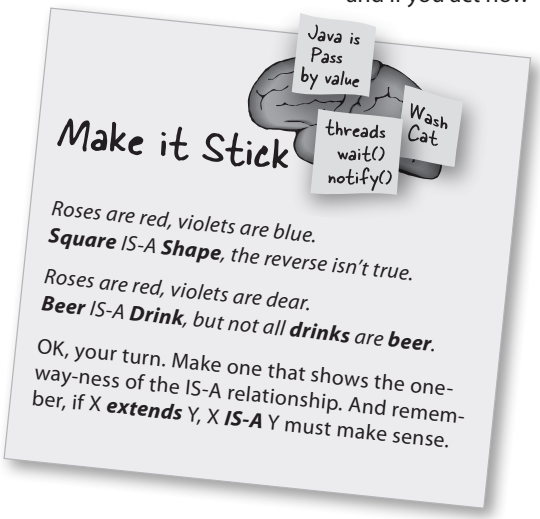
Java ships with hundreds of prebuilt classes. You don't have to reinvent the wheel if you know how to find what you need from the Java library, commonly known as the **Java API**. *You've got better things to do*. If you're going to write code, you might as well write *only* the parts that are custom 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

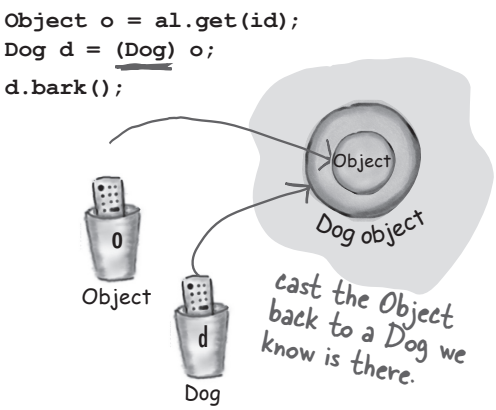
7 Better Living in Objectville

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.



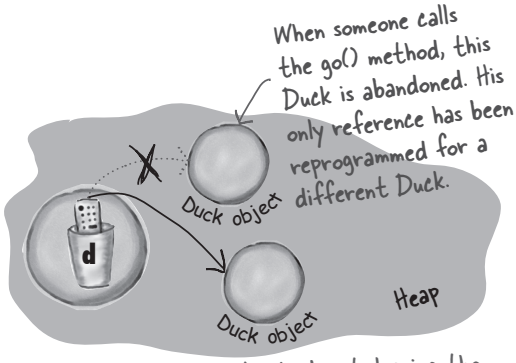
8 Serious Polymorphism

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...



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.

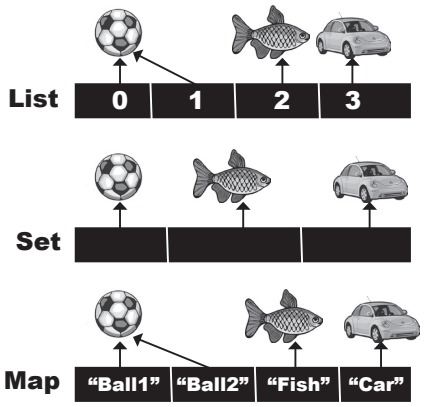
10 Numbers Matter

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.

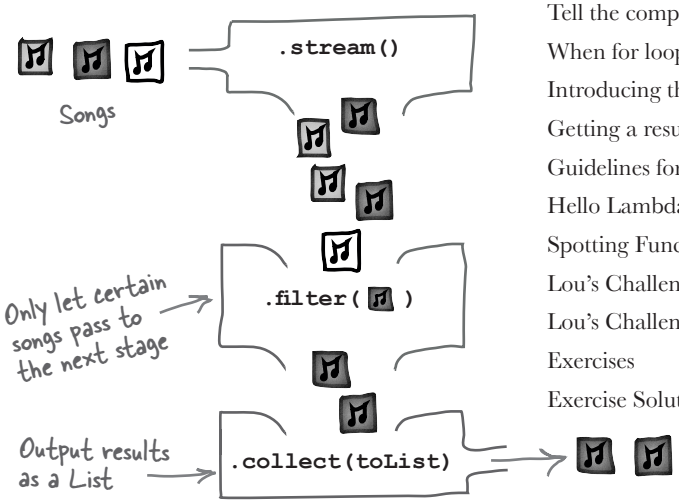
11 Data Structures

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?



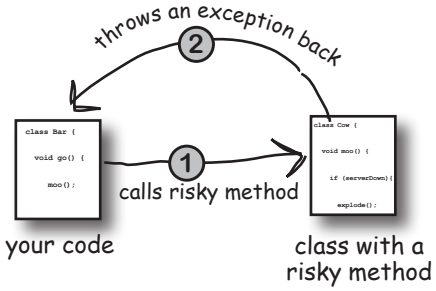
12 Lambdas and Streams: What, Not How

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.



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.

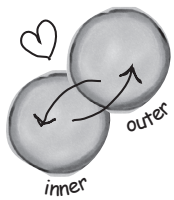


14 A Very Graphic Story

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() {
        }
    }

}
```

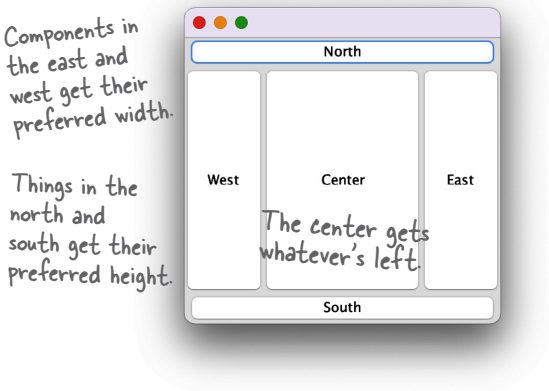
The outer and inner objects are now intimately linked.



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

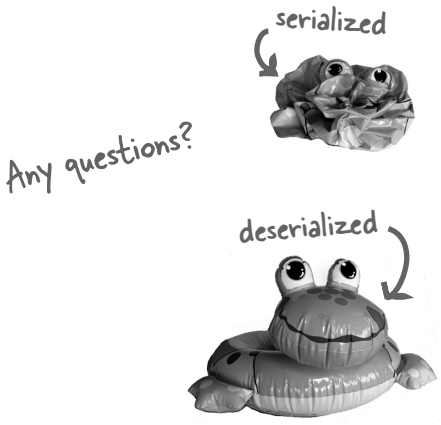
15 Work on Your Swing

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.



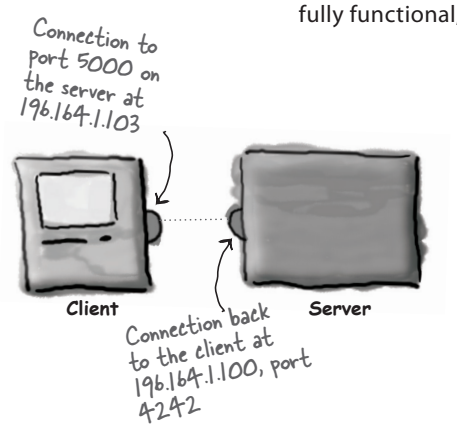
16 Saving Objects (and Text)

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.



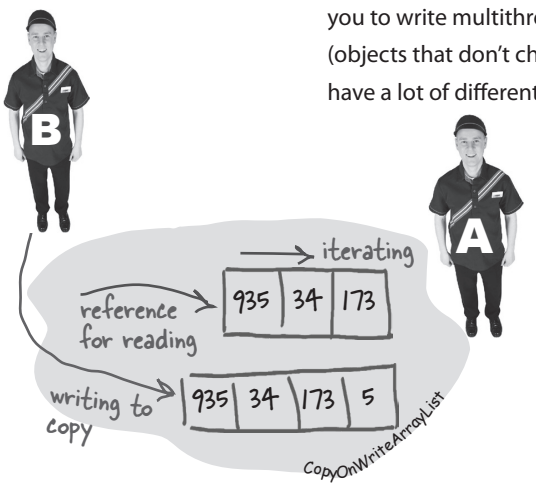
17 Make a Connection

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*?



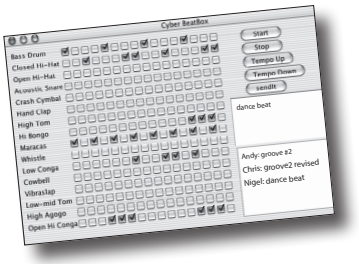
18 Dealing with Concurrency Issues

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.



A Appendix A

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



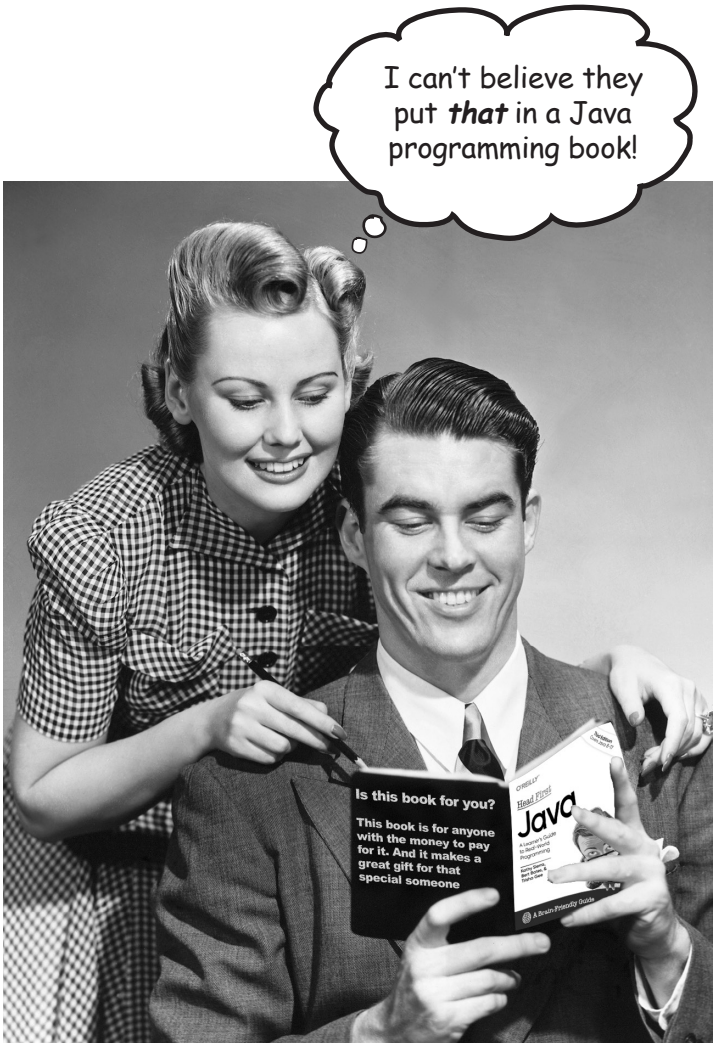
B Appendix B

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.

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.

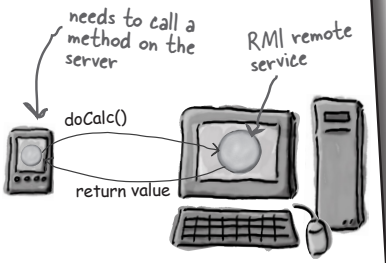
And there's no simple way to tell your brain, "Hey, brain, thank you very much, but no matter how dull this book is, and how little I'm registering on the emotional richter scale right now, I really *do* want you to keep this stuff around."

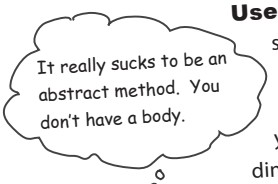
We think of a *Head First Java* reader as a learner.

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.

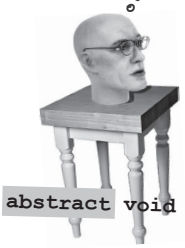
Some of the Head First learning principles:

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.

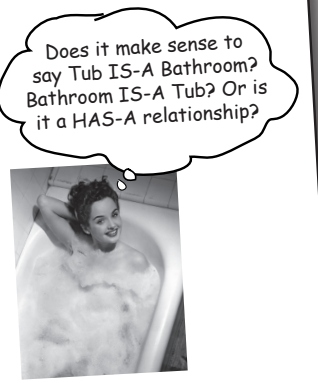




Use a conversational and personalized style. In recent studies, students performed up to 40% better on post-learning tests if the content spoke directly to the reader, using a first-person, conversational style rather than taking a formal tone. Tell stories instead of lecturing. Use casual language.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



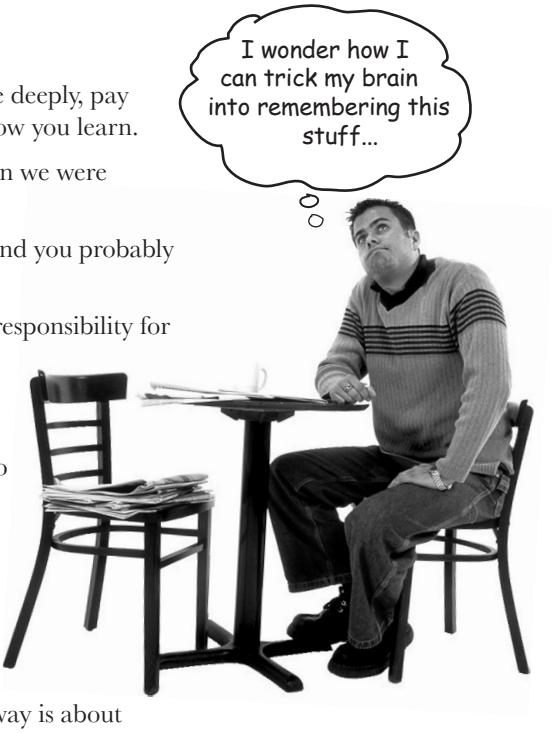
Get the learner to think more deeply. In other words, unless you actively flex your neurons, nothing much happens in your head. A reader has to be motivated, engaged, curious, and inspired to solve problems, draw conclusions, and generate new knowledge. And for that, you need challenges, exercises, thought-provoking questions, and activities that involve both sides of the brain and multiple senses.



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?

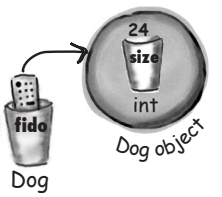
There's the slow, tedious way, or the faster, more effective way. The slow way is about sheer repetition. You obviously know that you *are* able to learn and remember even the dullest of topics, if you keep pounding on the same thing. With enough repetition, your brain says, "This doesn't *feel* important 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 makes 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.

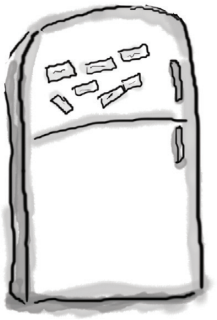
BULLET POINTS

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*.



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, expecially long examples and Ready-Bake Code, are at *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

  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.

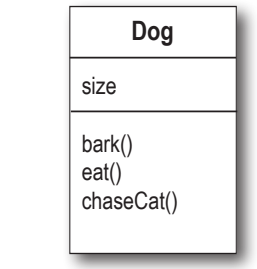
- Note: if you have trouble with your installation, we recommend you go to javaranch.com and join the Java-Beginning forum! Actually, you should do that whether you have trouble or not.

The code from this book is available at *https://oreil.ly/hfJava_3e_examples*.

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 a simpler, modified faux–UML



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.

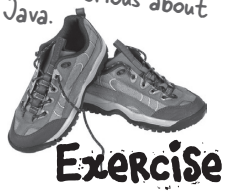
In this book, you can get on with the business of learning Java, without stressing over some of the organizational or administrative details of developing Java programs. You *will*, in the real world, need to know—and use—these details, but since building and deploying Java applications generally relies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

You should do ALL of the "Sharpen your pencil" activities

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.

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



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.

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.

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.

Technical Reviewers for the 3rd Edition

Marc Loy

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 Marin-Perez

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*.

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*.)

Other people to blame: credit, for the 2nd Edition

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:

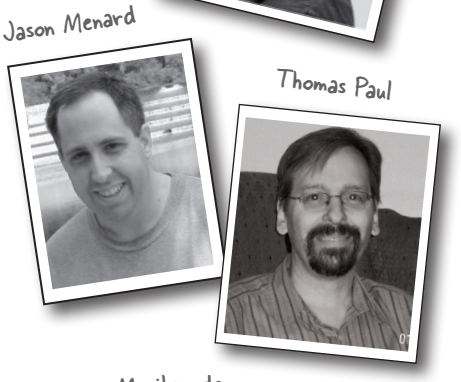
Our top honors and thanks go to the director of our javaranch tech review team, **Johannes de Jong**. This is your fifth time around with us on a Head First book, and we're thrilled you're still speaking to us. **Jeff Cumps** is on his third book with us now and relentless about finding areas where we 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). And thanks to our terrific agent, **David Rogelberg** of StudioB (but seriously, what about the *movie* rights?)

Some of our Java expert reviewers...



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, Suddhasatwa 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**), 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.

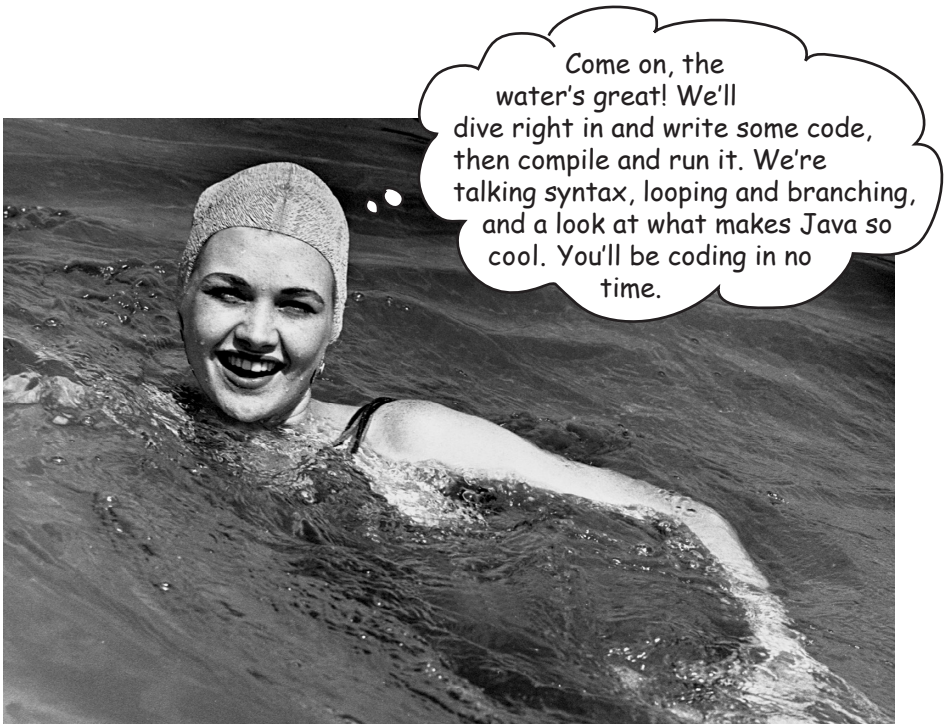
Sherry Dorris, for the things that really matter.

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.

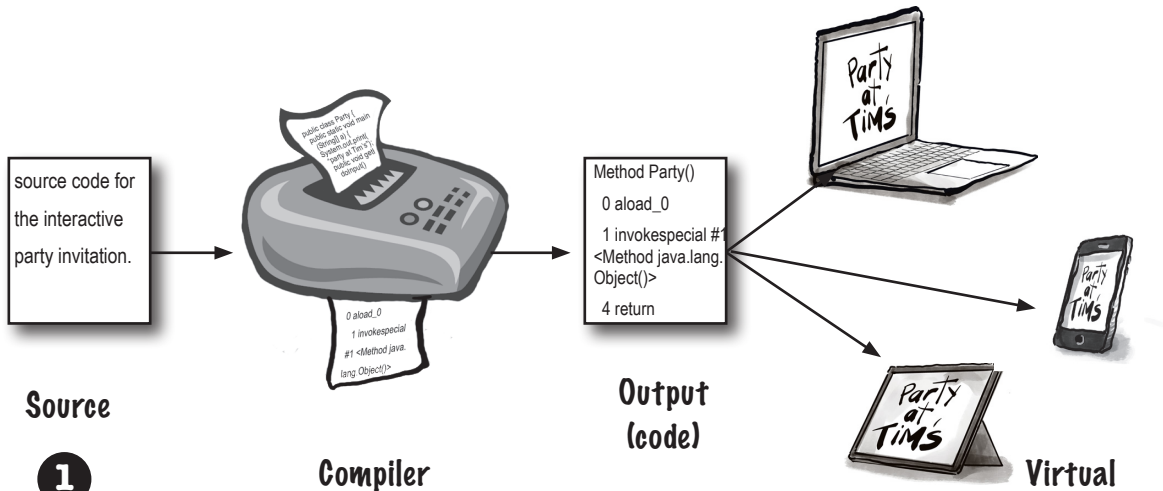
Breaking the Surface



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. 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.



Source

1

Create a source document. Use an established protocol (in this case, the Java language).

Compiler

2

Run your document through a source code compiler. The compiler checks for errors and won't let you compile until it's satisfied that everything will run correctly.

Output (code)

3

The compiler creates a new document, coded into Java ***bytecode***. Any device capable of running Java will be able to interpret/translate this file into something it can run. The compiled bytecode is platform-independent.

Virtual Machines

4

Your friends all have a Java ***virtual*** machine (JVM), implemented in software, running inside their electronic gadgets. When your friends run your program, the virtual machine reads and runs the bytecode.

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);
 }  // more code here...
}
```

