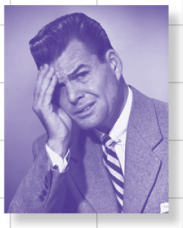
A Brain-Friendly Guide

Head First Java™

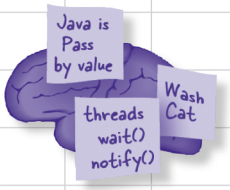
2nd Edition
Covers Java 5.0

Learn how threads can change your life



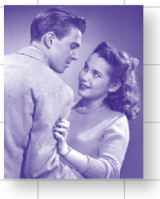
Avoid embarassing OO mistakes

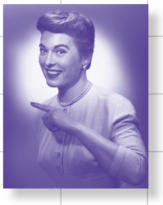
Bend your mind around 42 Java puzzles



Make Java concepts stick to your brain

Fool around in the Java Library





Make attractive and useful GUIs

O'REILLY®

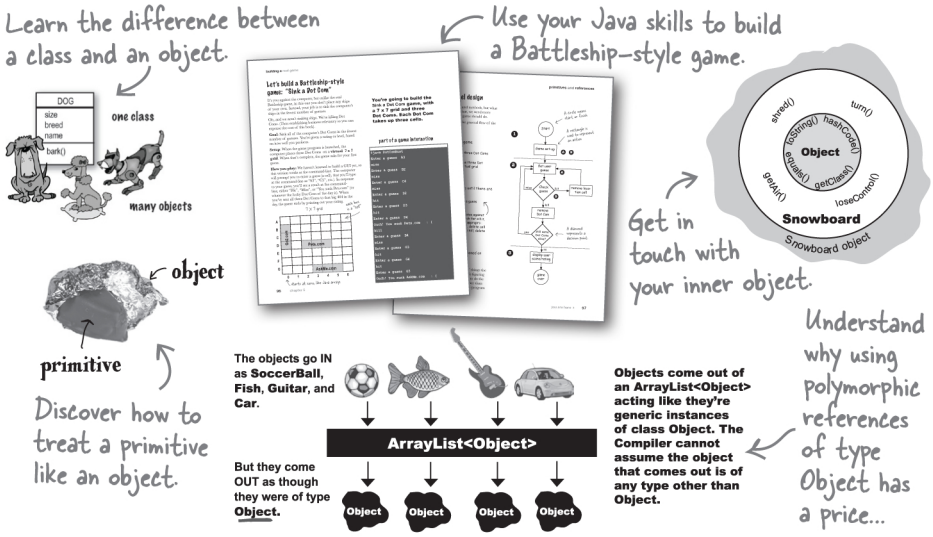
Kathy Sierra & Bert Bates

Head First Java™

Java

What will you learn from this book?

Head First Java is a complete learning experience in Java and object-oriented programming. This book helps you learn the Java language with a unique method that goes beyond syntax and how-to manuals and helps you understand how to be a great programmer. You'll learn language fundamentals, generics, threading, networking, and distributed programming, and you'll even build a "sink the dot com" game and networked drum machine chat client along the way.



What's so special about this book?

We think your time is too valuable to waste struggling with new concepts. Using the latest research in cognitive science and learning theory to craft a multi-sensory learning experience, *Head First Java* uses a visually rich format designed for the way your brain works, not a text-heavy approach that puts you to sleep.

"... 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, Sun Microsystems, Chairman, President, and CEO

"*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*

"It's fast, irreverent, fun, and engaging. Be careful—you might actually learn something!"

—Ken Arnold, coauthor (with James Gosling, creator of Java), The Java Programming Language

US $44.95 CAN $62.95
ISBN: 978-0-596-00920-5
9 780596 009205 54495

O'REILLY®
www.oreilly.com
www.headfirstlabs.com

What they're saying about *Head First*

Amazon named Head First Java a Top Ten Editor's Choice for Computer Books of 2003 (first edition)

Software Development Magazine named Head First Java a finalist for the 14th Annual Jolt Cola/Product Excellence Awards

"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
Co-author (with James Gosling, creator of Java), "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, 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

Praise for *Head First* Java

"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. I hope that there are more authors who are willing to break out of the same old mold of 'traditional' writing styles. Learning computer languages should be fun, not onerous."

— 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 & CEO, Geek Cruises (www.GeekCruises.com)

"If you're relatively new to programming and you are interested in Java, here's your book...Covering everything from objects to creating graphical user interfaces (GUI), exception (error) handling to networking (sockets) and multithreading, even packaging up your pile of classes into one installation file, this book is quite complete...If you like the style...I'm certain you'll love the book and, like me, hope that the Head First series will expand to many other subjects!"

— LinuxQuestions.org

"I was ADDICTED to the book's short stories, annotated code, mock interviews, and brain exercises."

— Michael Yuan, author, 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 a 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 view conference, learning from – and laughing with – peers... If you want to UNDERSTAND Java, go buy this book."

— Andrew Pollack, www.thenorth.com

"If anyone in the world is familiar with the concept of 'Head First,' it would be me. This book is so good, I'd marry it on TV!"

— Rick Rockwell, Comedian
The original FOX Television "Who Wants to Marry a Millionaire" groom

"This stuff is so fricking good it makes me wanna WEEP! I'm stunned."

— Floyd Jones, Senior Technical Writer/Poolboy, BEA

"A few days ago I received my copy of Head First Java by Kathy Sierra and Bert Bates. I'm only part way through the book, but what's amazed me is that even in my sleep-deprived state that first evening, I found myself thinking, 'OK, just one more page, then I'll go to bed.' "

— Joe Litton

Praise for other *Head First* books co-authored by Kathy and Bert

Amazon named Head First Servlets a Top Ten Editor's Choice for Computer Books of 2004 (first edition)

Software Development Magazine named Head First Servlets and Head First Design Patterns finalists for the 15th Annual Product Excellence Awards

"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

"FINALLY - a Java book written the way I would'a wrote it if I were me.
Seriously though - this book absolutely blows away every other software book I've ever read...
A good book is very difficult to write... you have to take a lot of time to make things unfold in a natural, "reader oriented" sequence. It's a lot of work. Most authors clearly aren't up to the challenge.
Congratulations to the Head First EJB team for a first class job!

— Wally Flint

"I could not have imagined a person smiling while studying an IT book! Using Head First EJB materials, I got a great score (91%) and set a world record as the youngest SCBCD, 14 years."

— Afsah Shafquat (world's youngest SCBCD)

"This Head First Servlets book is as good as the Head First EJB book, which made me laugh AND gave me 97% on the exam!"

— Jef Cumps, J2EE consultant, Cronos

Other related books from O'Reilly

Ant: The Definitive Guide
Better, Faster, Lighter Java™
Enterprise JavaBeans™ 3.0
Hibernate: A Developer's Notebook
Java™ 1.5 Tiger: A Developer's Notebook
Java™ Cookbook
Java™ in a Nutshell
Java™ Network Programming
Java™ Servlet & JSP Cookbook
Java™ Swing
JavaServer™ Faces
JavaServer Pages™
Programming Jakarta Struts
Tomcat: The Definitive Guide

Other books in O'Reilly's Head First series

Head First Java™
Head First Object-Oriented Analysis and Design (OOA&D)
Head Rush Ajax
Head First HTML with CSS and XHTML
Head First Design Patterns
Head First EJB™
Head First PMP
Head First SQL
Head First Software Development
Head First C#
Head First JavaScript
Head First Programming (2008)
Head First Ajax (2008)
Head First Physics (2008)
Head First Statistics (2008)
Head First Ruby on Rails (2008)
Head First PHP & MySQL (2008)

Head First Java™

Second Edition

Kathy Sierra
Bert Bates

O'REILLY®
Beijing • Boston • Farnham • Sebastopol • Tokyo

Head First Java™

Second Edition

by Kathy Sierra and Bert Bates

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

Editor: Mike Loukides

Cover Designer: Edie Freedman

Interior Designers: Kathy Sierra and Bert Bates

Printing History:

May 2003: First Edition.

February 2005: Second Edition.

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

ISBN: 978-0-596-00920-5

[M] [2016-02-26]

To our brains, for always being there

(despite shaky evidence)

Creators of the Head First series

Kathy has been interested in learning theory since her days as a game designer (she wrote games for Virgin, MGM, and Amblin'). She developed much of the Head First format while teaching New Media Authoring for UCLA Extension's Entertainment Studies program. More recently, she's been a master trainer for Sun Microsystems, teaching Sun's Java instructors how to teach the latest Java technologies, and a lead developer of several of Sun's Java programmer and developer certification exams. Together with Bert Bates, she has been actively using the concepts in Head First Java to teach hundreds of trainers, developers and even non-programmers. She is also the founder of one of the largest Java community websites in the world, javaranch.com, and the Creating Passionate Users blog.

Along with this book, Kathy co-authored Head First Servlets, Head First EJB, and Head First Design Patterns.

In her spare time she enjoys her new Icelandic horse, skiing, running, and the speed of light.

kathy@wickedlysmart.com

Bert is a software developer and architect, but a decade-long stint in artificial intelligence drove his interest in learning theory and technology-based training. He's been teaching programming to clients ever since. Recently, he's been a member of the development team for several of Sun's Java Certification exams.

He spent the first decade of his software career travelling the world to help broadcast clients like Radio New Zealand, the Weather Channel, and the Arts & Entertainment Network (A & E). One of his all-time favorite projects was building a full rail system simulation for Union Pacific Railroad.

Bert is a hopelessly addicted Go player, and has been working on a Go program for way too long. He's a fair guitar player, now trying his hand at banjo, and likes to spend time skiing, running, and trying to train (or learn from) his Icelandic horse Andi.

Bert co-authored the same books as Kathy, and is hard at work on the next batch of books (check the blog for updates).

You can sometimes catch him on the IGS Go server (under the login *jackStraw*).

terrapin@wickedlysmart.com

Although Kathy and Bert try to answer as much email as they can, the volume of mail and their travel schedule makes that difficult. The best (quickest) way to get technical help with the book is at the *very* active Java beginners forum at javaranch.com.

Table of Contents (summary)

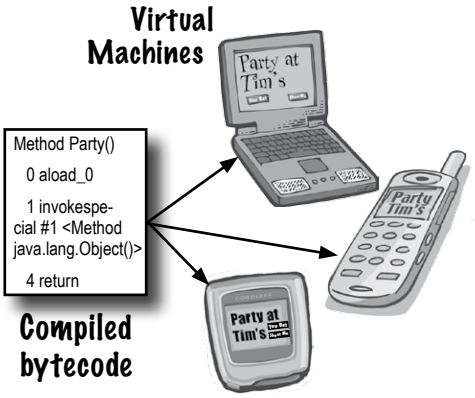
Table of Contents (the full version)

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?