Source

1

Type your source code.

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

```
File  Edit  Window  Help  Plead
%javac 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***

```
File  Edit  Window  Help  Swear
%java Party
```

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.

Sharpen your pencil

→ Answers on page 6.

Look how easy it is to write Java

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

```
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.");
}
```

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!

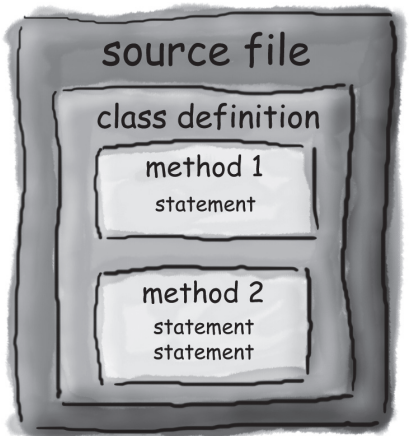
Sharpen your pencil answers

Look how easy it is to write Java

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...*

`int size = 27;`	declare an integer variable named 'size' and give it the value 27
`String name = "Fido";`	declare a string of characters variable named 'name' and give it the value "Fido"
`Dog myDog = new Dog(name, size);`	declare a new Dog variable 'myDog' and make the new Dog using 'name' and 'size'
`x = size - 5;`	subtract 5 from 27 (value of 'size') and assign it to a variable named 'x'
`if (x < 15) myDog.bark(8);`	if x (value of 22) is less than 15, tell the dog to bark 8 times
`while (x > 3) {`	keep looping as long as x is greater than 3...
`  myDog.play();`	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
`int[] numList = {2, 4, 6, 8};`	declare a list of integers variable 'numList', and put 2,4,6,8 into the list.
`System.out.print("Hello");`	print out "Hello"... probably at the command line
`System.out.print("Dog: " + name);`	print out "Dog: Fido" (the value of 'name' is "Fido") at the command line
`String num = "8";`	declare a character string variable 'num' and give it the value of "8"
`int z = Integer.parseInt(num);`	convert the string of characters "8" into an actual numeric value 8
`try {`	try to do something...maybe the thing we're trying isn't guaranteed to work...
`  readTheFile("myFile.txt");`	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...
`catch (FileNotFoundException ex) {`	this must be where you find out if the thing you tried didn't work...
`  System.out.print("File not found.");`	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;
   }
}
```

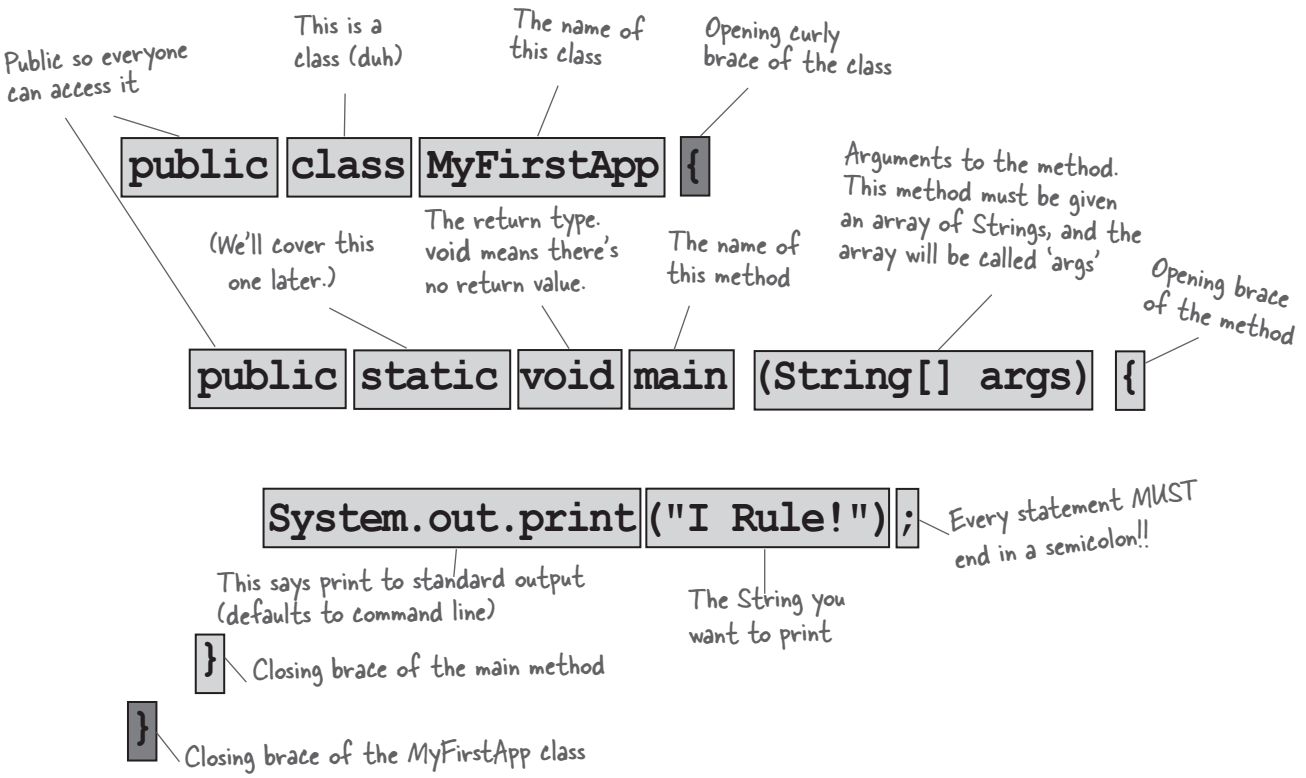
statements

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()

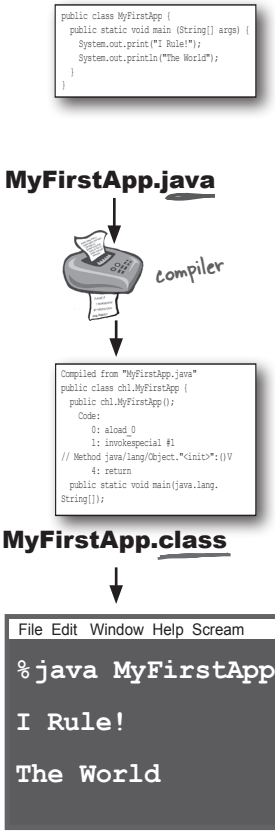
In Java, everything goes in a **class**. You'll type your source code file (with a *.java* extension), then compile it into a new class file (with a *.class* extension). When you run your program, you're really running a class.

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
```

2 Compile

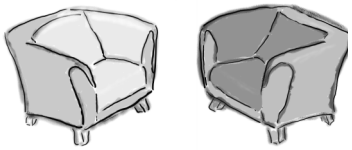
```
javac MyFirstApp.java
```

3 Run

```
java MyFirstApp
```

```
public class MyFirstApp {

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

}
```

Fireside Chats



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

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?

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 ClassCastExceptions 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:

1 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
```

2 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);
}
```

3 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..."

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");
  }
}
```

This is the output

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

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");
  }
}
```

Code output

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

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");
  }
}
```

New output

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

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 print***ln*** as **print*new*line**), 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 print**ln**. 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

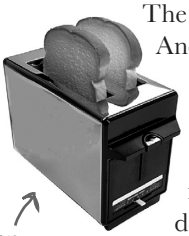
Bob's alarm clock rings at 8:30 Monday morning, just like every other weekday. But Bob had a wild weekend and reaches for the SNOOZE button. And that's when the action starts, and the Java-enabled appliances come to life...

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.

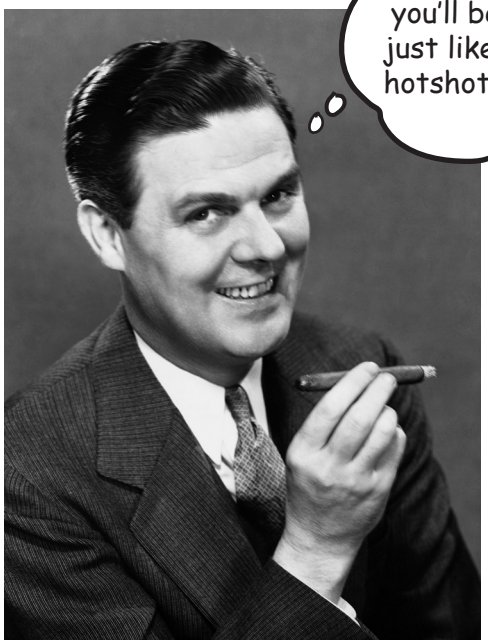
His coffee steams.

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.