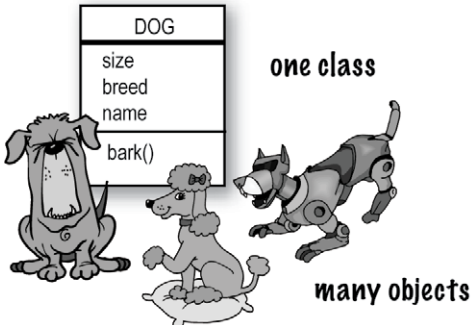
1 Breaking the Surface

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

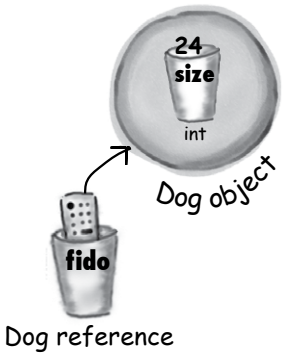
I was told there would be objects. In Chapter 1, we put all of our code in the main() method. That's not exactly object-oriented. So now we've got to leave that procedural world behind and start making some objects of our own. We'll look at what makes object-oriented (OO) development in Java so much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

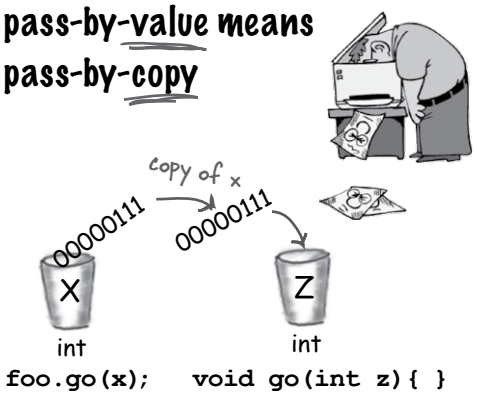
There's gotta be more to life than integers, Strings, and arrays. What if you have a PetOwner object with a Dog instance variable? Or a Car with an Engine? In this chapter we'll unwrap the mysteries of Java types and look at what you can *declare* as a variable, what you can *put* in a variable, and 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 Dot Com (similar to Battleship).

6 Using the Java Library

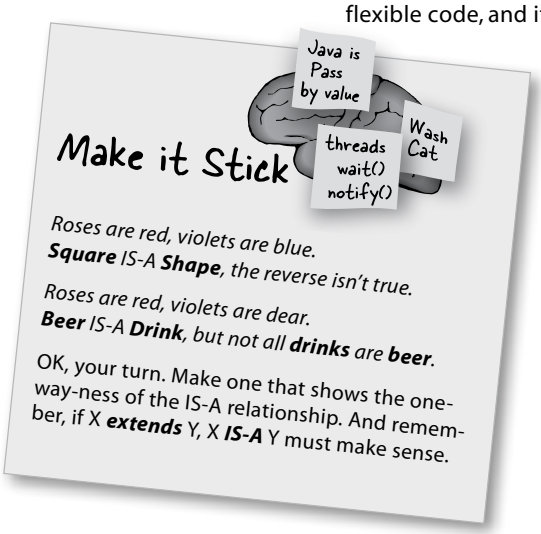
Java ships with hundreds of pre-built classes. You don't have to reinvent the wheel if you know how to find what you need from the Java library, commonly known as the **Java API**. *You've got better things to do*. If you're going to write code, you might as well write *only* the parts that are custom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

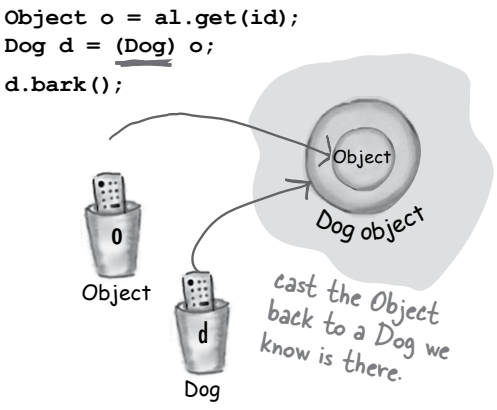
7 Better Living in Objectville

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.



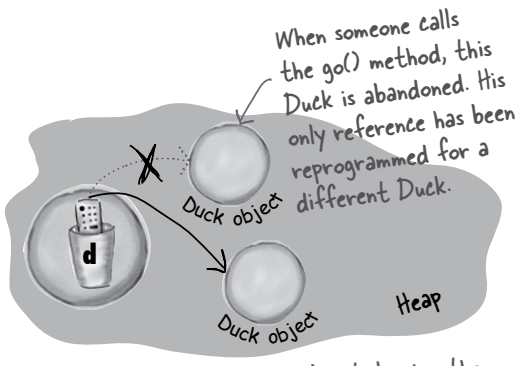
8 Serious Polymorphism

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..

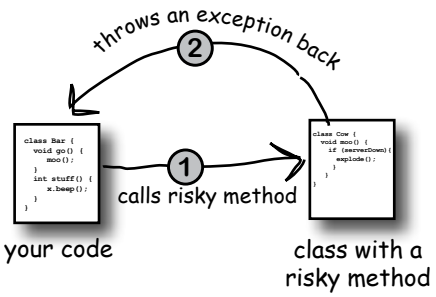
10 Numbers Matter

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 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.

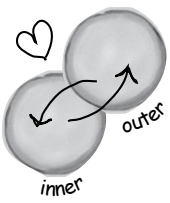


12 A Very Graphic Story

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.



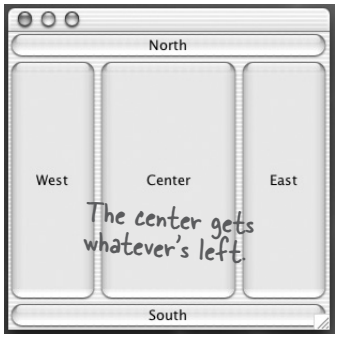
These two objects on the heap have a special bond. The inner can use the outer's variables (and vice-versa).

13 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.

Components in the east and west get their preferred width.

Things in the north and south get their preferred height.



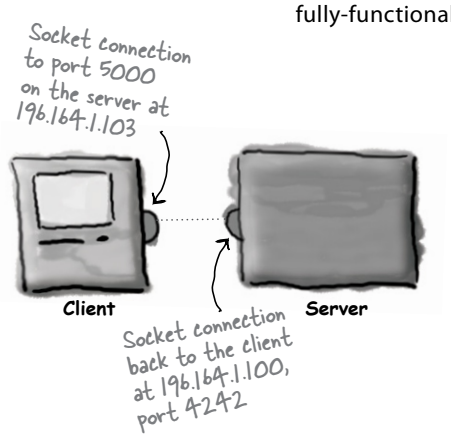
Saving Objects

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

Any questions?

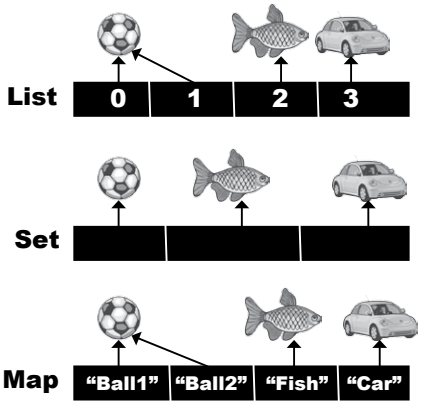
15 Make a Connection

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*?



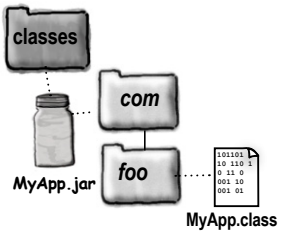
16 Data Structures

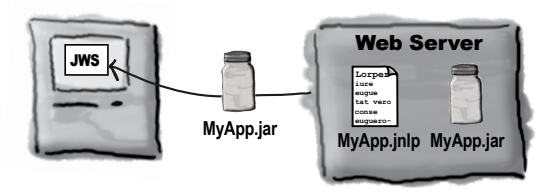
Sorting is a snap in Java. You have all the tools for collecting and manipulating your data without having to write your own sort algorithms The Java Collections Framework has a data structure that should work for virtually anything you'll ever need to do. Want to keep a list that you can easily keep 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?



17 Release Your Code

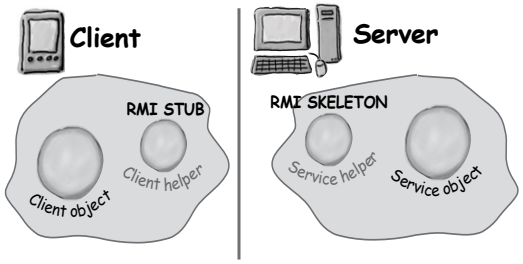
It's time to let go. You wrote your code. You tested your code. You refined your code. You told everyone you know that if you never saw a line of code again, that'd be fine. But in the end, you've created a work of art. The thing actually runs! But now what? In these final two chapters, we'll explore how to organize, package, and deploy your Java code. We'll look at local, semi-local, and remote deployment options including executable jars, Java Web Start, RMI, and Servlets. Relax. Some of the coolest things in Java are easier than you think.





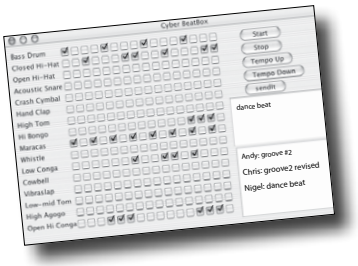
18 Distributed Computing

Being remote doesn't have to be a bad thing. Sure, things *are* easier when all the parts of your application are in one place, in one heap, with one JVM to rule them all. But that's not always possible. Or desirable. What if your application handles powerful computations? What if your app needs data from a secure database? In this chapter, we'll learn to use Java's amazingly simple Remote Method Invocation (RMI). We'll also take a quick peek at Servlets, Enterprise Java Beans (EJB) , and Jini.



A Appendix A

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

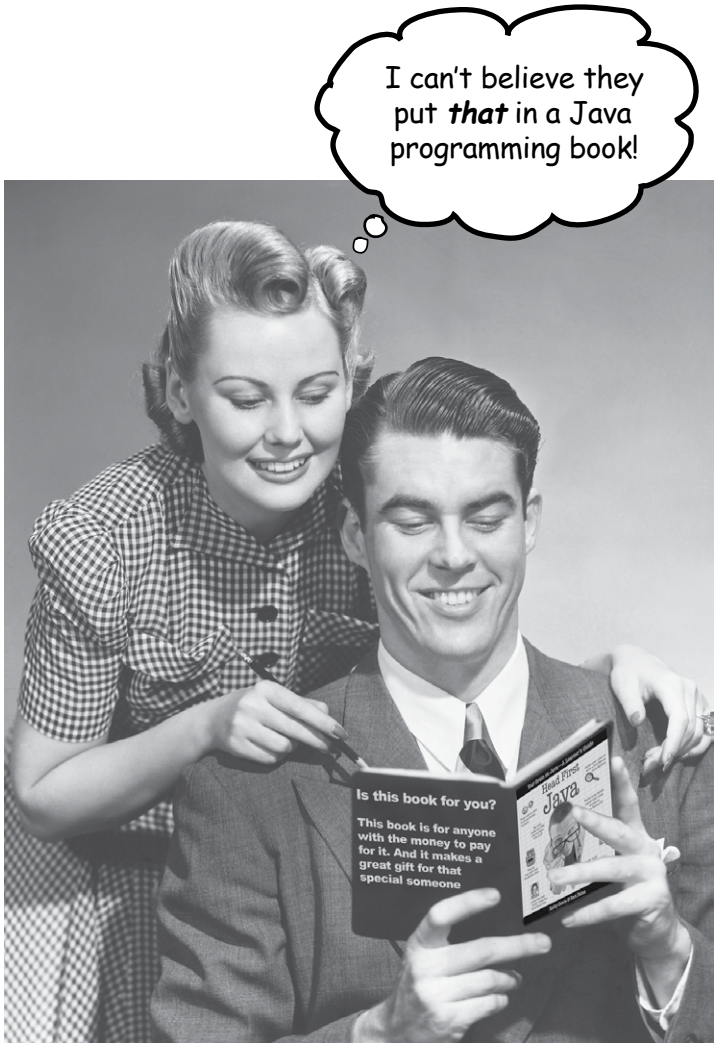


B Appendix B

The Top Ten Things that didn't make it into 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

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!