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) {

    // 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"};

    // find out how many words are in each list
    int oneLength = wordListOne.length;
    int twoLength = wordListTwo.length;
    int threeLength = wordListThree.length;

    // 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);

    // now build a phrase
    String phrase = wordListOne[rand1] + " " +
wordListTwo[rand2] + " " + wordListThree[rand3];

    // 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 microservices pipeline

opinionated loosely coupled REST API

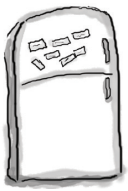
agent-based microservices library

AI-enabled service oriented orientation

agnostic pub-sub DSL

functional IoT perspective

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("-");
```

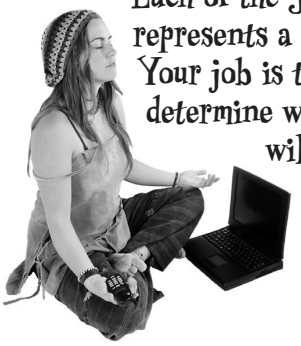
```
while (x > 0) {
```

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.

A

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

B

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

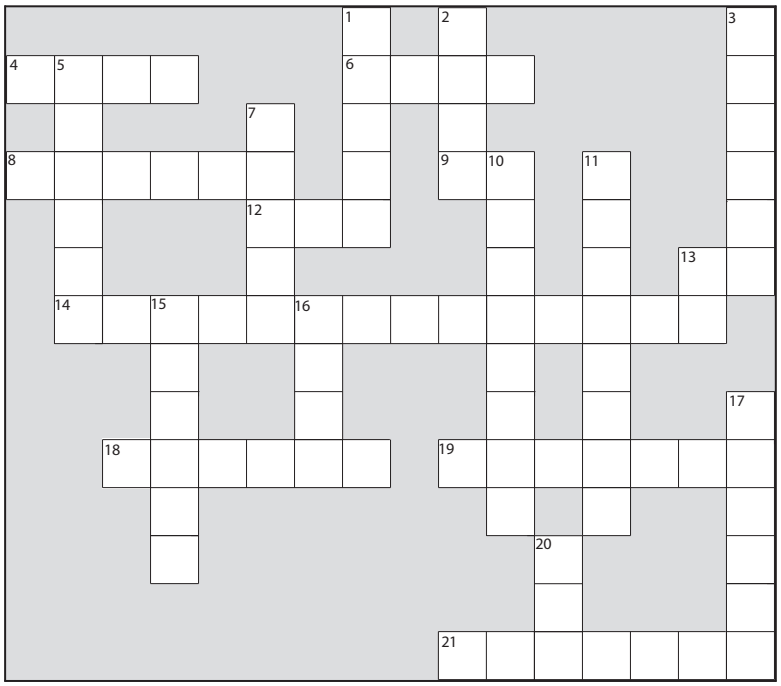
C

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

JavaCross

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

Match each candidate with one of the possible outputs

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
```

Answers on page 26.

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.

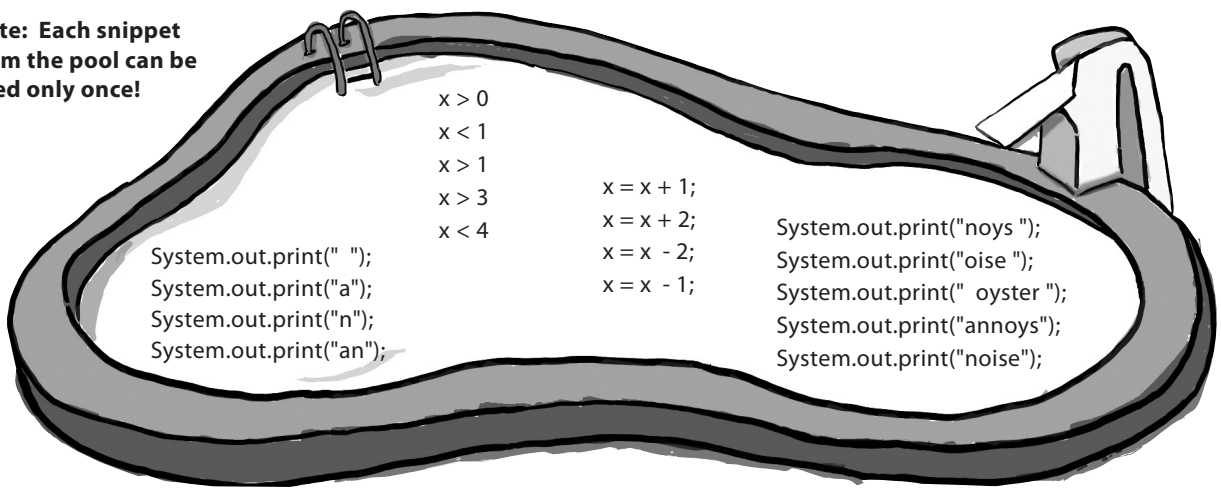
→ Answers on page 26.

Output

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

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

    while ( ___________ ) {

      _____________________________
      if ( x < 1 ) {
        _____________________________
      }
      _____________________________

      if ( ___________ ) {

        _____________________________

        ___________
      }
      if ( x == 1 ) {

        _____________________________
      }
      if ( ___________ ) {

        _____________________________
      }
      System.out.println();

      ___________
    }
  }
}
```

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



```
x > 0
x < 1
x > 1
x > 3
x < 4

x = x + 1;
x = x + 2;
x = x  - 2;
x = x  - 1;

System.out.print("  ");
System.out.print("a");
System.out.print("n");
System.out.print("an");

System.out.print("noys ");
System.out.print("oise ");
System.out.print("  oyster ");
System.out.print("annoys");
System.out.print("noise");
```

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

```
class Exercise1a {
  public static void main(String [] args) {
    int x = 1;
    while ( x < 10 ) {
      x = x + 1;          <- Add this line to prevent it running forever...
      if ( x > 3) {
        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
  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"
  public static void main(String [] args) {
    int x = 5;
    while ( x > 1 ) {
      x = x - 1;
      if ( x < 3) {
        System.out.println("small x");
      }
    }
  }
}
```

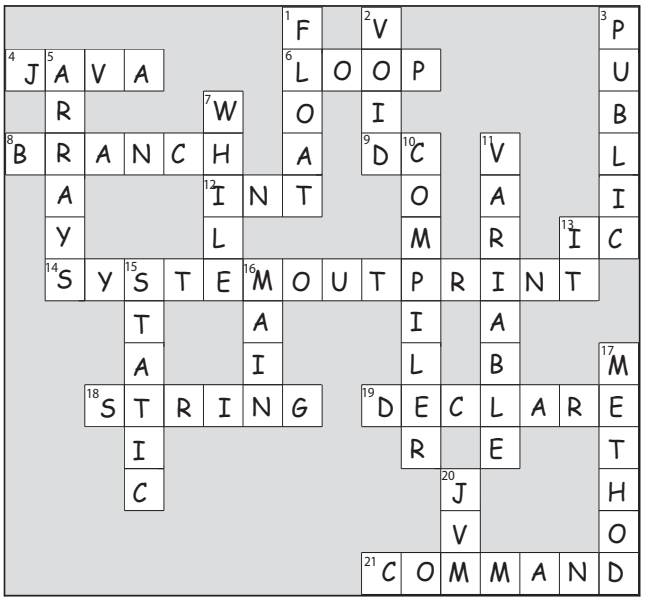
The while loop code must be inside a method. It can't just be hanging out inside the class.

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
```

JavaCross (from page 22)



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
```

(from page 23)

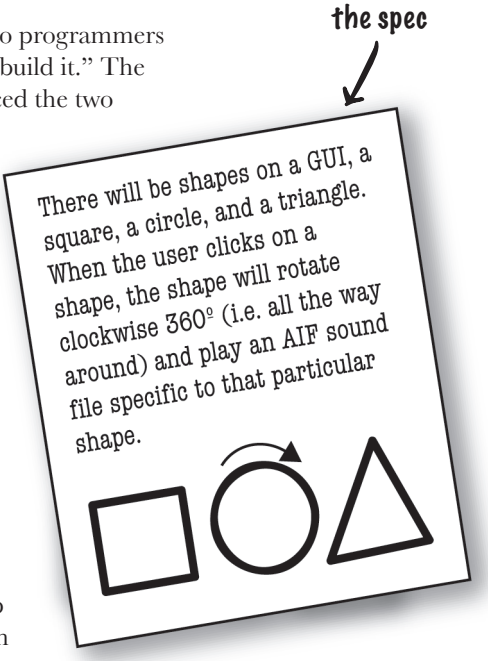
A Trip to Objectville

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?"***

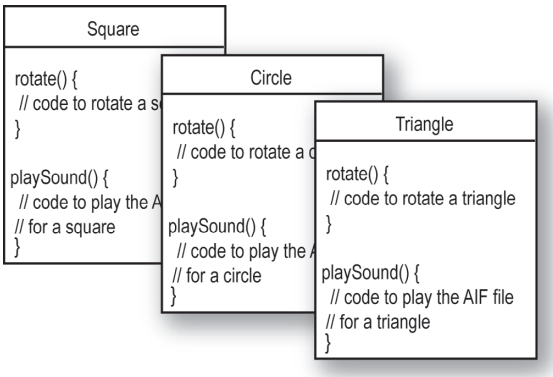
At Laura's desk

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.

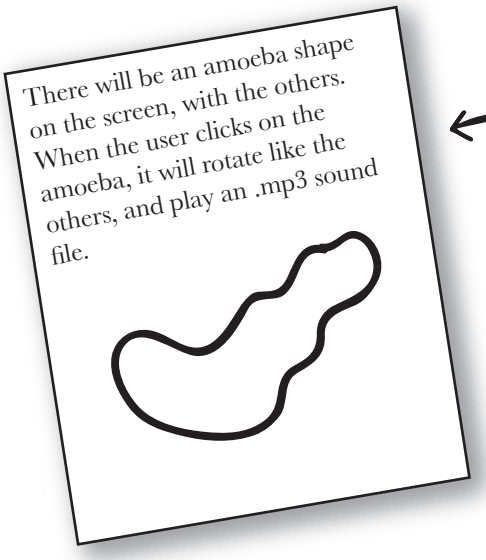


Laura thought she'd nailed it. She could almost feel the rolled steel of the Aeron beneath her...

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

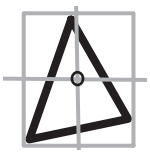
Brad smiled, sipped his fruit frappe, and *wrote one new class*. Sometimes the thing he loved most about OO was that he didn't have to touch code he'd already tested and delivered. "Flexibility, extensibility, ..." he mused, reflecting on the benefits of OO.

Amoeba

```
rotate() {
 // code to rotate an amoeba
 }
playSound() {
 // code to play the new
 // .mp3 file for an amoeba
 }
```

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..."

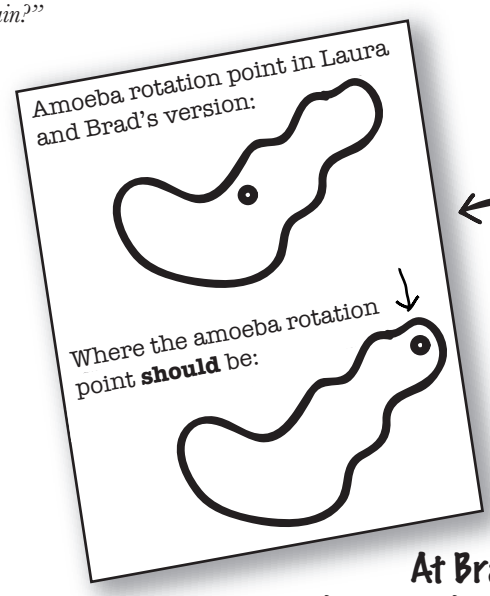


Turns out, both programmers had written their rotate code like this:

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, hmmmm. 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?"*



What the spec conveniently forgot to mention

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.

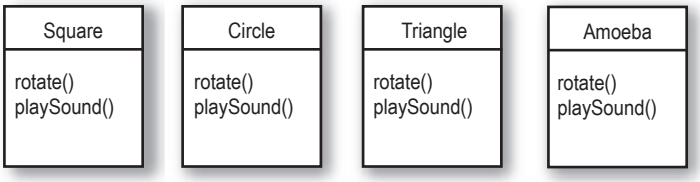
BRAD: It's a ***method***, not a *procedure*. And they're ***classes***, not *things*.

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)

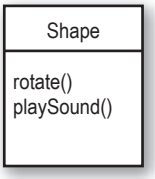


1

I looked at what all four classes have in common.

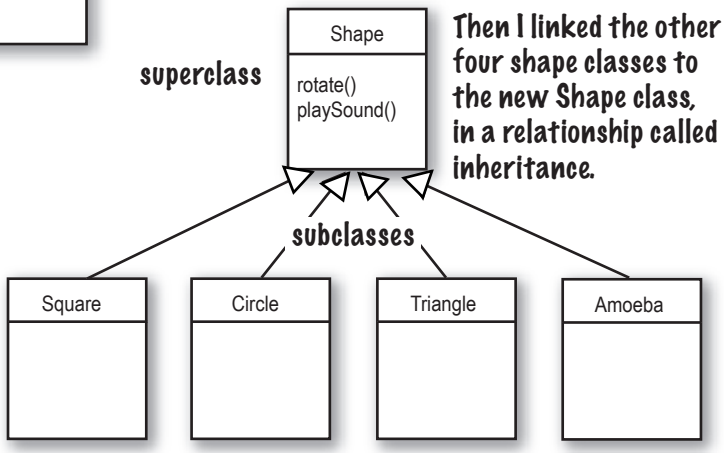
2

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.

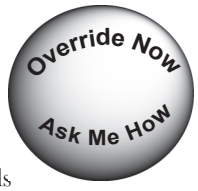
The Shape class is called the **superclass** of the other four classes. The other four are the **subclasses** of Shape. The subclasses inherit the methods of the superclass. In other words, *if the Shape class has the functionality, then the subclasses automatically get that same functionality.*

What about the Amoeba rotate()?

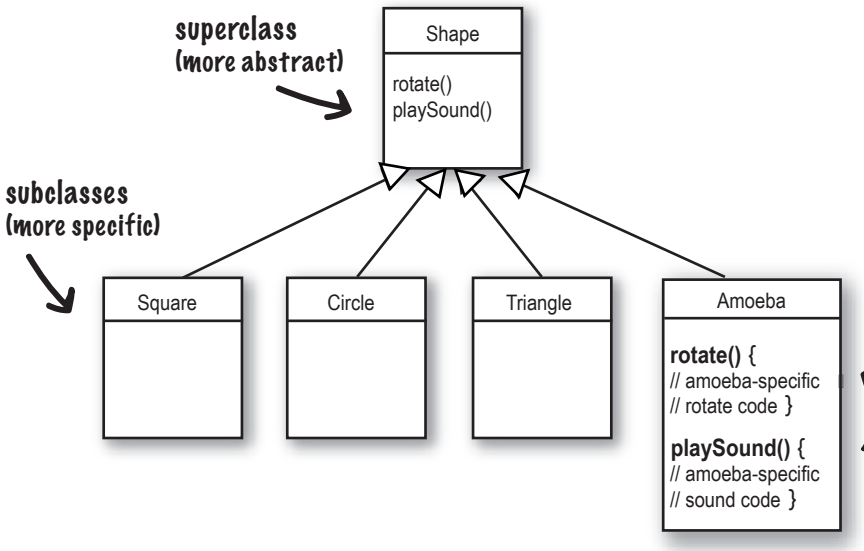
LAURA: Wasn't that the whole problem here—that the amoeba shape had a completely different rotate and playSound procedure?

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.



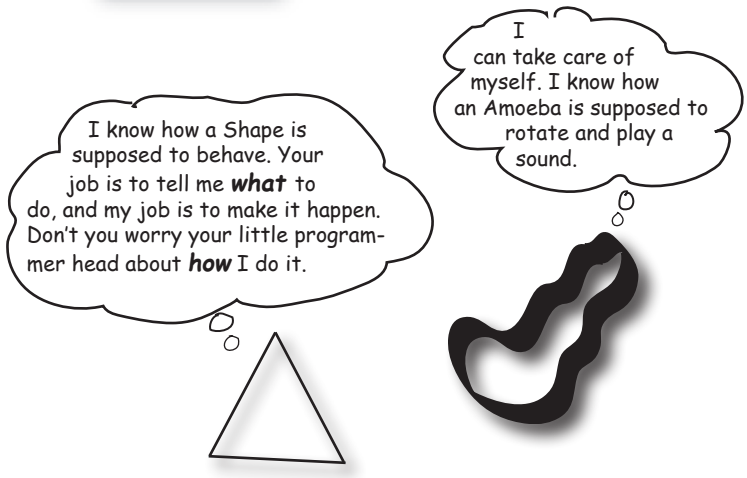
4

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.

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

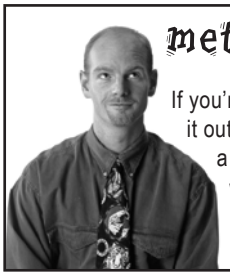
"Besides the chair?"
-Amy, 34, programmer

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?

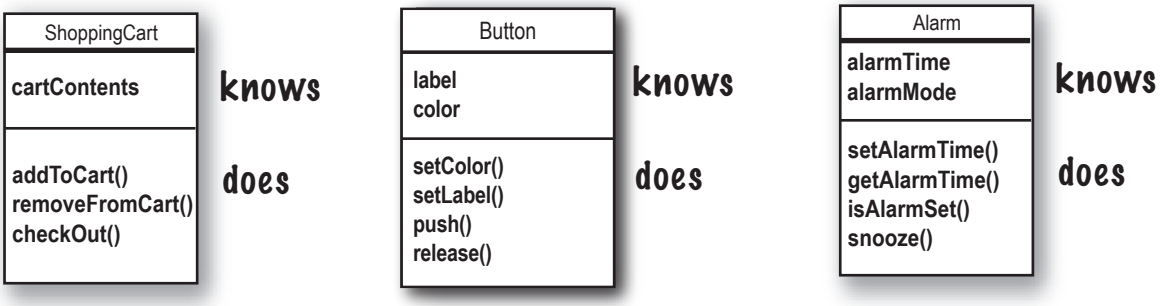


metacognitive tip

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**

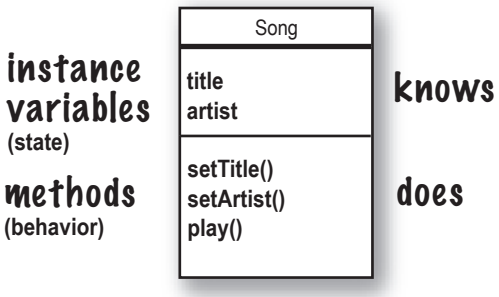


Things an object *knows* about itself are called

- instance variables

Things an object can *do* are called

- methods



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.

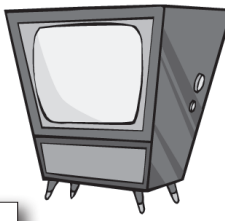
Think of instance as another way of saying object.

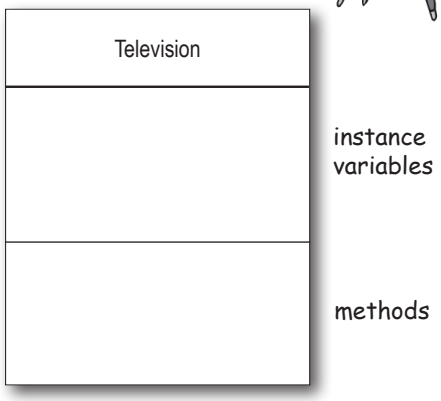
Things an object can ***do*** are called **methods**. When you design a class, you think about the data an object will need to know about itself, and you also design the methods that operate on that data. It's common for an object to have methods that read or write the values of the instance variables.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

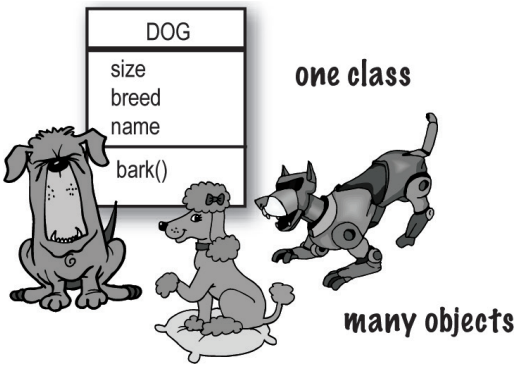
Fill in what a television object might need to know and do.





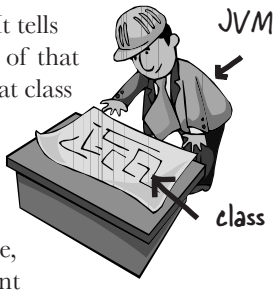
→ Yours to solve.

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



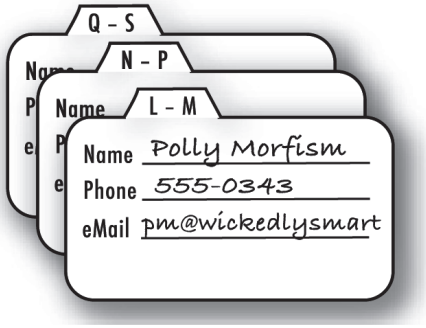
A class is not an object

(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*.

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;
```