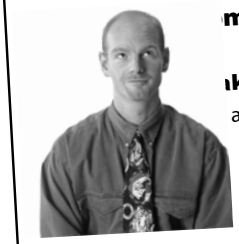
But imagine you're at home, or in a library. It's a safe, warm, tiger-free zone. You're studying. Getting ready for an exam. Or trying to learn some tough technical topic your boss thinks will take a week, ten days at the most.

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."

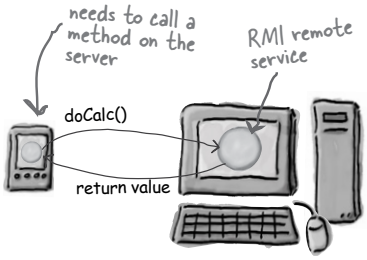
We think of a "Head First Java" reader as a learner.

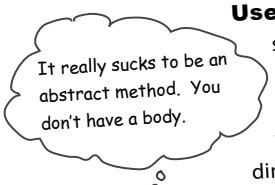
So what does it take to *learn* something? First, you have to *get* it, then make sure you don't *forget* it. It's not about pushing facts into your head. Based on the latest research in cognitive science, neurobiology, and educational psychology, *learning* takes a lot more than text on a page. We know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



me of the Head First learning principles:

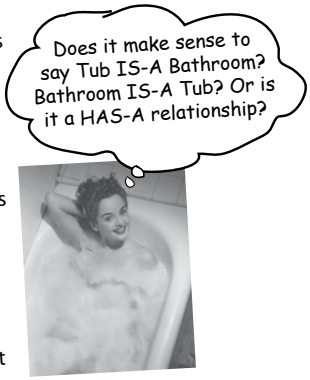
ıke it visual. Images are far more memorable than words alone, and make learning much more effective (Up to 89% improvement in recall and transfer studies). It also makes things more understandable. **Put the words within or near the graphics** they relate to, rather than on the bottom or on another 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, and 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?

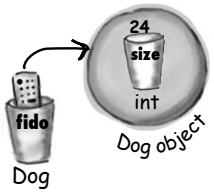
There's the slow, tedious way, or the faster, more effective way. The slow way is about sheer repetition. You obviously know that you *are* able to learn and remember even the dullest of topics, if you keep pounding on the same thing. With enough repetition, your brain says, "This doesn't *feel* important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

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 judgements.

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 refridgerator.

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 side-bars—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 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). For long examples or Ready-bake code, you can download the source files from wickedlysmart.com

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 (and *especially* not until after chapter 16). 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.

If you are using Java 6, don't worry! For the most part Java 6 added only a few minor additions to the API. In other words, this book is for you if you're using Java 5 or Java 6.

SETTING UP JAVA

- If you don't already have a **1.5** or greater **Java 2 Standard Edition SDK** (Software Development Kit), you need it. If you're on Linux, Windows, or Solaris, you can get it for free from java.sun.com (Sun's website for Java developers). It usually takes no more than two clicks from the main page to get to the J2SE downloads page. Get the latest *non-beta* version posted. The SDK includes everything you need to compile and run Java.
  If you're running Mac OS X 10.4, the Java SDK is already installed. It's part of OS X, and you don't have to do *anything* else. If you're on an earlier version of OS X, you have an earlier version of Java that will work for 95% of the code in this book.
  Note: This book is based on Java 1.5, but for stunningly unclear marketing reasons, shortly before release, Sun renamed it Java 5, while still keeping "1.5" as the version number for the developer's kit. So, if you see Java 1.5 or Java 5 or Java 5.0, or "Tiger" (version 5's original code-name), *they all mean the same thing.* There was never a Java 3.0 or 4.0—it jumped from version 1.4 to 5.0, but you will still find places where it's called 1.5 instead of 5. Don't ask.
  (Oh, and just to make it more entertaining, Java 5 and the Mac OS X 10.4 were both given the same code-name of "Tiger", and since OS X 10.4 is the version of the Mac OS you need to run Java 5, you'll hear people talk about "Tiger on Tiger". It just means Java 5 on OS X 10.4).
- The SDK does *not* include the **API documentation,** and you need that! Go back to java.sun.com and get the J2SE API documentation. You can also access the API docs online, without downloading them, but that's a pain. Trust us, it's worth the download.
- You need a **text editor**. Virtually any text editor will do (vi, emacs, pico), including the GUI ones that come with most operating systems. Notepad, Wordpad, TextEdit, etc. all work, as long as you make sure they don't append a ".txt" on to the end of your source code.
- Once you've downloaded and unpacked/zipped/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. For example, if the J2SDK puts a directory on your drive called "j2sdk1.5.0", look inside that directory and you'll find the "bin" directory where the Java binaries (the tools) live. The bin directory is the one you need a PATH to, so that when you type:

  ```
  % javac
  ```

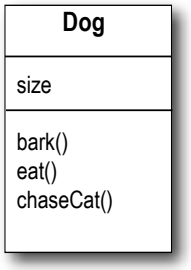
  at the command-line, your terminal will know how to find the *javac* compiler.
  Note: if you have trouble with your installation, we recommend you go to javaranch.com, and join the Java-Beginning forum! Actually, you should do that whether you have trouble or not.

Note: much of the code from this book is available at wickedlysmart.com

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 until the end of the book.

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, so we cover them in depth. But we save them for the end of the book (chapter 17). Relax while you ease into Java, gently.

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 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. (Some of our *suggested* answers are available on wickedlysmart.com)

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 Editors

"Credit goes to all, but mistakes are the sole reponsibility of the author...". Does anyone really believe that? See the two people on this page? If you find technical problems, it's probably *their* fault. :)

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 Masters 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 Valentin Crettaz has a Masters 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 co-founder 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:

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 acknowledgements*.

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:

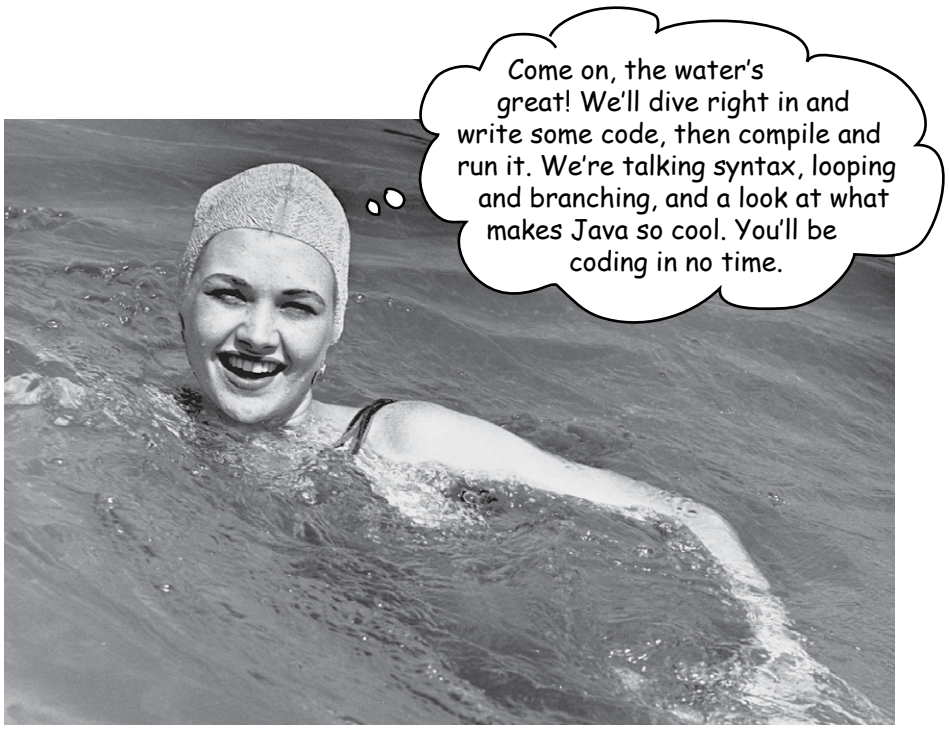
Paul Wheaton, the javaranch Trail Boss for supporting thousands of Java learners.
Solveig Haugland, mistress of J2EE and author of "Dating Design Patterns".
Authors **Dori Smith** and **Tom Negrino (backupbrain.com)**, for helping us navigate the tech book world.
Our Head First partners in crime, **Eric 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 acknowledgements is because we're testing the theory that everyone mentioned in a book acknowledgement will buy at least one copy, probably more, what with relatives and everything. If you'd like to be in the acknowledgement of our *next* book, and you have a large family, write to us.

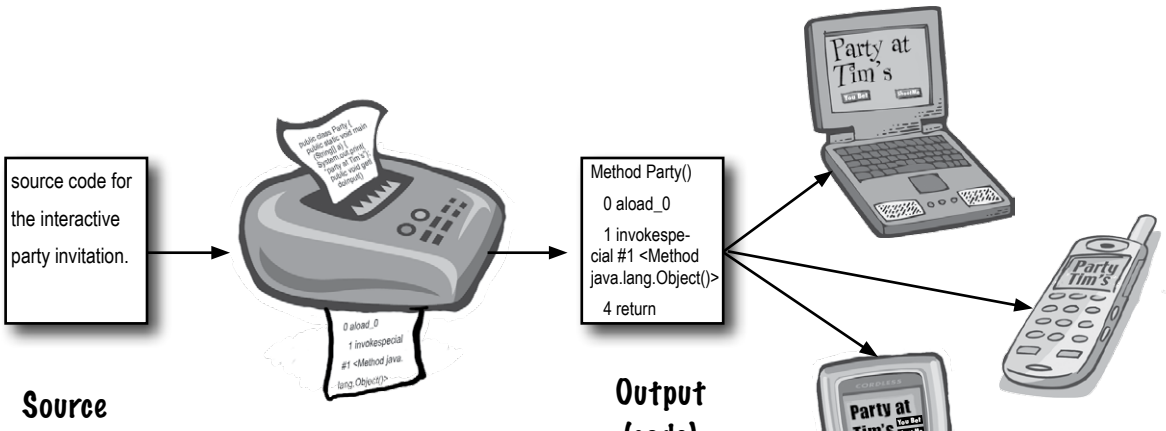
Breaking the Surface



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 applet to work. But *you*, why, *you* get to ride the **sleeker, faster, much more powerful** 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 don't have a physical Java Machine, but they all have a ***virtual*** Java machine (implemented in software) running inside their electronic gadgets. 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, 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
```

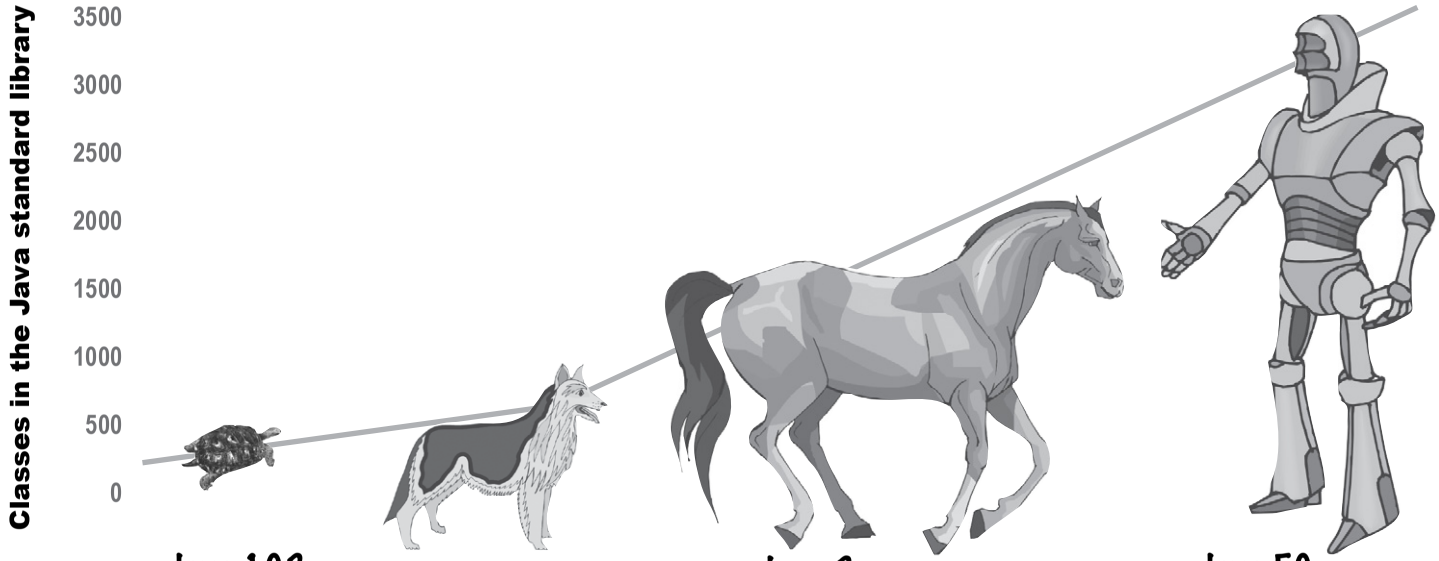
Virtual Machines

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.)

A very brief history of Java



Java 1.02

250 classes

Slow.

Cute name and logo. Fun to use. Lots of bugs. ***Applets*** are the Big Thing.

Java 1.1

500 classes

A ***little*** faster.

More capable, ***friendlier***. Becoming very ***popular***. Better GUI code.

Java 2 (versions 1.2 - 1.4)

2300 classes

Much faster.

Can (sometimes) run at native speeds. Serious, ***powerful***. Comes in three flavors: Micro Edition (J2ME), Standard Edition (J2SE) and Enterprise Edition (J2EE). Becomes the ***language of choice*** for new enterprise (especially web-based) and mobile applications.

Java 5.0 (versions 1.5 and up)

3500 classes

More power, easier to develop with.

Besides adding more than a thousand additional classes, Java 5.0 (known as "Tiger") added major changes to the language itself, making it easier (at least in theory) for programmers and giving it new features that were popular in other languages.

Look how easy it is to write Java.

Sharpen your pencil

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

Q: I see Java 2 and Java 5.0, but was there a Java 3 and 4? And why is it Java 5.*0* but not Java 2.*0*?

A: The joys of marketing... when the version of Java shifted from 1.1 to 1.2, the changes to Java were so dramatic that the marketers decided we needed a whole new "name", so they started calling it ***Java 2***, even though the actual version of Java was 1.2. But versions 1.3 and 1.4 were still considered ***Java 2***. There never *was* a Java 3 or 4. Beginning with Java version 1.5, the marketers decided once again that the changes were so dramatic that a new name was needed (and most developers agreed), so they looked at the options. The next number in the name sequence would be "3", but calling Java 1.5 ***Java 3*** seemed more confusing, so they decided to name it ***Java 5.0*** to match the "5" in version "1.5".

So, the original Java, versions 1.02 (the first official release) through 1.1, were just "Java". Versions 1.2, 1.3, and 1.4 were "Java 2". And beginning with version 1.5, Java is called "Java 5.0". But you'll also see it called "Java 5" (without the ".0") and "Tiger" (its original code-name). We have no idea what will happen with the *next* release...

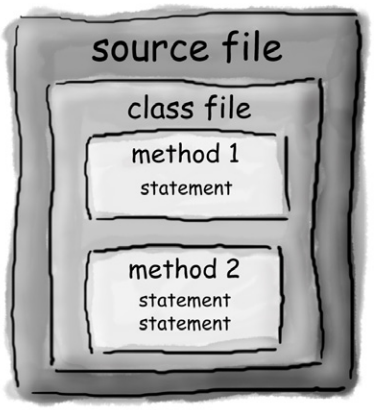
Sharpen your pencil answers

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



Put a class in a source file.

Put methods in a class.

Put statements in a method.

What goes in a source file?

A source code file (with the *.java* extension) 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;
   }
}
```

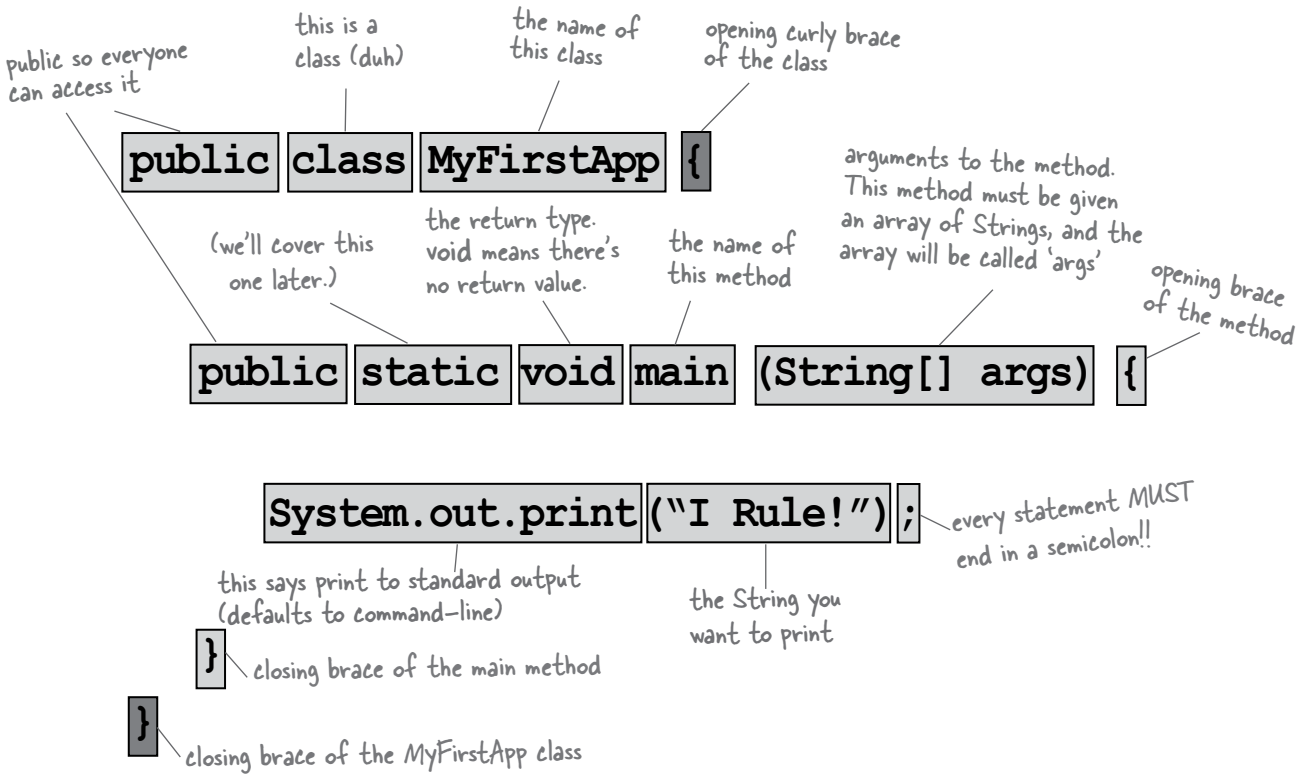
statements

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

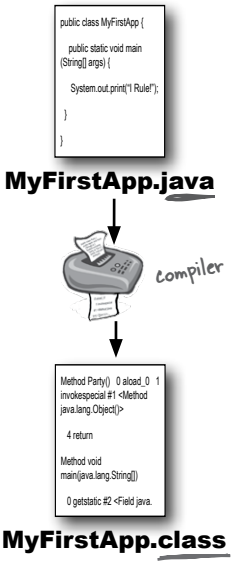
In Java, everything goes in a **class**. You'll type your source code file (with a *.java* extension), then compile it into a new class file (with a *.class* extension). When you run your program, you're really running a *class*.

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, we go deeper into the whole *class* thing, but for now, all you need to think 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.




```
public class MyFirstApp {

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

}
```

1 Save

`MyFirstApp.java`

2 Compile

```
javac MyFirstApp.java
```

3 Run

```
File Edit Window Help Scream
%java MyFirstApp
I Rule!
The World
```

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

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 three standard looping constructs: *while*, *do-while*, and *for*. You'll get the full loop scoop later in the book, but not for awhile, so let's do *while* for now.

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* including:

< (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 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. You might write test classes, though, that have main methods for testing your *other* classes.

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 beer...", you'll say, "***if*** there's still beer..."

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

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

code output

The code above 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 beer, keep coding, *else* (otherwise) get more beer, and then continue on..."

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

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

new output

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

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

Did you spot the difference?

System.out.***println*** inserts a newline (think of 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");
    }
  }
}
```

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, "99 bottles of beer."

```
public class BeerSong {
   public static void main (String[] args) {
     int beerNum = 99;
     String word = "bottles";

     while (beerNum > 0) {

        if (beerNum == 1) {
          word = "bottle"; // singular, as in ONE bottle.
        }

        System.out.println(beerNum + " " + word + " of beer on the wall");
        System.out.println(beerNum + " " + word + " of beer.");
        System.out.println("Take one down.");
        System.out.println("Pass it around.");
        beerNum = beerNum - 1;

        if (beerNum > 0) {
           System.out.println(beerNum + " " + word + " of beer on the wall");
        } else {
           System.out.println("No more bottles of beer on the wall");
        } // end else
     } // end while loop
  } // end main method
} // end class
```

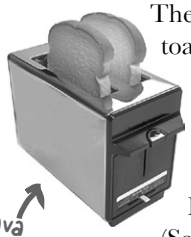
There's still one little flaw in our code. It compiles and runs, but the output isn't 100% perfect. See if you can spot the flaw , and fix it.

Monday morning at Bob's

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 Nokia Navigator™ cell phone, "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 a little trip to Radio Shack™ 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.***

***You* can have a Java-enabled home**. Stick with a sensible solution using Java, Ethernet, and Jini technology. Beware of imitations using other so-called "plug and play" (which actually means "plug and play with it for the next three days trying to get it to work") or "portable" platforms. Bob's sister Betty tried one of those *others*, and the results were, well, not very appealing, or safe. Bit of a shame about her dog, too...

Could this story be true? Yes and no. While there *are* versions of Java running in devices including PDAs, cell phones (*especially* cell phones), pagers, rings, smart cards, and more –you might not find a Java toaster or dog collar. But even if you can't find a Java-enabled version of your favorite gadget, you can still run it as if it *were* a Java device by controlling it through some other interface (say, your laptop) that *is* running Java. This is known as the Jini *surrogate architecture*. Yes you *can* have that geek dream home.

**IP multicast* if you're gonna be all picky about protocol

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

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

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

1     // make three sets of words to choose from. Add your own!
      String[] wordListOne = {"24/7","multi-
Tier","30,000 foot","B-to-B","win-win","front-
end", "web-based","pervasive", "smart", "six-
sigma","critical-path", "dynamic"};

      String[] wordListTwo = {"empowered", "sticky",
"value-added", "oriented", "centric", "distributed",
"clustered", "branded","outside-the-box", "positioned",
"networked", "focused", "leveraged", "aligned",
"targeted", "shared", "cooperative", "accelerated"};

      String[] wordListThree = {"process", "tipping-
point", "solution", "architecture", "core competency",
"strategy", "mindshare", "portal", "space", "vision",
"paradigm", "mission"};

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

3     // generate three random numbers
      int rand1 = (int) (Math.random() * oneLength);
      int rand2 = (int) (Math.random() * twoLength);
      int rand3 = (int) (Math.random() * threeLength);

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

5     // print out the phrase
      System.out.println("What we need is a " + phrase);
      }
}
```

Phrase-O-Matic

How it works.