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();
    d.size = 40;
    d.bark();
  }
}
```

Make a Dog object

Use the dot operator (.) to set the size of the Dog

and to call its bark() method

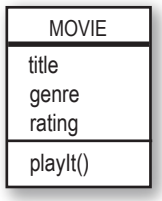
Dot operator

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;
  }
}
```

Sharpen your pencil



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 MovieTest-Drive 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!

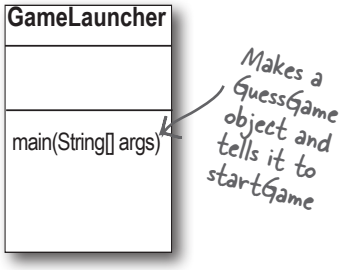
As long as you're in main(), you're not really in Objectville. It's fine for a test program to run within the main method, but in a true OO application, you need objects talking to other objects, as opposed to a static main() method creating and testing objects.

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.)

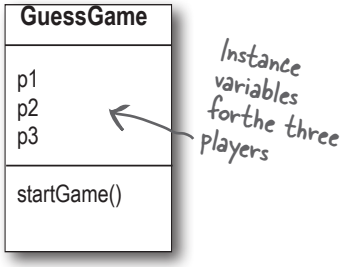
As a "sneak preview," though, of how a real Java application might behave, here's a little example. Because we're still at the earliest stages of learning Java, we're working with a small toolkit, so you'll find this program a little clunky and inefficient. You might want to think about what you could 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:

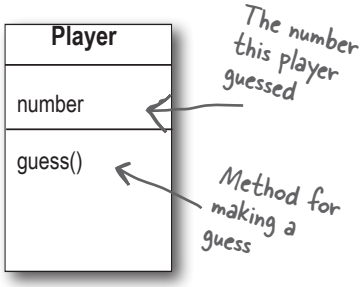
`GuessGame.class` `Player.class` `GameLauncher.class`

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

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

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

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

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

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

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

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

      if (p1isRight || p2isRight || p3isRight) {

        System.out.println("We have a winner!");
        System.out.println("Player one got it right? " + p1isRight);
        System.out.println("Player two got it right? " + p2isRight);
        System.out.println("Player three got it right? " + p3isRight);
        System.out.println("Game is over.");
        break; // 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

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.
```

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.

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).

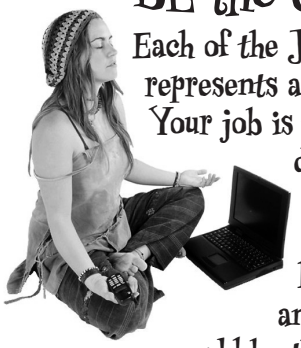
Q: What if I have a hundred classes? Or a thousand? Isn't that a big pain to deliver all those individual files? Can I bundle them into one *Application Thing*?

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 **J**ava **AR**chive—*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

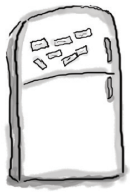
```
class Episode {

  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

Answers on page 46.

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.



```
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
```

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!

```
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);
  }
}
```

Output

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

Bonus Question !

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

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

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

```
x                x < 4
y                x < 5
e2               x > 0
count            x > 1

e1 = e1 + 1;
e1 = count + 1;
e1.count = count + 1;
e1.count = e1.count + 1;

Echo
Tester
echo( )
count( )
hello( )

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

x == 3
x == 4
```


Answers on page 47.

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.

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);
  }
}
```

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

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

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

B

```
class Episode {
  int seriesNumber;
  int episodeNumber;

  void play() {
    System.out.println("Playing episode " + episodeNumber);
  }

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

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

The line: episode.play(); wouldn't compile without a play method in the episode class!

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

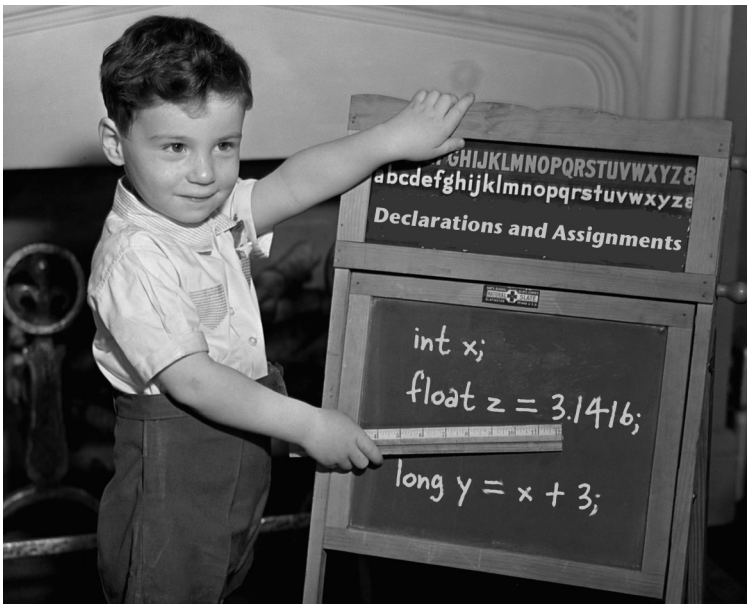
```
File  Edit  Window  Help  Assimilate
%java EchoTestDrive
helloooo...
helloooo...
helloooo...
helloooo...
10
```

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

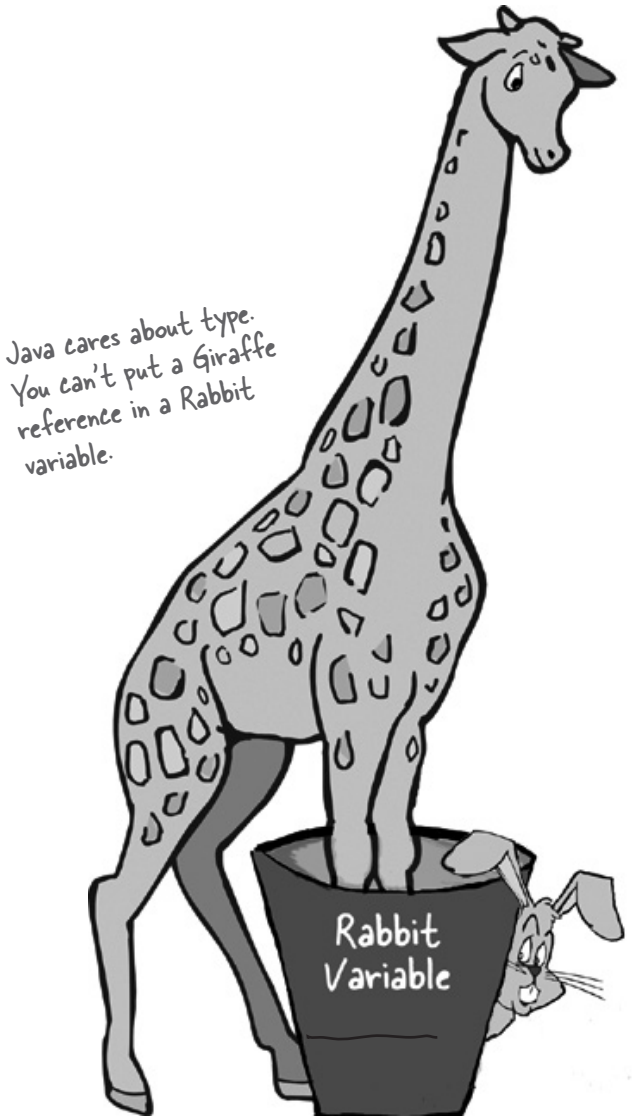
Note: both classes and objects are said to have state and behavior. They're defined in the class, but the object is also said to "have" them. Right now, we don't care where they *technically* live.

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 primited 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 → int

name → count

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."

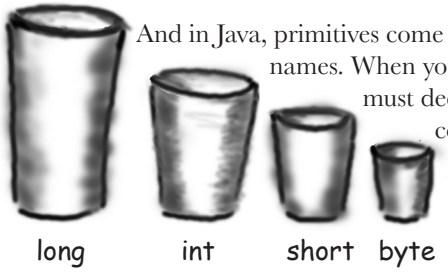
When you think of Java variables, think of cups. Coffee cups, tea cups, giant cups that hold lots and lots of your favorite drink, those big cups the popcorn comes in at the movies, cups with wonderful tactile handles, and cups with metallic trim that you learned can never, ever go in the microwave.

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.

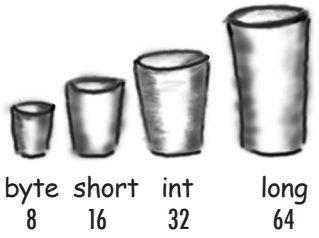
Primitives are like the cups they have at the coffee shop. If you've been to a Starbucks, you know what we're talking about here. They come in different sizes, and each has a name like "short," "tall," and, "I'd like a 'grande' mocha half-caff with extra whipped cream."

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.

You can't put a large value into a small cup.

Well, OK, you can, but you'll lose some. You'll get, as we say, *spillage*. The compiler tries to help prevent this if it can tell from your code that something's not going to fit in the container (variable/cup) you're using.

For example, you can't pour an int-full of stuff into a byte-sized container, as follows:

```
int x = 24;
byte b = x;
//won't work!!
```

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.

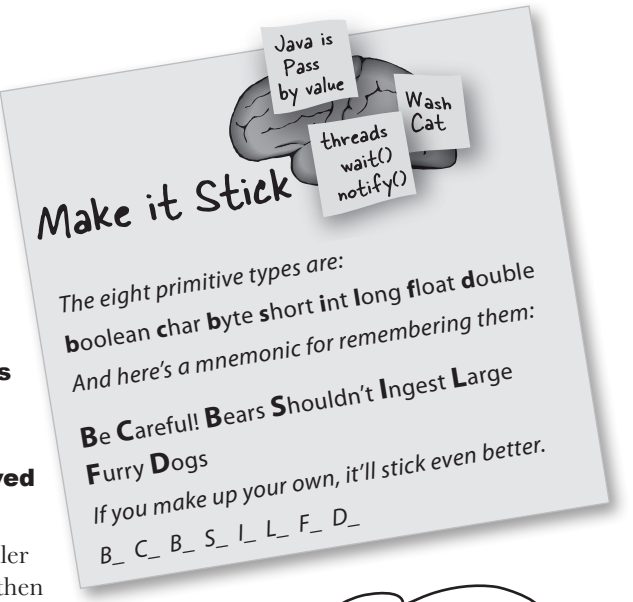
Back away from that keyword!