In a nutshell, the program makes three lists of words, then randomly picks one word from each of the three lists, and prints out the result. Don't worry if you don't understand *exactly* what's happening in each line. For gosh sakes, 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.

what we need here is a...

pervasive targeted process

dynamic outside-the-box tipping-point

smart distributed core competency

24/7 empowered mindshare

30,000 foot win-win vision

six-sigma networked portal

3. We need three random numbers. Java ships out-of-the-box, off-the-shelf, shrink-wrapped, and core competent with a set of math methods (for now, think of them as functions). The **random()** method returns a random number between 0 and not-quite-1, so we have to multiply it by the number of elements (the array length) in the list we're using. We have to force the result to be an integer (no decimals allowed!) so we put in a cast (you'll get the details in chapter 4). It's the same as if we had any floating point number that we wanted to convert to an integer:

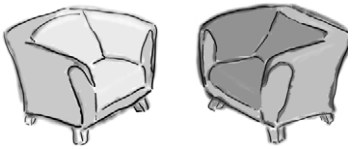
```
int x = (int) 24.6;
```

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 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... voila! *We're in marketing.*

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 guy 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 wall paper, kindling, lining the bird cage what*ever*, 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 nit-picky 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. Java's had a challenging enough time convincing people that it's finally fast and powerful enough for most jobs.

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 doing your word processing by writing raw postscript. And I would appreciate it if you would *not* refer to me as "buddy."

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

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—

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 him 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.

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("-");
```

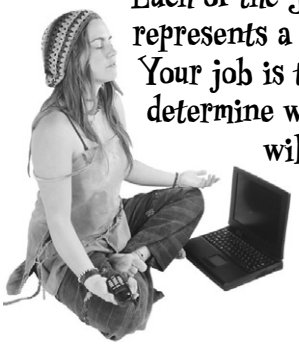
```
while (x > 0) {
```

Output:

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

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?



A

```
class Exercise1b {
  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

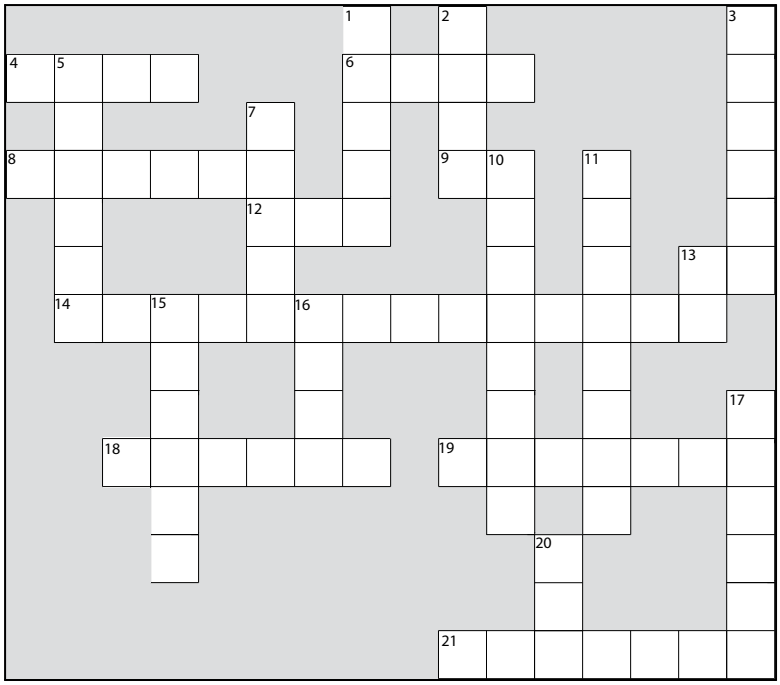
```
class Exercise1b {
    int x = 5;
    while ( x > 1 ) {
      x = x - 1;
      if ( x < 3) {
        System.out.println("small x");
      }
    }
}
```

puzzle: crossword

JavaCross 7.0

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. Dept. of LAN jockeys
15. Shocking modifier
16. Just gotta have one
17. How to get things done
20. Bytecode consumer

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

```
class Test {
  public static void main(String [] args) {
    int x = 0;
    int y = 0;
    while ( x < 5 ) {
      // candidate code goes here
      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
```

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.

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

Code Magnets:

```
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
```

A

```
class Exercise1b {
  public static void main(String [] args) {
    int x = 1;
    while ( x < 10 ) {
      x = x + 1;
      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

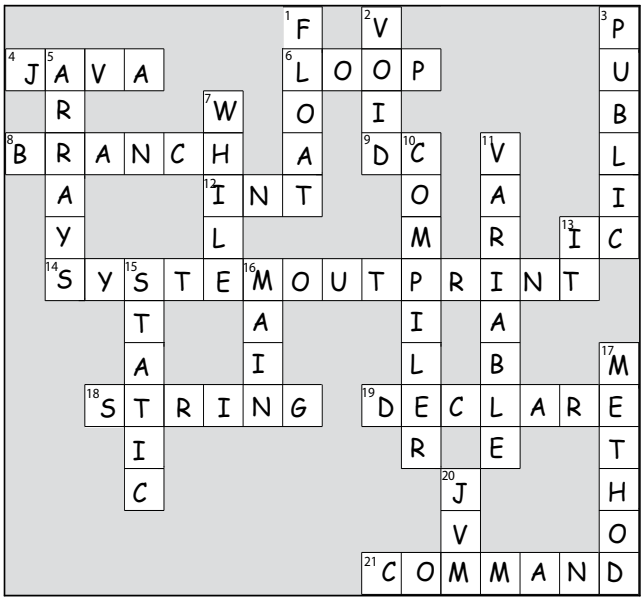
```
class Foo {
  public static void main(String [] args) {
    int x = 5;
    while ( x > 1 ) {
      x = x - 1;
      if ( x < 3) {
        System.out.println("small x");
      }
    }
  }
}
```

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

C

```
class Exercise1b {
  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.




```
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
```

```
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
```

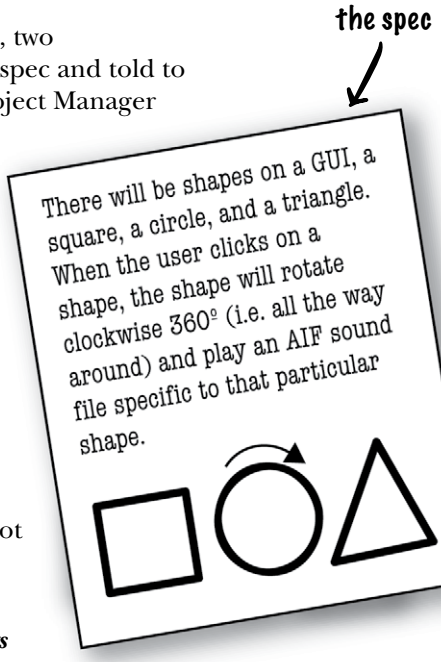
A Trip to Objectville

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 one of those cool Aeron™ chairs all the Silicon Valley guys have. Larry, the procedural programmer, and Brad, the OO guy, both knew this would be a piece of cake.



Larry, sitting in his cube, thought to himself, "What are the things this program has to *do*? What ***procedures*** do we need?". And he answered himself , "**rotate** and **playSound**." So off he went to build the procedures. After all, what *is* a program if not a pile of procedures?

Brad, meanwhile, kicked back at the cafe 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 objects 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 Larry built their programs, and for the answer to your burning question, ***"So, who got the Aeron?"***

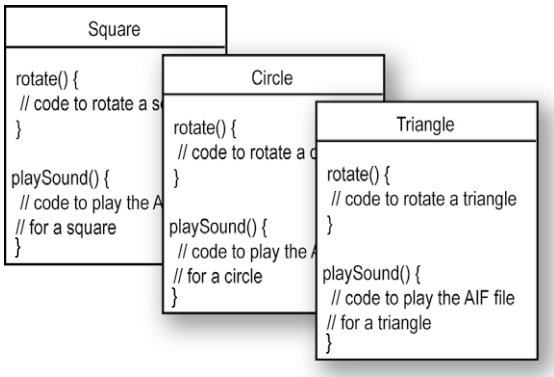
In Larry's cube

As he had done a gazillion times before, Larry set about writing his **Important Procedures**. He 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

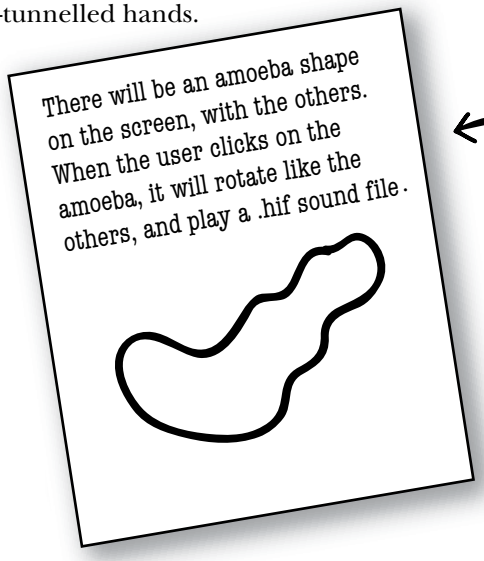


Larry thought he'd nailed it. He could almost feel the rolled steel of the Aeron beneath his...

But wait! There's been a spec change.

"OK, *technically* you were first, Larry," 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 Larry, knowing that spec-change-no-problem was a fantasy. *"And yet Brad looks strangely serene. What's up with that?"* Still, Larry held tight to his core belief that the OO way, while cute, was just slow. And that if you wanted to change his mind, you'd have to pry it from his cold, dead, carpal-tunnelled hands.



what got added to the spec

Back in Larry's cube

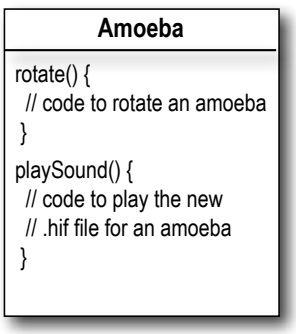
The rotate procedure would still work; the code used a lookup table to match a shapeNum to an actual shape graphic. But ***playSound would have to change.*** And what the heck is a .hif file?