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.***

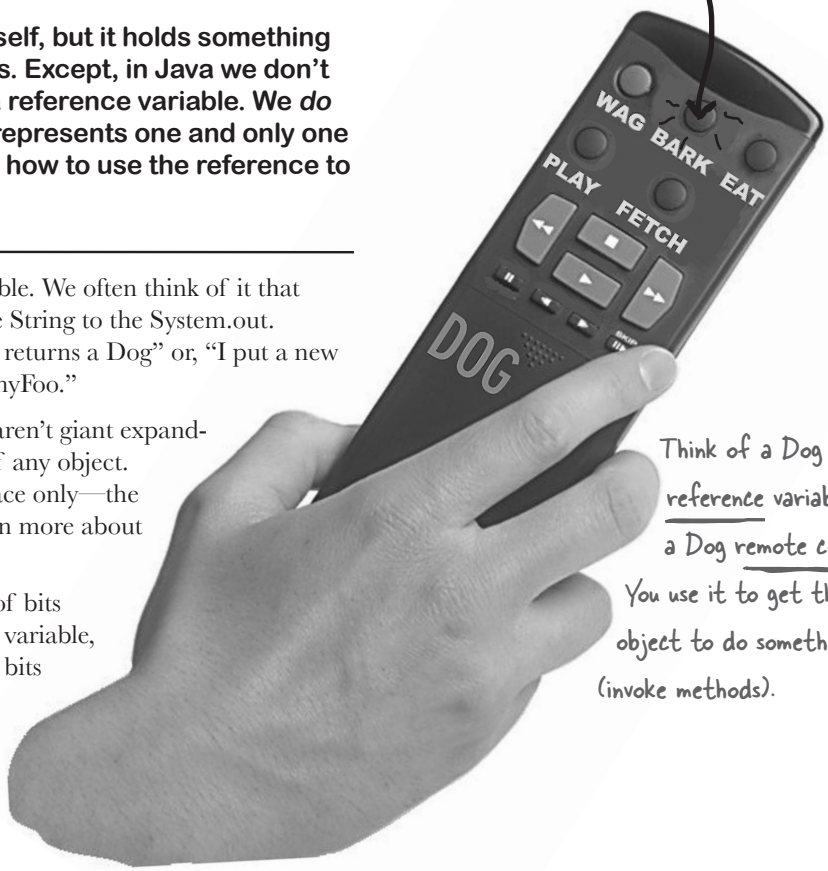
You use the dot operator (.) on a reference variable to say, "use the thing *before* the dot to get me the thing *after* the dot." For example:

```
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

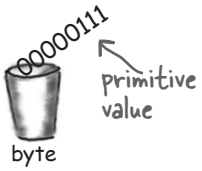
Something that goes in a cup.

Only this time, the value is a remote control.

Primitive Variable

```
byte x = 7;
```

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

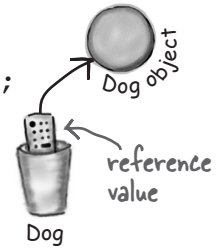


Reference Variable

```
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

```
1          3  2
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

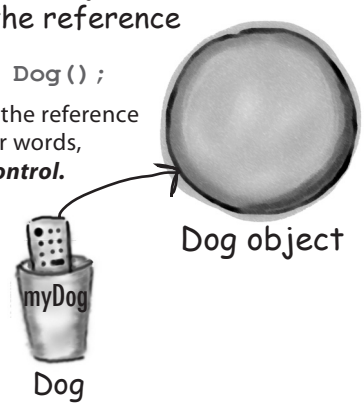
```
Dog myDog = new Dog();
```

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."

Java Exposed

This week's interview: Object Reference

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

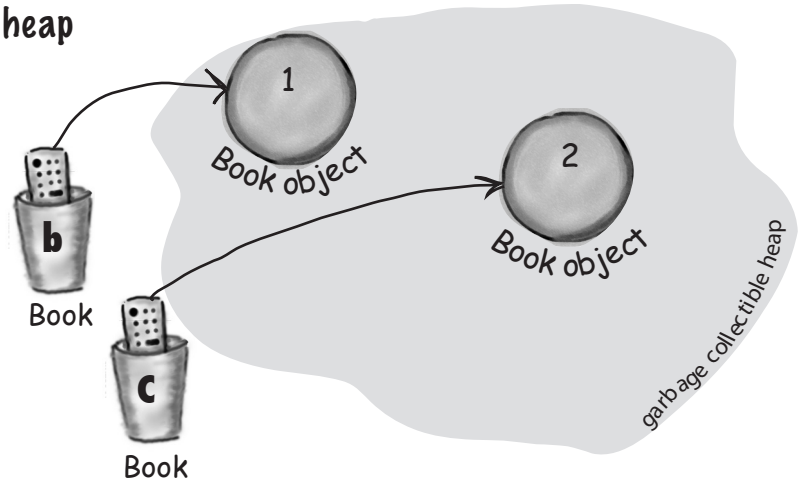
```
Book b = new Book();
Book c = new Book();
```

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;
```

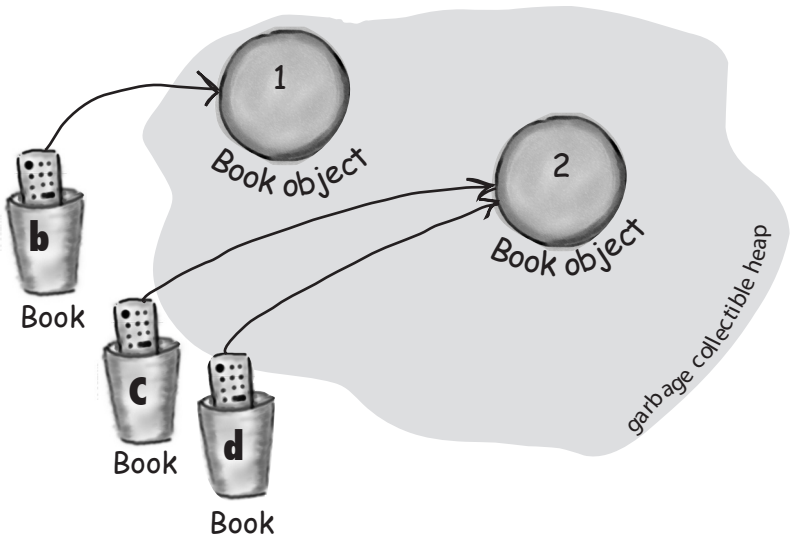
Declare a new Book reference variable. Rather than creating a new, third Book object, assign the value of variable ***c*** to variable ***d***. But what does this mean? It's like saying "Take the bits in ***c***, make a copy of them, and stick that copy into ***d***."

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;
```

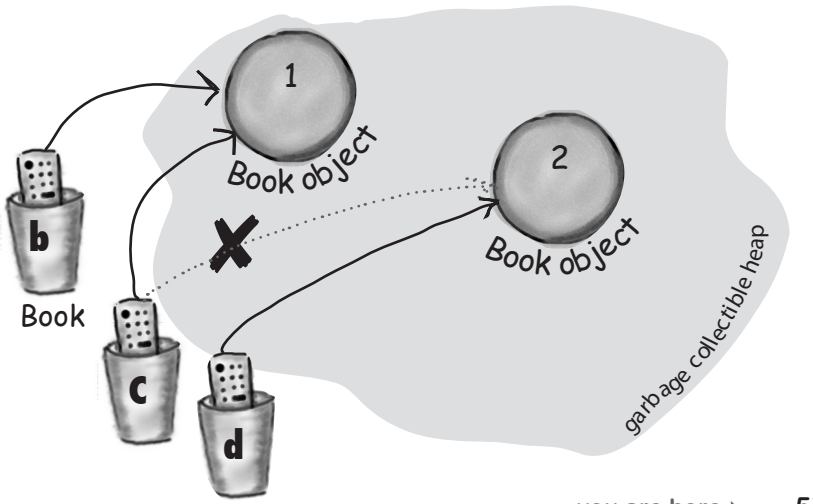
Assign the value of variable ***b*** to variable ***c***. By now you know what this means. The bits inside variable ***b*** are copied, and that new copy is stuffed into variable ***c***.

Both b and c refer to the same object.

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

References: 3

Objects: 2



Life and death on the heap

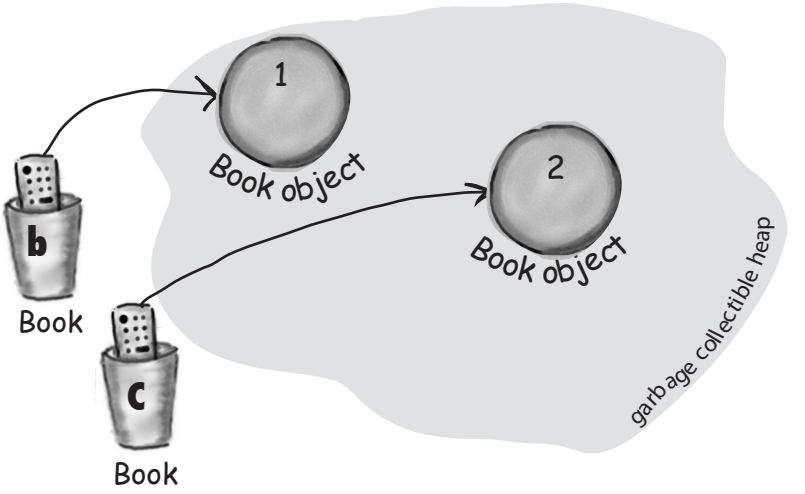
```
Book b = new Book();
Book c = new Book();
```

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.

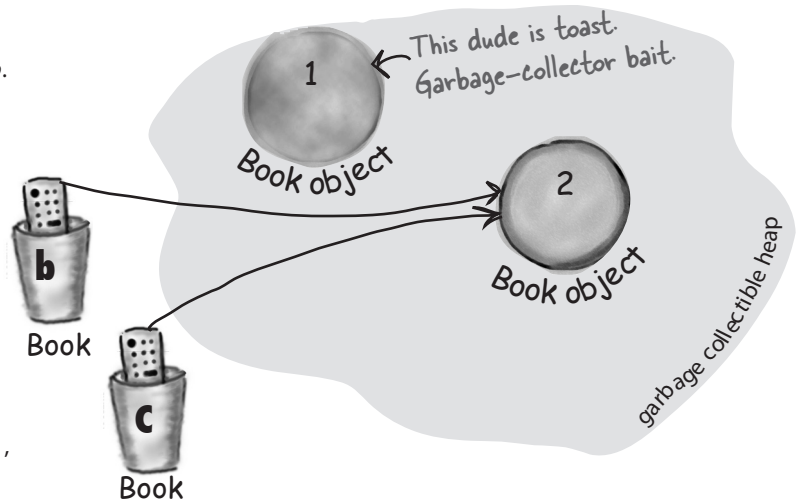
Both b and c refer to the same object. Object 1 is abandoned and eligible for Garbage Collection (GC).

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.

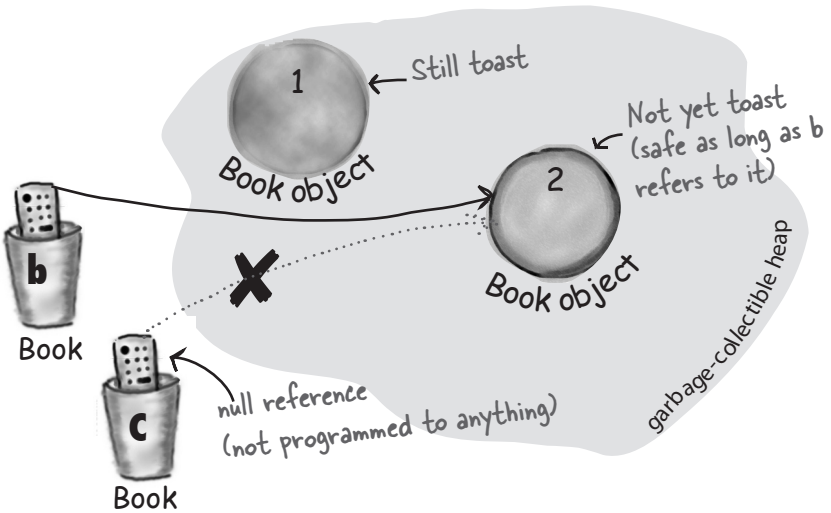
Object 2 still has an active reference (b), and as long as it does, the object is not eligible for GC.

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`

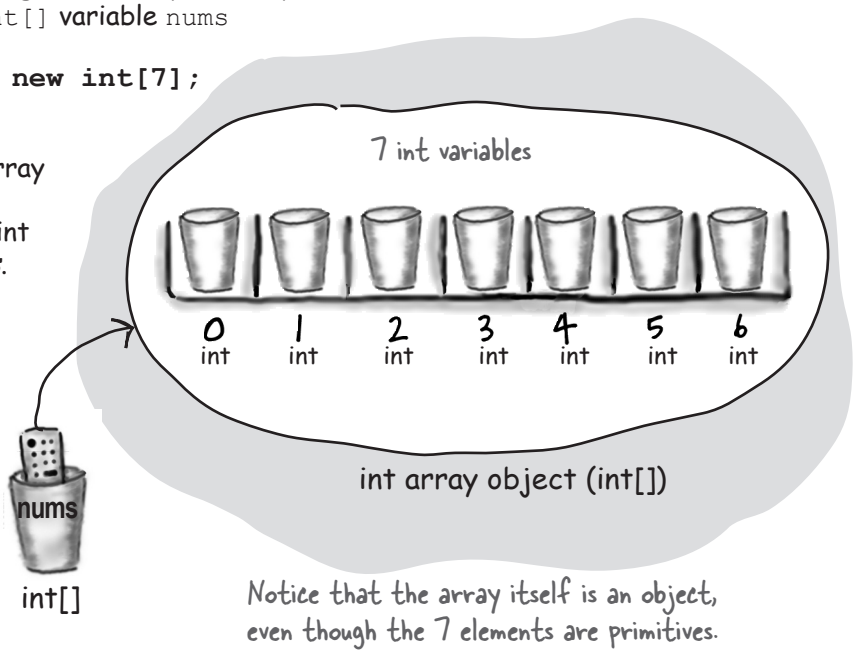
```
nums = new int[7];
```

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

Arrays are always objects, whether they're declared to hold primitives or object references.



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!

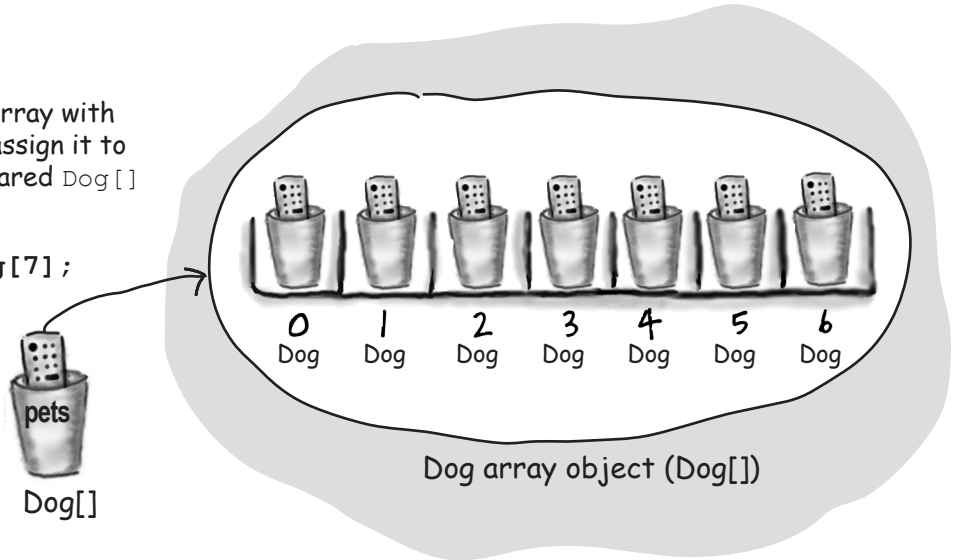
Make an array of Dogs

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];
```