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

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

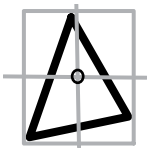
At Brad's laptop at the beach

Brad smiled, sipped his margarita, 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.



Larry snuck in just moments ahead of Brad.

(Hah! So much for that foofy OO nonsense). But the smirk on Larry'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..."



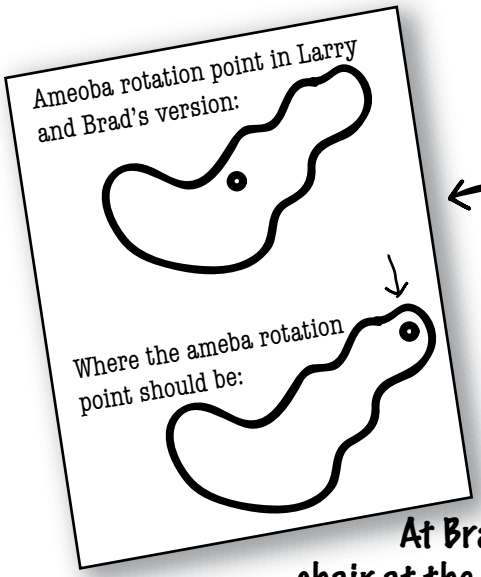
Turns out, both programmers had written their rotate code like this:

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 Larry, 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 his head said, *"Big Mistake. Do you honestly think the spec won't change again?"*



What the spec conveniently forgot to mention

Back in Larry's cube

He figured he 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 (wirelessly) 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
  // .hif file for an amoeba
}
```

So, Brad the OO guy got the chair, right?

Not so fast. Larry found a flaw in Brad's approach. And, since he was sure that if he got the chair he'd also get Lucy in accounting, he had to turn this thing around.

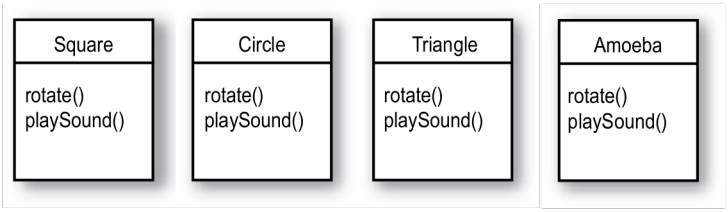
LARRY: You've got duplicated code! The rotate procedure is in all four Shape things.

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

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

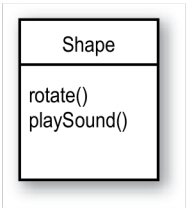
BRAD: Oh, I guess you didn't see the final design. Let me show you how OO **inheritance** works, Larry.

What Larry wanted
(figured the chair would impress her)

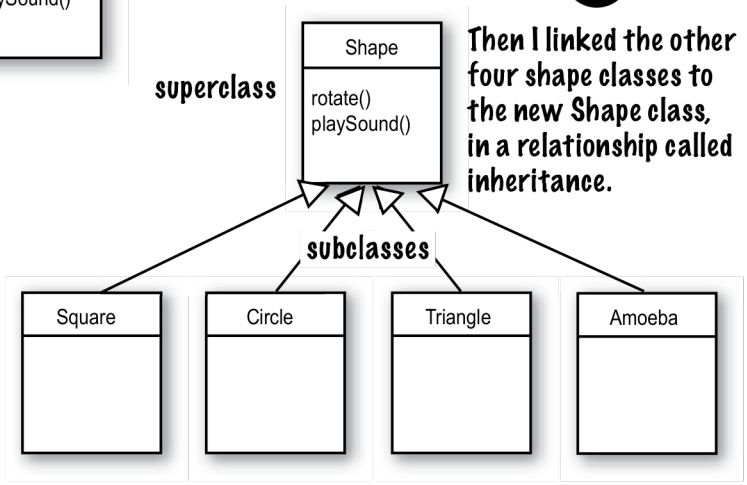


1 I looked at what all four classes have in common.

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.*

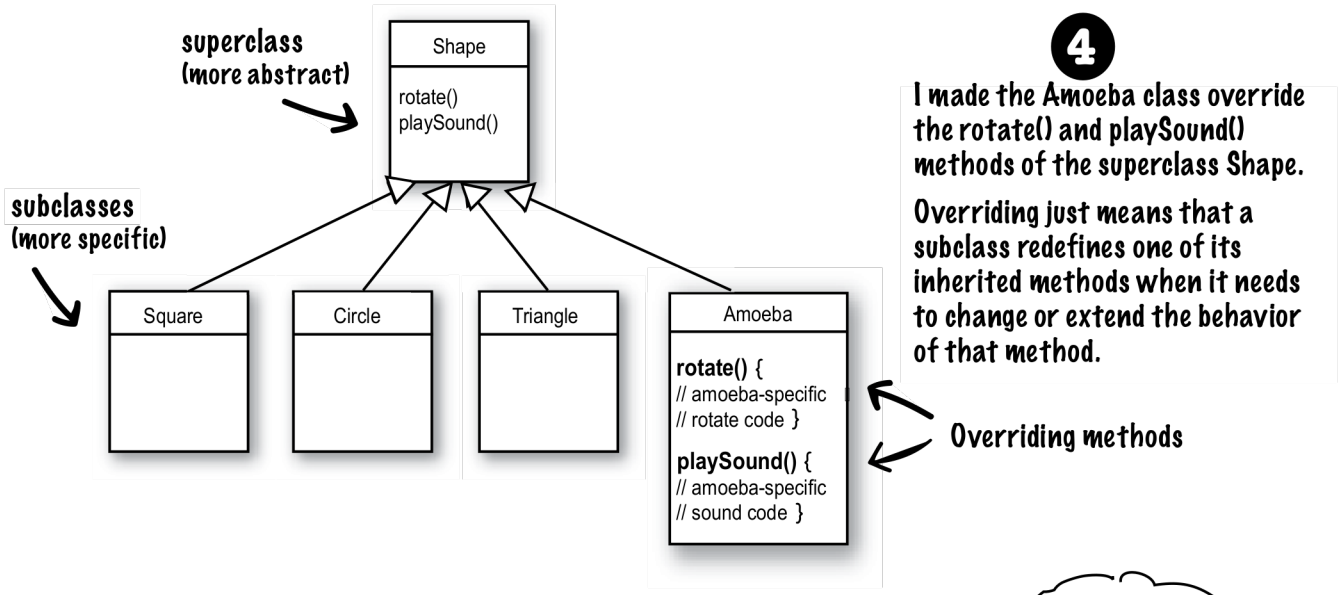
What about the Amoeba rotate()?

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

BRAD: Method.

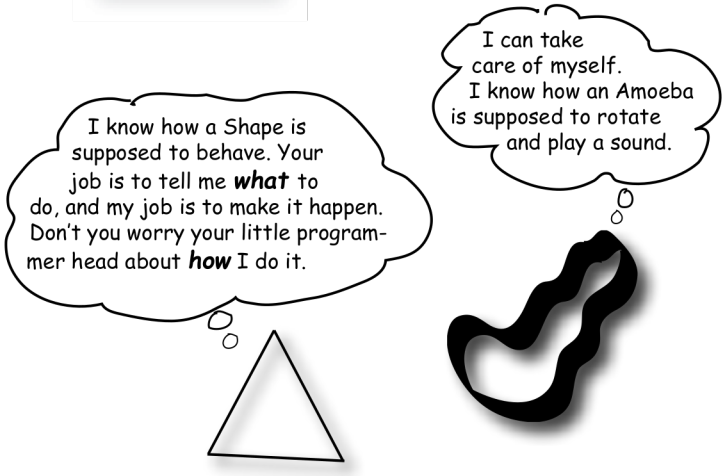
LARRY: Whatever. How can amoeba do something different if it "inherits" its functionality from the Shape class?

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.



LARRY: 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?

Amy from the second floor.

(unbeknownst to all, the Project Manager had given the spec to *three* programmers.)

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."
-Josh, 22, beer drinker

"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 just said that. He hasn't written a line of code in 5 years."
-Daryl, 44, works for Chris

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

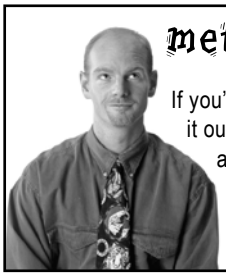
BRAIN POWER

Time to pump some neurons.

You just read a story bout 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?

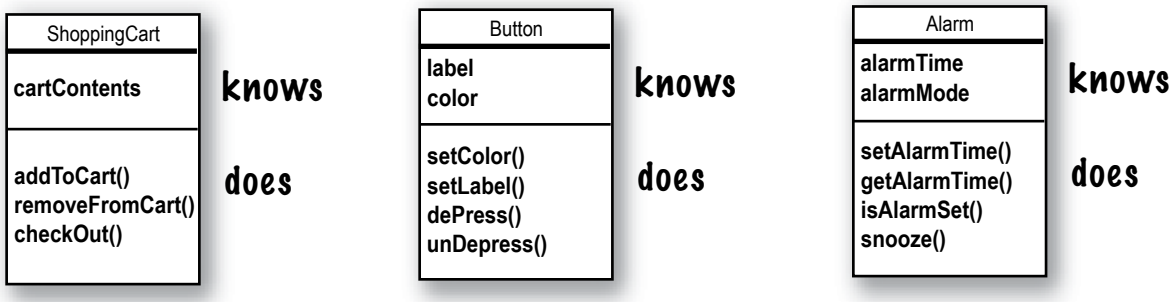


metacognitive tip

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**

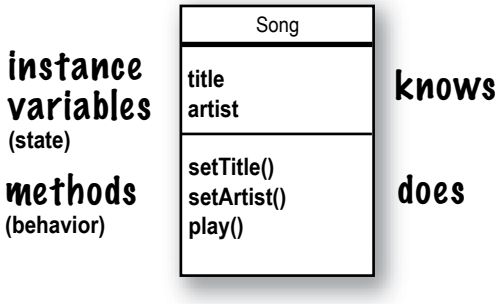


Things an object *knows* about itself are called

- 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.

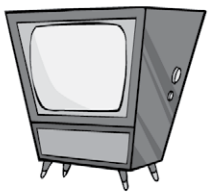
Think of instance as another way of saying object.

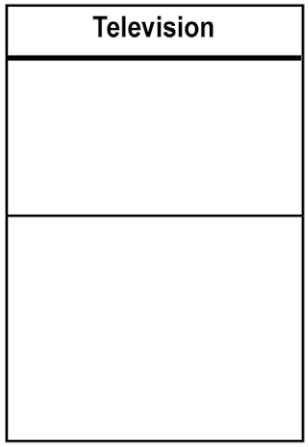
Things an object can ***do*** are called **methods**. When you design a class, you think about the data an object will need to know about itself, and you also design the methods that operate on that data. It's common for an object to have methods that read or write the values of the instance variables.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

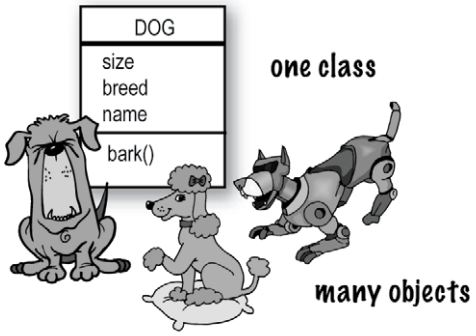
Sharpen your pencil

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





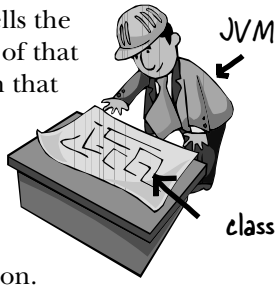
What's the difference between a class and an object?



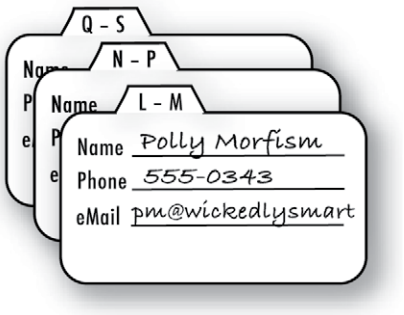
A class is not an object.

(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.



Look at it this way...



An object is like one entry in your address book.

One analogy for objects is a packet of unused Rolodex™ cards. Each card has the same blank fields (the instance variables). When you fill out a card you are creating an instance (object), and the entries you make on that card represent its state.

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

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

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 class type.

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 type (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

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

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

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

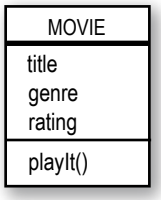
dot operator

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

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 MovieTestDrive class also invokes (calls) a method on one of the objects. Fill in the chart to the right with the values the three objects have at the end of main().

object 1	title genre rating
object 2	title genre rating
object 3	title genre rating

Quick! Get out of main!

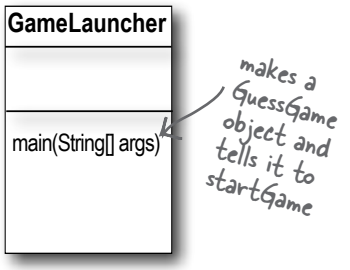
As long as you're in main(), you're not really in Objectville. It's fine for a test program to run within the main method, but in a true OO application, you need objects talking to other objects, as opposed to a static main() method creating and testing objects.

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 , 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 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.)

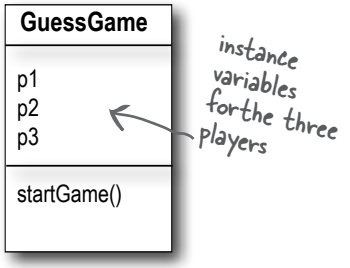
As a 'sneak preview', though, of how a real Java application might behave, here's a little example. Because we're still at the earliest stages of learning Java, we're working with a small toolkit, so you'll find this program a little clunky and inefficient. You might want to think about what you could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

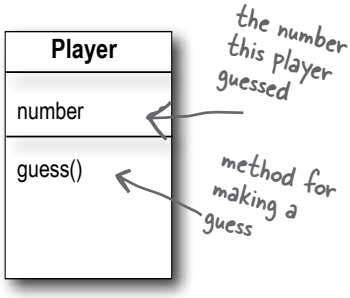
Summary:

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, 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

```
public class Player {
   int number = 0;  // where the guess goes

   public void guess() {
     number = (int) (Math.random() * 10);
     System.out.println("I'm guessing "
                        + number);
   }
}


public class GameLauncher {
   public static void main (String[] args) {
      GuessGame game = new GuessGame();
      game.startGame();
   }
}
```

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 installer programs that let you bundle your classes with a variety of JVM's (say, for different platforms), and put it all on a CD-ROM. Then the end-user can install the correct version of the JVM (assuming they don't already have it on their machine.)

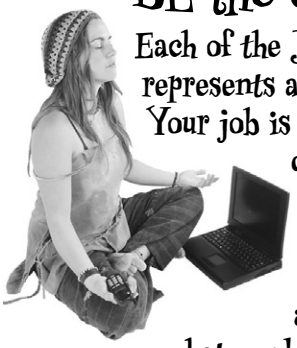
Q: What if I have a hundred classes? Or a thousand? Isn't that a big pain to deliver all those individual files? Can I bundle them into one *Application Thing*?

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

BULLET POINTS

- Object-oriented programming lets you extend a program without having to touch previously-tested, working code.
- All Java code is defined in a **class**.
- A class describes how to make an object of that class type. **A class is like a blueprint.**
- An object can take care of itself; you don't have to know or care *how* the object does it.
- An object **knows** things and **does** things.
- Things an object knows about itself are called **instance variables**. They represent the *state* of an object.
- Things an object does are called **methods**. They represent the *behavior* of an object.
- When you create a class, you may also want to create a separate test class which 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 TapeDeck {

  boolean canRecord = false;

  void playTape() {
    System.out.println("tape playing");
  }

  void recordTape() {
    System.out.println("tape recording");
  }
}

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

    t.canRecord = true;
    t.playTape();

    if (t.canRecord == true) {
      t.recordTape();
    }
  }
}
```

B

```
class DVDPlayer {

  boolean canRecord = false;

  void recordDVD() {
    System.out.println("DVD recording");
  }
}

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

    DVDPlayer d = new DVDPlayer();
    d.canRecord = true;
    d.playDVD();

    if (d.canRecord == true) {
      d.recordDVD();
    }
  }
}
```

Code Magnets

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

```
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. Some of the exercises and puzzles in this book might have more than one correct answer. If you find another correct answer, give yourself bonus points!

Output

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

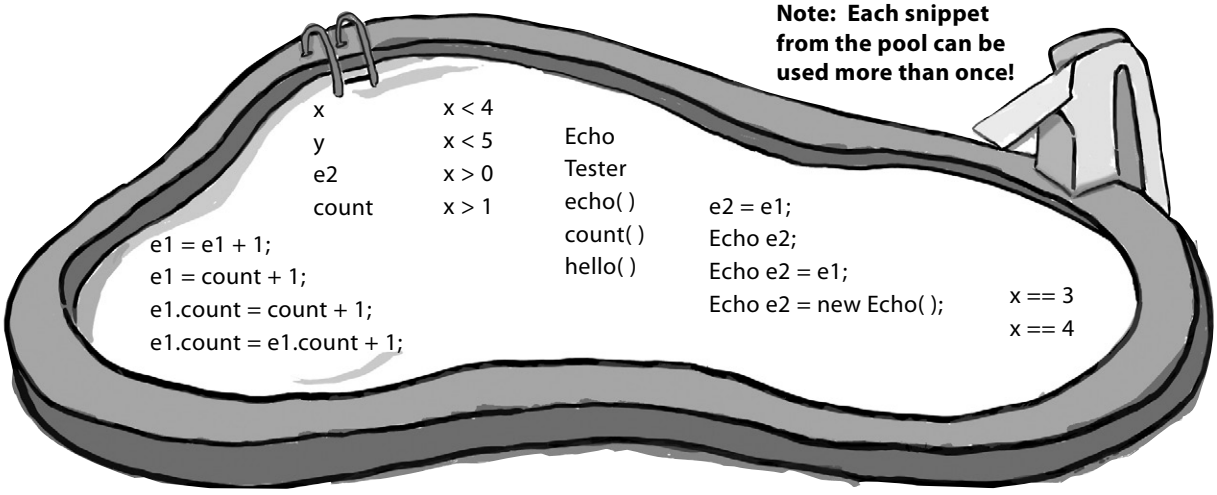
Bonus Question !

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

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

    _____________________________

    int x = 0;

    while ( _____________ ) {

      e1.hello();

      _____________________________

      if ( _____________ ) {

        e2.count = e2.count + 1;

      }

      if ( _____________ ) {

        e2.count = e2.count + e1.count;

      }

      x = x + 1;

    }

    System.out.println(e2.count);

  }
}
```

```
class _____________ {

  int _____________ = 0;

  void _____________ {

    System.out.println("helloooo... ");

  }
}
```

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



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. _____

Exercise Solutions

Code Magnets:

```
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:

A

```
class TapeDeck {
  boolean canRecord = false;
  void playTape() {
    System.out.println("tape playing");
  }
  void recordTape() {
    System.out.println("tape recording");
  }
}

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

    TapeDeck t = new TapeDeck( );
    t.canRecord = true;
    t.playTape();

    if (t.canRecord == true) {
      t.recordTape();
    }
  }
}
```

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

B

```
class DVDPlayer {
  boolean canRecord = false;
  void recordDVD() {
    System.out.println("DVD recording");
  }
  void playDVD ( ) {
    System.out.println("DVD playing");
  }
}

class DVDPlayerTestDrive {
  public static void main(String [] args) {
    DVDPlayer d = new DVDPlayer();
    d.canRecord = true;
    d.playDVD();
    if (d.canRecord == true) {
      d.recordDVD();
    }
  }
}
```

The line: d.playDVD( ); wouldn't compile without a method !

Puzzle Solutions

Pool Puzzle

```
public class EchoTestDrive {
  public static void main(String [] args) {
    Echo e1 = new Echo();
    Echo e2 = new Echo( );  // the correct answer
          - or -
    Echo e2 = e1;   // is the bonus 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... ");
  }
}
```

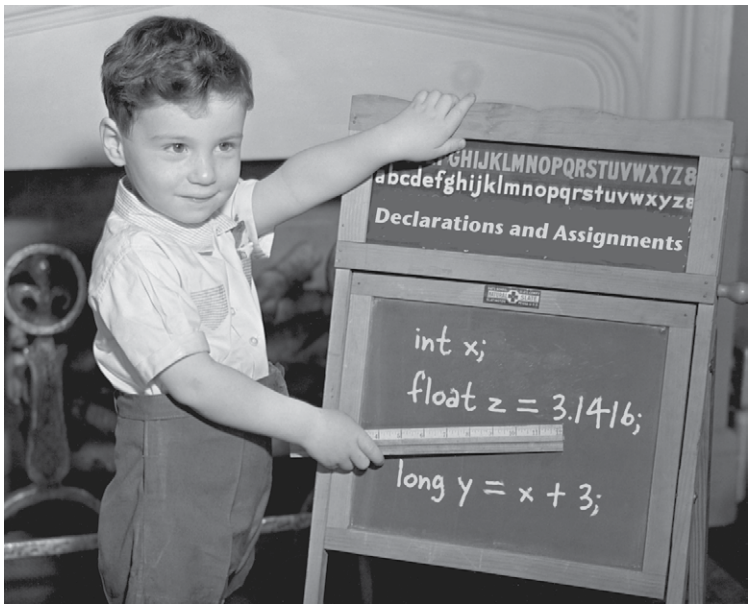
```
File  Edit  Window  Help  Assimilate
%java EchoTestDrive
helloooo...
helloooo...
helloooo...
helloooo...
10
```

Who am I?

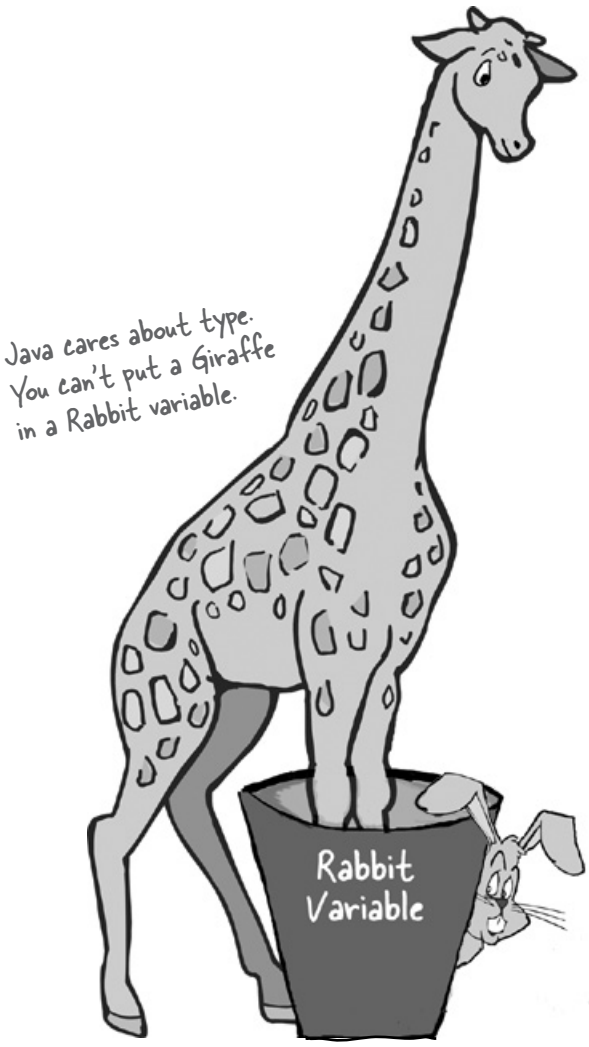
I am compiled from a .java file.	class
My instance variable values can be different from my buddy's values.	object
I behave like a template.	class
I like to do stuff.	object, method
I can have many methods.	class, object
I represent 'state'.	instance variable
I have behaviors.	object, class
I am located 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 come in two flavors: primitive and reference. 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 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 *acknowledge to the compiler* that you know you might lose precision (like, everything after the decimal point).

The compiler can spot most problems:

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

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

For all this type-safety to work, you must declare the type of your variable. Is it an integer? a Dog? A single character? Variables come in two flavors: ***primitive*** and ***object reference***. Primitives hold fundamental values (think: simple bit patterns) including integers, booleans, and floating point numbers. Object references hold, well, *references* to *objects* (gee, didn't *that* clear it up.)

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

variables must have a type

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

variables must have a name