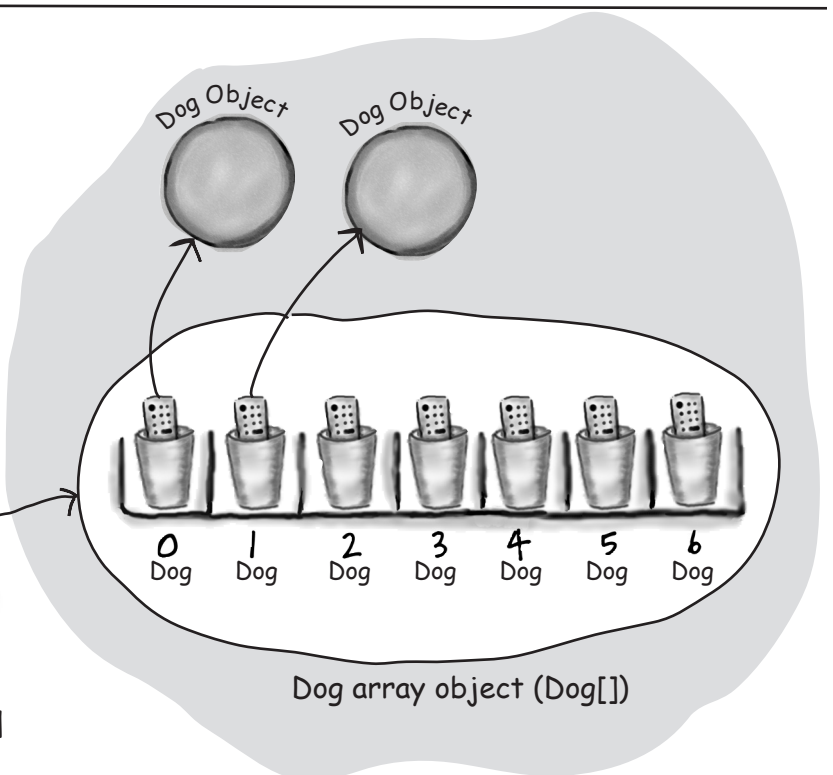


What's missing?

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

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
name
bark() eat() chaseCat()

Control your Dog

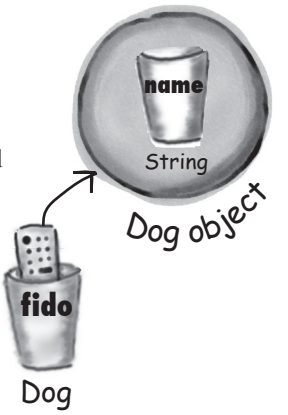
(with a reference variable)

```
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();
```

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.

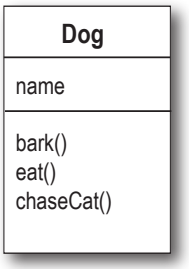
*Yes we know we're not demonstrating encapsulation here, but we're trying to keep it simple. For now. We'll do encapsulation in Chapter 4.

```
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";

    // Hmmmm... 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).

Arrays have a variable 'length' that gives you the number of elements in the array.

A Dog example



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.

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) {
```


Answers on page 68.

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 = _____________________
```

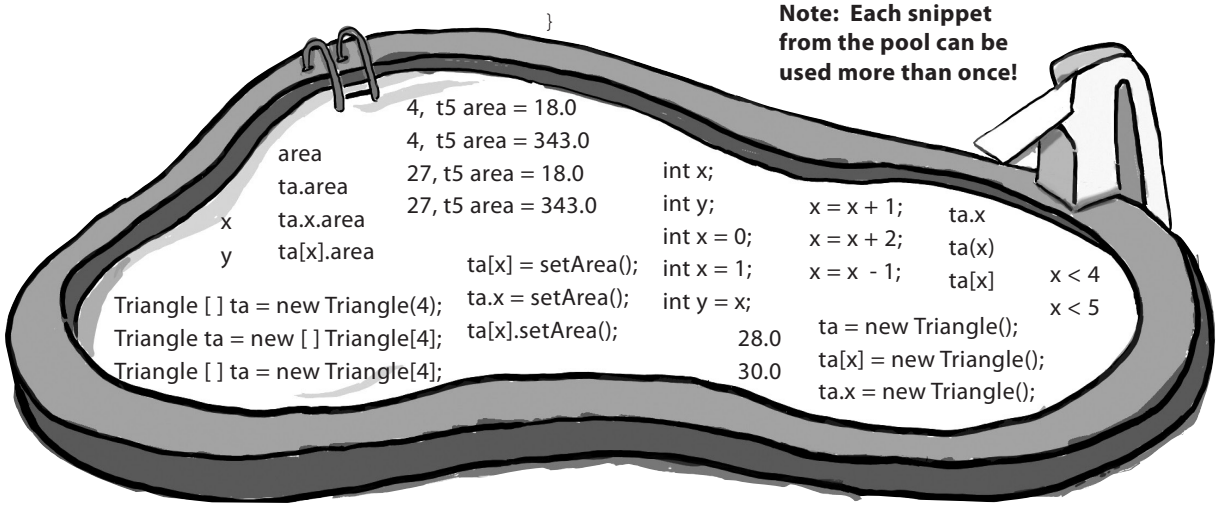
Bonus Question!

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

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

  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;
  }
}
```

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

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



Answers on page 69.

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).

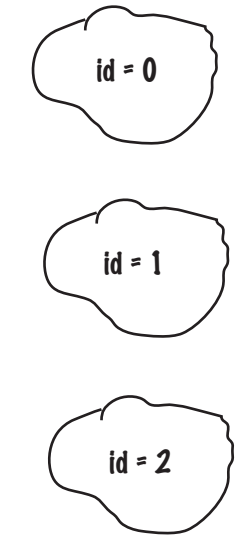
```
class HeapQuiz {
  int id = 0;

  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.

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."

Five-Minute Mystery

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

Sharpen your pencil (from page 52)

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; ✗

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)

```
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();
    myBooks[1] = new Books();
    myBooks[2] = new Books();
    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;
    }
  }
}
```

A

Remember: We have to actually make the Book objects!

```
class Hobbits {
  String name;

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

B

Remember: arrays start with element 0!

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
```

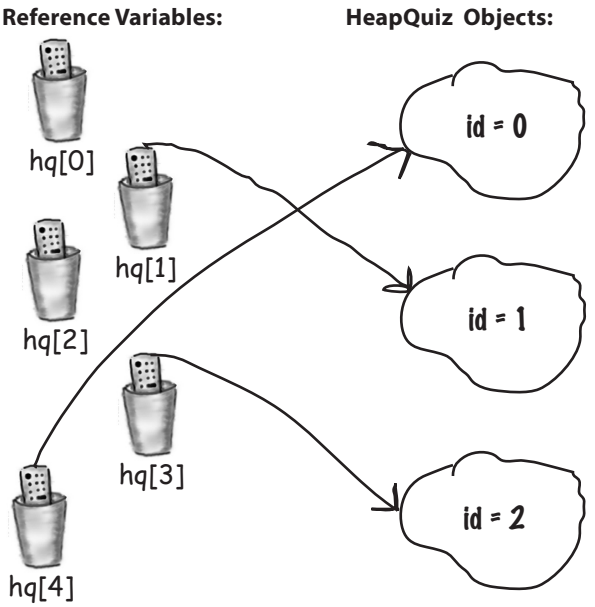
Five-Minute Mystery (from page 67)

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)

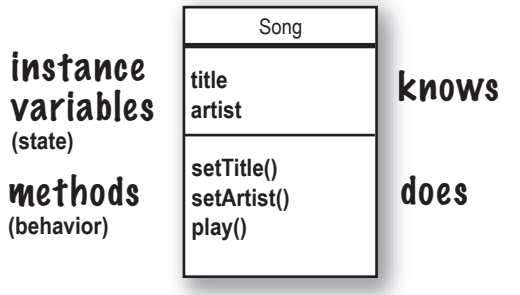


How Objects Behave

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.***

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?



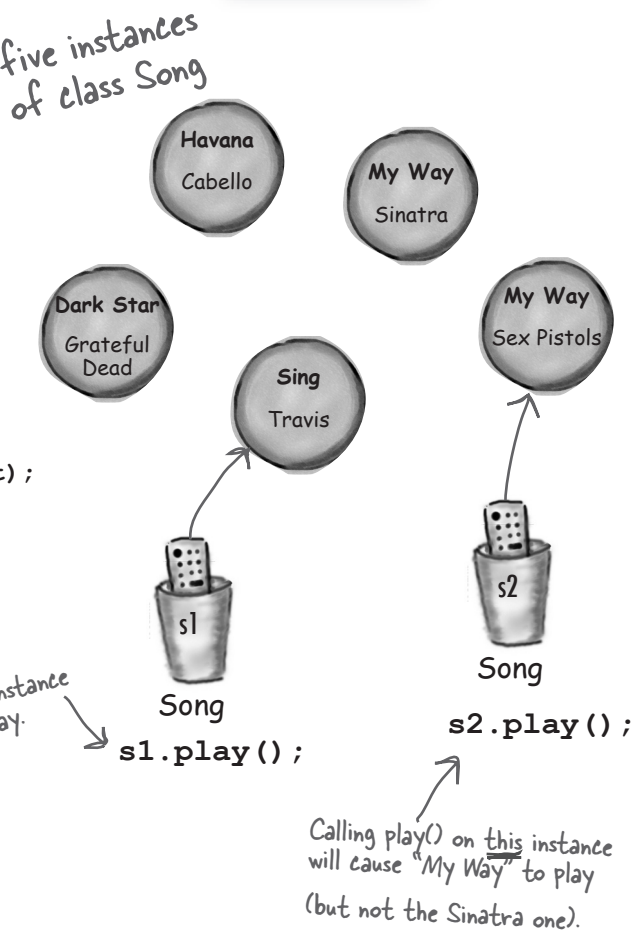
Can every object of that type have different method behavior?

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 the 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);
}
```



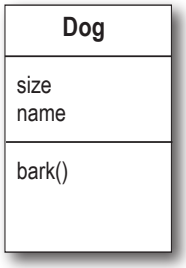

```
Song song1 = new Song();
song1.setArtist("Travis");
song1.setTitle("Sing");
Song song2 = new Song();
song2.setArtist("Sex Pistols");
song2.setTitle("My Way");
```

*Yes, another stunningly clear answer!

The size affects the bark

A small Dog's bark is different from a big Dog's bark.

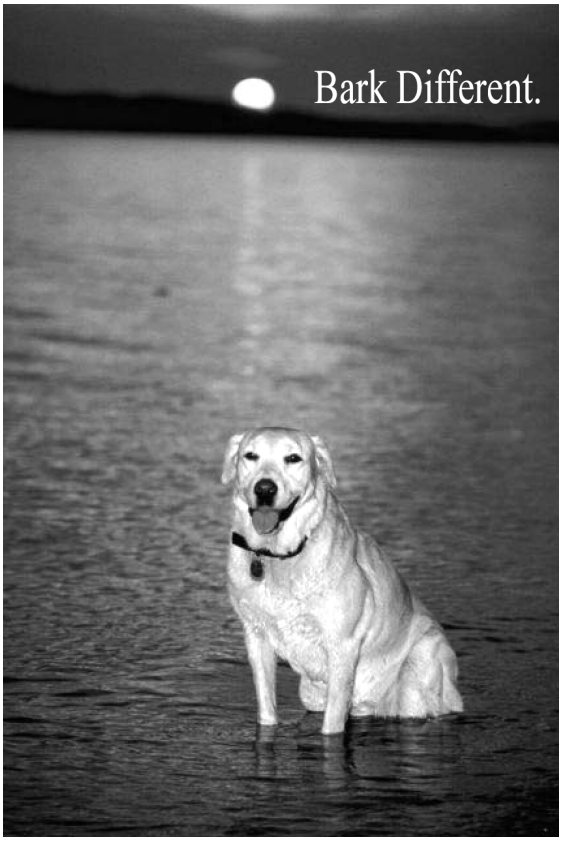
The Dog class has an instance variable *size* that the *bark()* method uses to decide what kind of bark sound to make.




```
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:

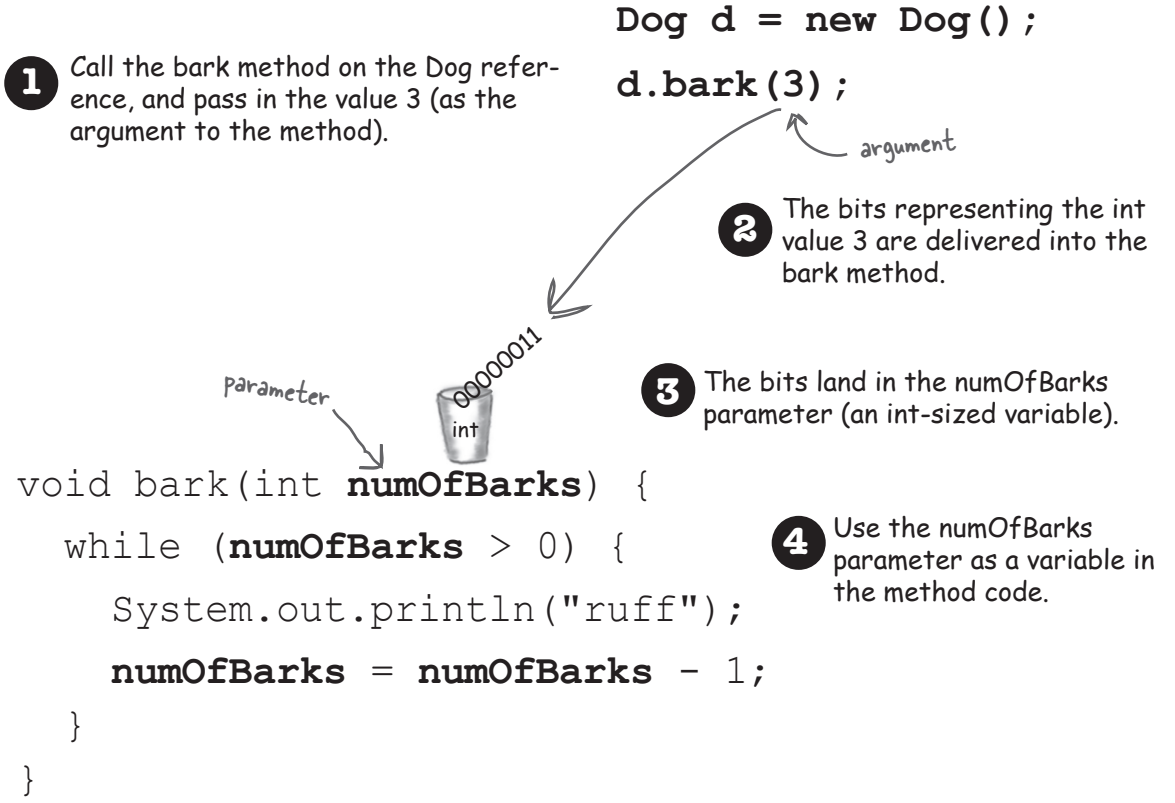
```
d.bark(3);
```

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, hairballs, 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() {
}
```

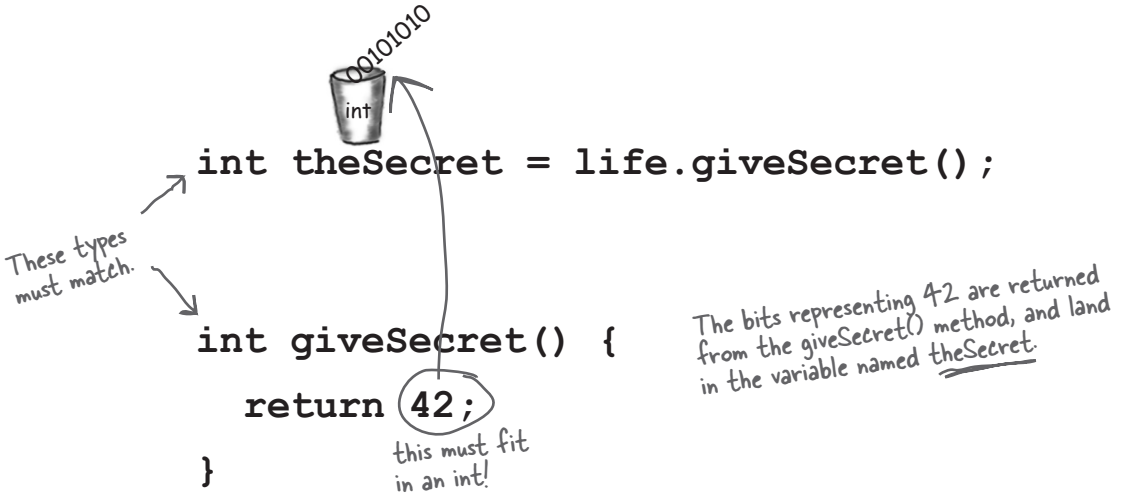
But we can declare a method to give a specific type of value back to the caller, such as:

```
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.

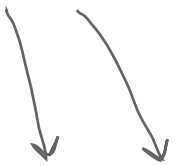


You can send more than one thing to a method

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

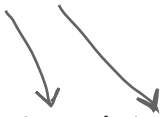
```
void go() {
  TestStuff t = new TestStuff();
  t.takeTwo(12, 34);
}

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



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.

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);
}

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



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.

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.

Java is pass-by-value.

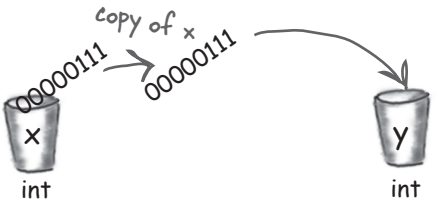
That means pass-by-copy.

```
int x = 7;
```

1 Declare an int variable and assign it the value '7'. The bit pattern for 7 goes into the variable named x.


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

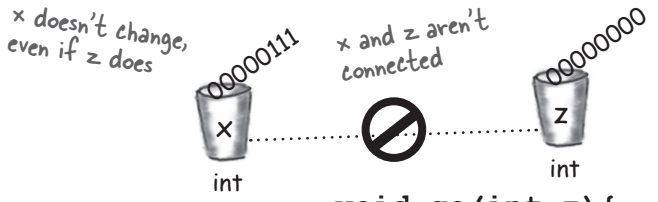
2 Declare a method with an int parameter named z.




```
foo.go(x);
```

```
void go(int 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.




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

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?

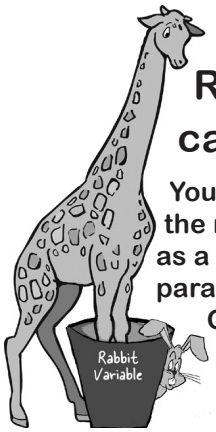
A: Sort of. A method can declare only one return value. BUT...if you want to return, say, three int values, then the declared return type can be an int *array*. Stuff those ints into the array, and pass it on back. It's a little more involved to return multiple values with different types; we'll be talking 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.

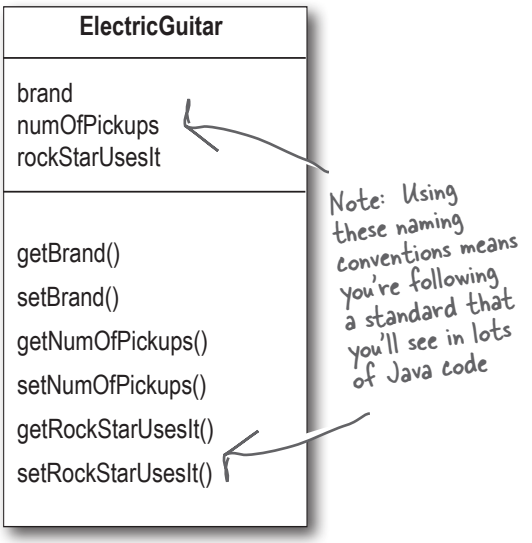
BULLET POINTS

- 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;
  }
}
```

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)

Java Exposed

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.

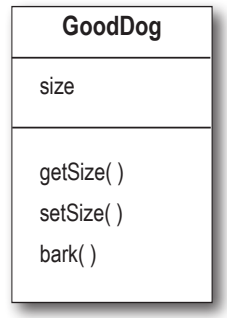
HeadFirst: I see what you mean. So how does encapsulation let you set boundaries?

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 miniscule 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;
  }

  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();
  }
}
```

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.

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:

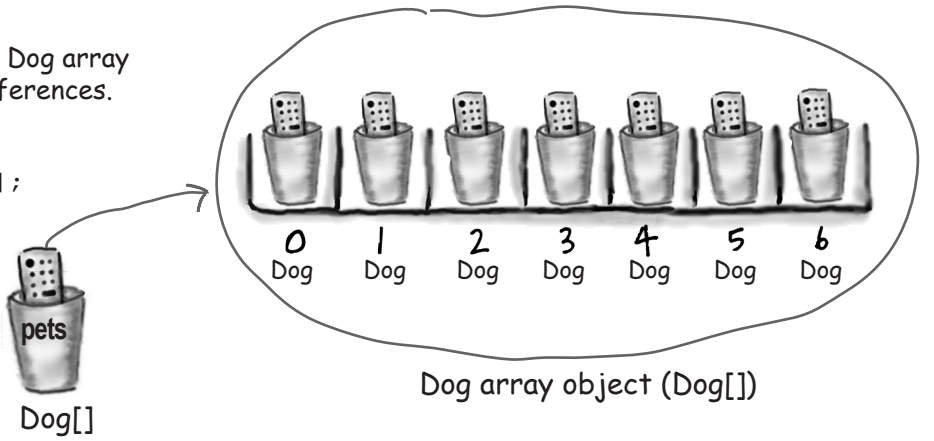
```
int x = 3 + one.getSize();
```

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];
```

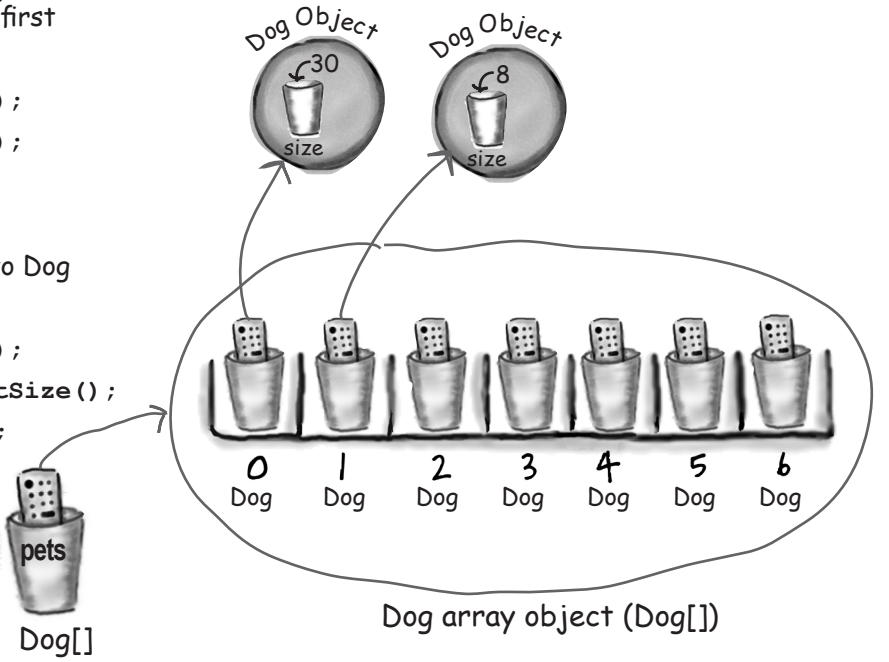


2 Create two new Dog objects, and assign them to the first two array elements.

```
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?

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

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

  public int getSize() {
    return size;
  }

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

Declare two instance variables, but don't assign a value

What will these return??

```
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());
  }
}
```

What do you think? Will this even compile?

```
File Edit Window Help CallVet
% java PoorDogTestDrive
Dog size is 0
Dog name is null
```

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() {        <-- LOCAL variable
    int total = a + b;
    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.

Use == to compare two primitives or to see if two references refer to the *same* object.

Use the equals() method to see if two *different* objects are equal.

(E.g., two different String objects that both contain the characters "Fred")

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.

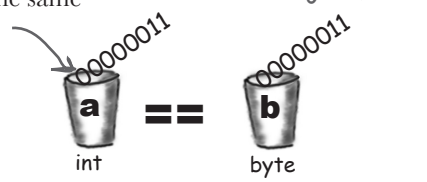
To compare two primitives, use the == operator

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 }
```

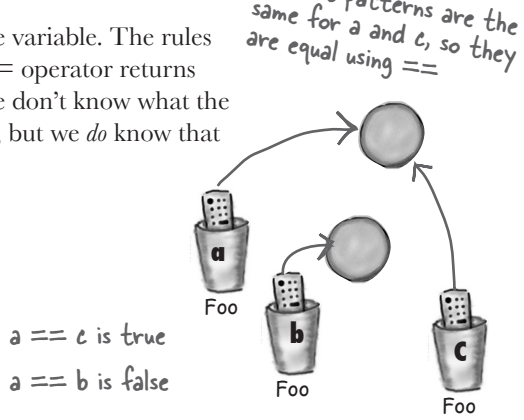
(There are more zeros on the left side of the int, but we don't care about that here)



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?

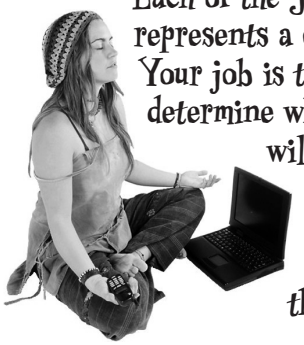
Put a checkmark next to the ones that are legal. (Some statements are there to assign values used in the method calls.)


```
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

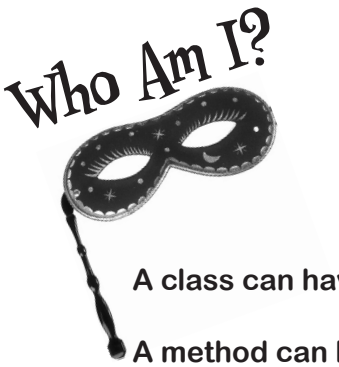
```
class Clock {
  String time;

  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.

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 attendee, then write down all for whom that sentence applies. Fill in the blanks next to the sentence with the names of one or more attendees.

Tonight's attendees:

instance variable, argument, return, getter, setter, encapsulation, public, private, pass by value, method



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.

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.

```
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;
  }
}
```

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
```

Answers on page 94.

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.

→ Answers on page 94.

Output

```
File  Edit  Window  Help  BellyFlop
%java Puzzle4
result 543345
```

```
public class Puzzle4 {
  public static void main(String [] args) {

    _____________________________________
    int number = 1;
    int i = 0;
    while (i < 6) {

      _____________________________
      _____________________________
      number = number * 10;

      ___________________
    }

    int result = 0;
    i = 6;
    while (i > 0) {

      ___________________
      result = result + ____________________
    }
    System.out.println("result " + result);
  }
}

class ___________ {
  int intValue;
  _________  _______ doStuff(int __________) {
    if (intValue > 100) {
      return ______________________________
    } else {
      return ______________________________
    }
  }
}
```

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

doStuff(i);
values.doStuff(i);
values[i].doStuff(factor);
values[i].doStuff(i);
intValue = i;
values.intValue = i;
values[i].intValue = i;
values[i].intValue = number;
intValue
factor
public
private
intValue + factor;
intValue * (2 + factor);
intValue * (5 - factor);
intValue * factor;
Puzzle4
Value
Value( )
int
short
Puzzle4 [ ] values = new Puzzle4[6];
Value [ ] values = new Value[6];
Value [ ] values = new Puzzle4[6];
i = i + 1;
i = i - 1;
values [i] = new Value(i);
values [ ] = new Value( );
values [i] = new Value( );
values = new Value( );

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);
```

BE the Compiler (from page 88)

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.

B

```
class Clock {
  String time;

  void setTime(String t) {
    time = t;
  }

  String 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);
  }
}
```

Note: 'Getter' methods have a return type by definition.

Who Am I? (from page 89)

A class can have any number of these.	instance variables, getter, setter, method
A method can have only one of these.	return
This can be implicitly promoted.	return, argument
I prefer my instance variables private.	encapsulation
It really means "make a copy."	pass by value
Only setters should update these.	instance variables
A method can have many of these.	argument
I return something by definition.	getter
I shouldn't be used with instance variables	public
I can have many arguments.	method
By definition, I take one argument.	setter
These help create encapsulation.	getter, setter, public, private
I always fly solo.	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

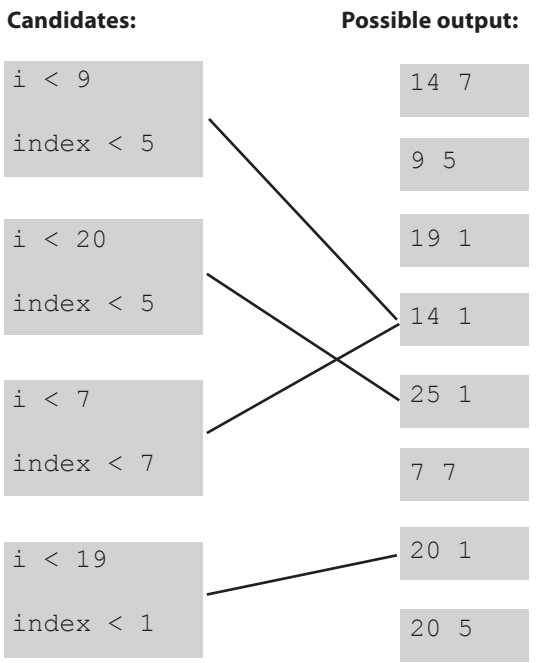
```
File  Edit  Window  Help  BellyFlop
%java Puzzle4
result 543345
```

Five-Minute Mystery (from page 92)

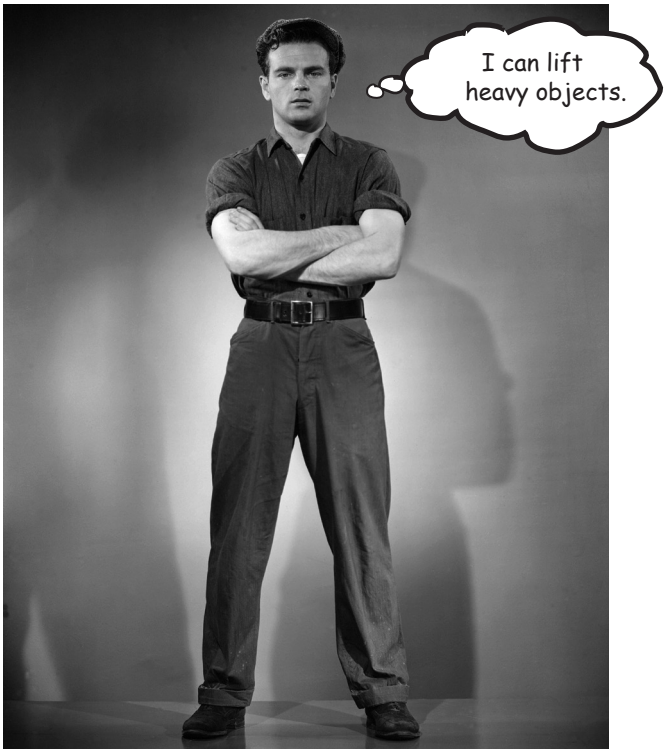
What did Jai suspect?

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)



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"

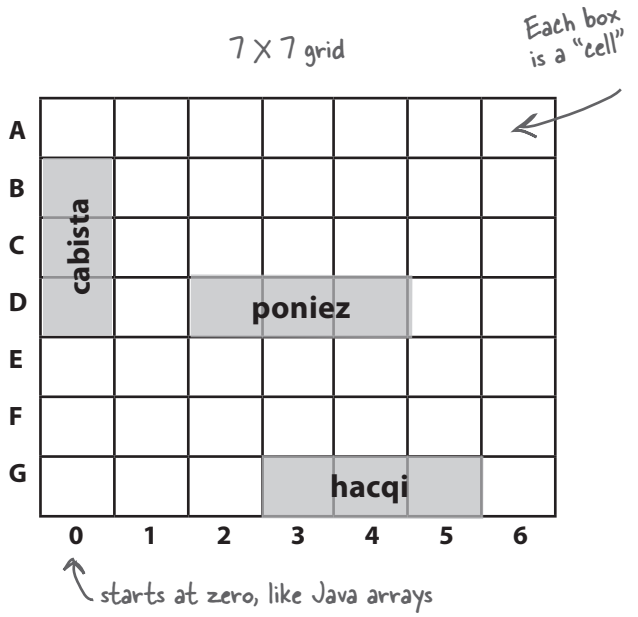
It's you against the computer, but unlike the real Battleship game, in this one you don't place any ships of your own. Instead, your job is to sink the computer's ships in the fewest number of guesses.

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.

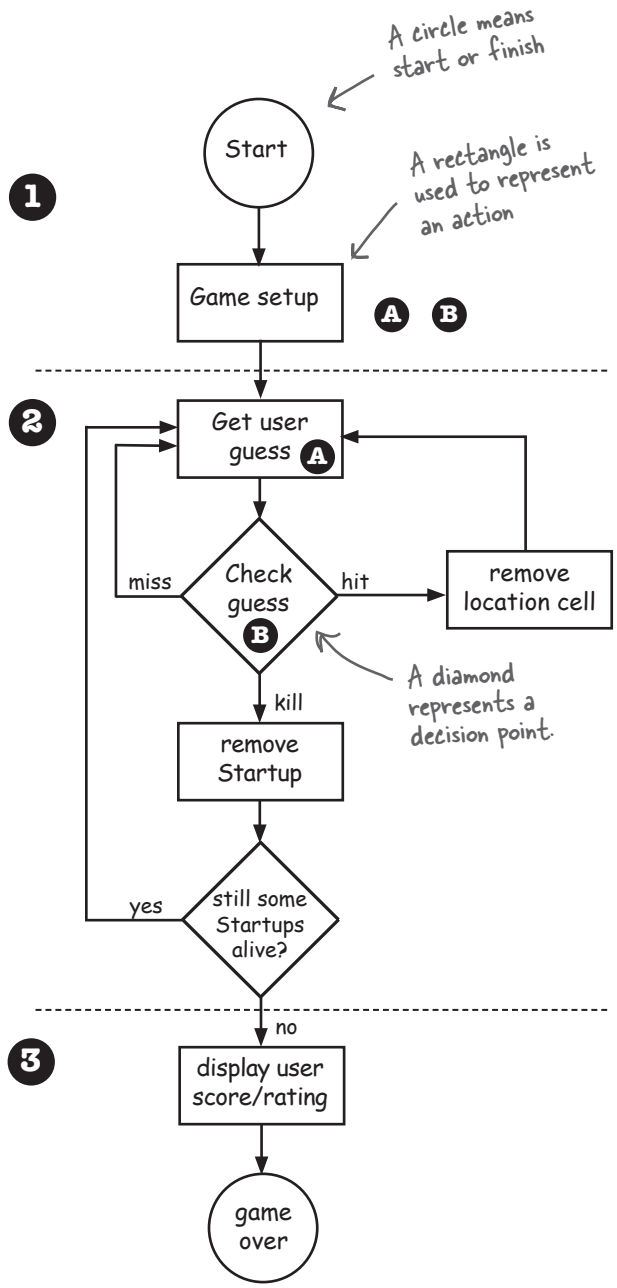
Repeat the following until there are no more Startups:

- **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***.



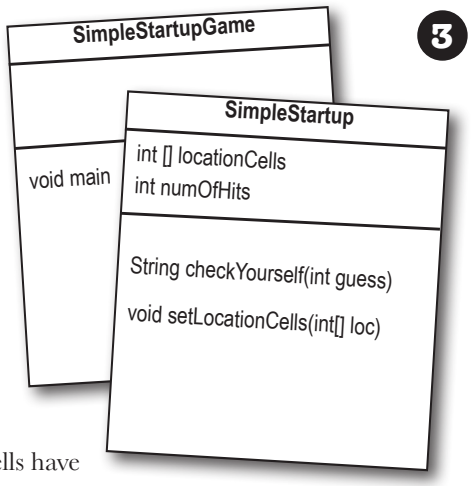
Whoa. A real flow chart.

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.

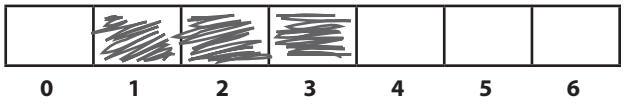


In this simple version, the game class has no instance variables, and all the game code is in the main() method. In other words, when the program is launched and main() begins to run, it will make the one and only Startup instance, pick a location for it (three consecutive cells on the single virtual 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:

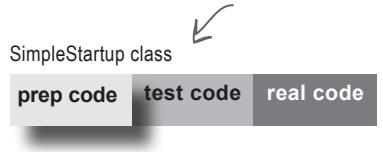
- ☐ Figure out what the class is supposed to *do*.
- ☐ List the **instance variables and methods**.
- ☐ Write **prep code** for the methods. (You'll see this in just a moment.)
- ☐ Write **test code** for the methods.
- ☐ **Implement** the class.
- ☐ **Test** the methods.
- ☐ **Debug** and **reimplement** 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:

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.



prep code

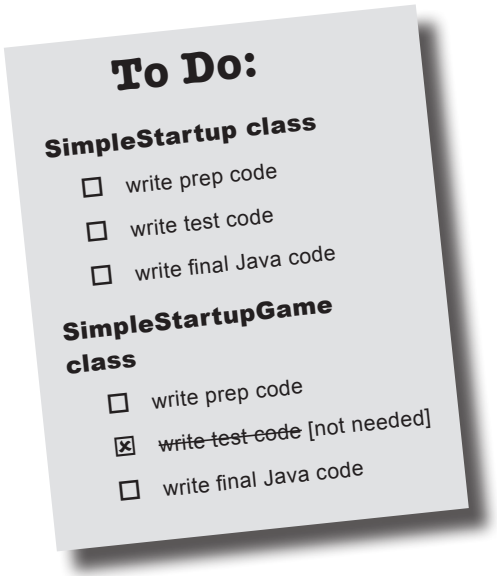
A form of pseudocode, to help you focus on the logic without stressing about syntax.

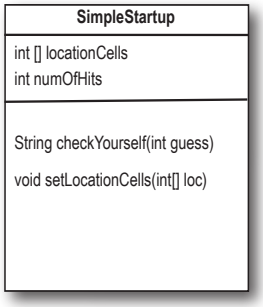
test code

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.





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
```

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").

prep code | test code | real code

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();

    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);

  }

}
```

Instantiate a SimpleStartup object.

Make an int array for the location of the Startup (3 consecutive ints out of a possible 7).

Invoke the setter method on the Startup.

Make a fake user guess.

Invoke the checkYourself() method on the Startup object, and pass it the fake guess.

If the fake guess (2) gives back a "hit", it's working.

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";
  (1)
  for (int cell : locationCells) {
    if (guess == cell) {
      result = "hit";
      (2)
      numOfHits++;
      (3)
      break;
    } // end if
  } // end for
  if (numOfHits == locationCells.length) {
    result = "kill";
  } // end if
  System.out.println(result);
  return result;
} // end method
```

Make a variable to hold the result we'll return. put "miss" in as the default (i.e., we assume a "miss")

Repeat with each cell in the locationCells array (each cell location of the object)

Compare the user guess to this element (cell) in the array

We got a hit!

Get out of the loop, no need to test the other cells

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"

Display the result for the user ("miss", unless it was changed to "hit" or "kill")

Return the result back to the calling method

prep code | test code | real code

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

```
for (int cell : locationCells) { }
```

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..."

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!

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 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 if
    } // end for
    if (numOfHits ==
        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 checkYouself() method. If the code is working correctly, we should see the result print out:

```
% java SimpleStartupTestDrive
hit
passed
```

prep code | test code | real code

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 *somehow* 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)

INVOKE 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

INVOKE the *checkYourself()* method on the SimpleStartup instance

INCREMENT *numOfGuesses* variable

// CHECK for Startup death

IF result is "kill"

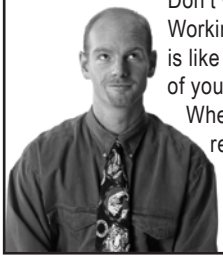
SET *isAlive* to false (which means we won't enter the loop again)

PRINT the number of user guesses

END IF

END WHILE

END METHOD



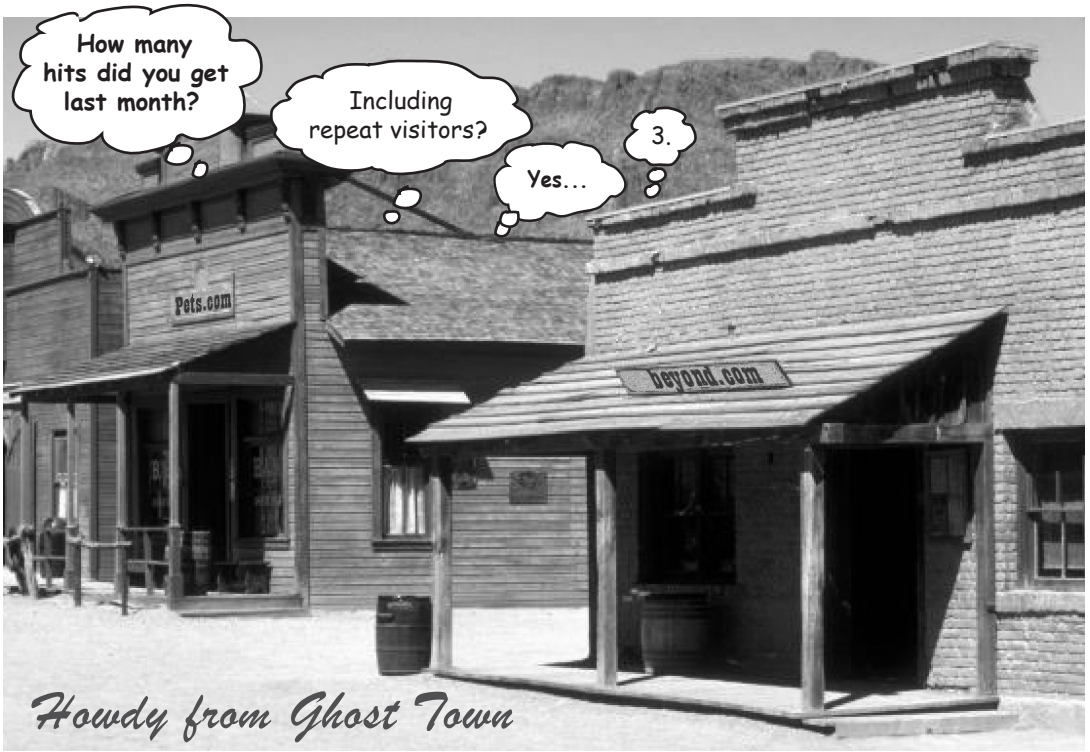
metacognitive tip

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).



prep code | test code | real code

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;

  GameHelper helper = new GameHelper();


  SimpleStartup theStartup = new SimpleStartup();

  int randomNum = (int) (Math.random() * 5);


  int[] locations = {randomNum, randomNum + 1, randomNum + 2};

  theStartup.setLocationCells(locations);

  boolean isAlive = true;


  while (isAlive) {

    int guess = helper.getUserInput("enter a number");

    String result = theStartup.checkYourself(guess);

    numOfGuesses++;

    if (result.equals("kill")) {

      isAlive = false;

      System.out.println("You took " + numOfGuesses + " guesses");

    } // close if

  } // close while
```

DECLARE a variable to hold user guess count, and set it to 0

Make a variable to track how many guesses the user makes.

This is a special class we wrote that has the method for getting user input. For now, pretend it's part of Java.

MAKE a SimpleStartup object

Make the Startup object.

COMPUTE a random number between 0 and 4

(1)

Make a random number for the first cell, and use it to make the cell locations array.

MAKE an int array with the 3 cell locations, and

INVOKE setLocationCells on the Startup object

Give the Startup its locations (the array).

DECLARE a boolean isAlive

Make a boolean variable to track whether the game is still alive, to use in the while loop test.

WHILE the Startup is still alive

repeat while game is still alive.

GET user input

(2)

Get user guess.

// CHECK it

INVOKE checkYourself() on Startup

Ask the Startup to check the guess; save the returned result.

INCREMENT numOfGuesses

Increment guess count by one.

IF result is "kill"

Was it a "kill"? if so, set isAlive to false (so we won't re-enter the loop) and print user guess count.

SET isAlive to false

PRINT the number of user guesses

prep code | test code | real code

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

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).

```
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.

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.

Ready-Bake Code

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.


```
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();
  }
}
```

*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*.