```
int count;
```

type — name

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

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

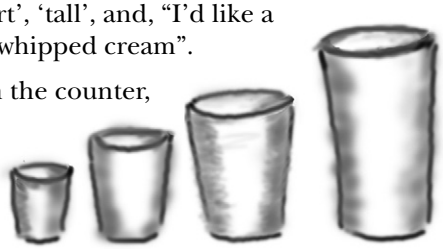
When you think of Java variables, think of cups. Coffee cups, tea cups, giant cups that hold lots and lots of beer, those big cups the popcorn comes in at the movies, cups with curvy, sexy handles, and cups with metallic trim that you learned can never, ever go in the microwave.

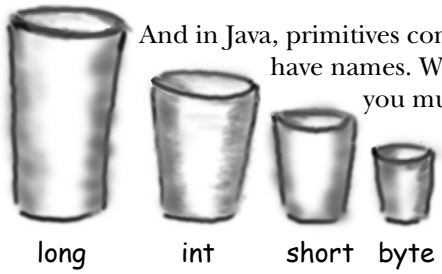
A variable is just a cup. A container. It *holds* something.

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 coffeehouse. 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 a 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 = 3456789;
float f = 32.5f;
```

Note the 'f'. Gotta have that with a float, because Java thinks anything with a floating point is a double, unless you use 'f'.

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 isCrazy;`	declare a boolean named *isCrazy* (no assignment)
`isCrazy = 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;
```

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 when we looked at writing our first main class:

```
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.

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

Java's keywords and other reserved words (in no useful order). If you use these for names, the compiler will 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."

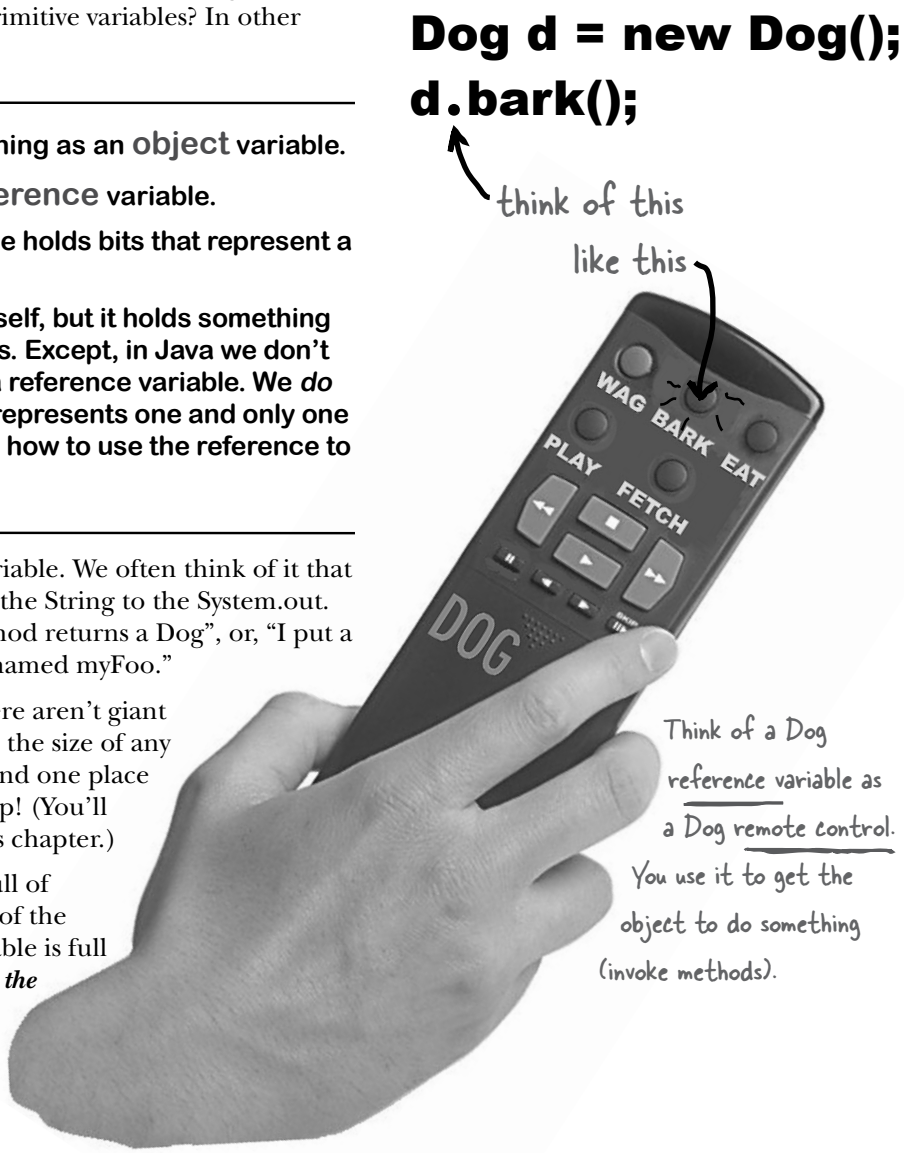
But that's not what happens. There aren't giant expandable cups that can grow to the size of any object. Objects live in one place and one place only—the garbage collectible heap! (You'll learn more about that later in this chapter.)

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.



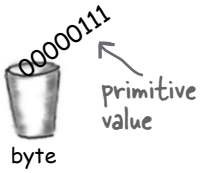
An object reference is just another variable value.

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).

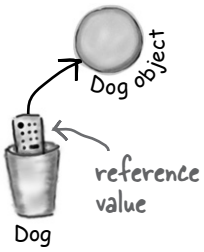


Reference Variable

```
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!



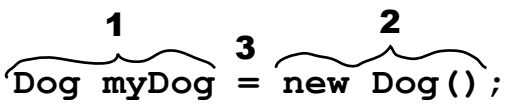
With primitive variables, the value of the variable is... the *value* (5, -26.7, 'a').

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 1's and 0's 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 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

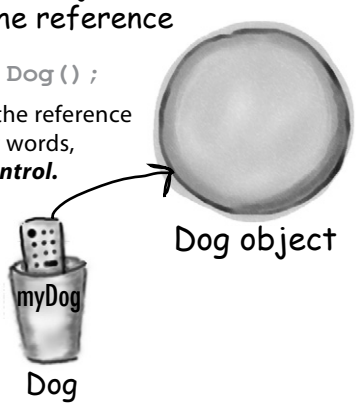
```
Dog myDog = new Dog();
```

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.)

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

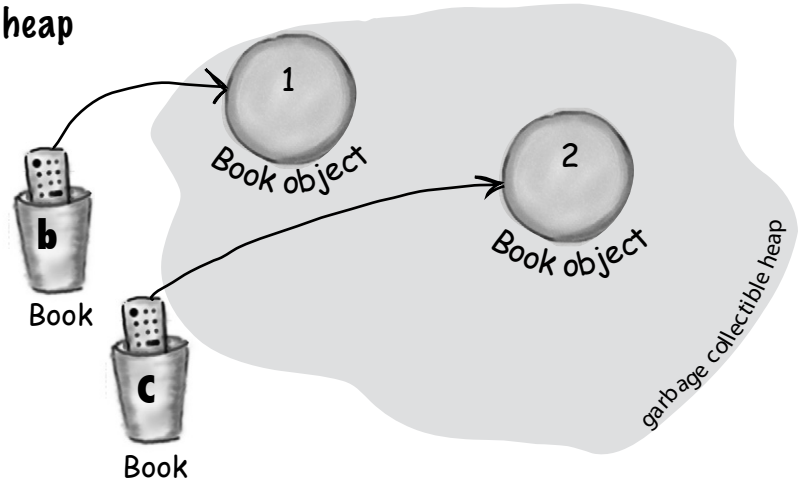
```
Book b = new Book();
Book c = new Book();
```

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;
```

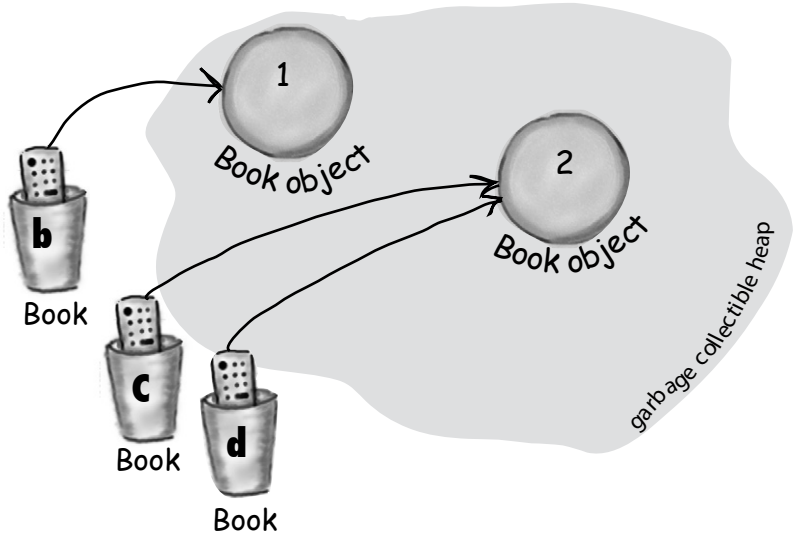
Declare a new Book reference variable. Rather than creating a new, third Book object, assign the value of variable ***c*** to variable ***d***. But what does this mean? It's like saying, "Take the bits in ***c***, make a copy of them, and stick that copy into ***d***."

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



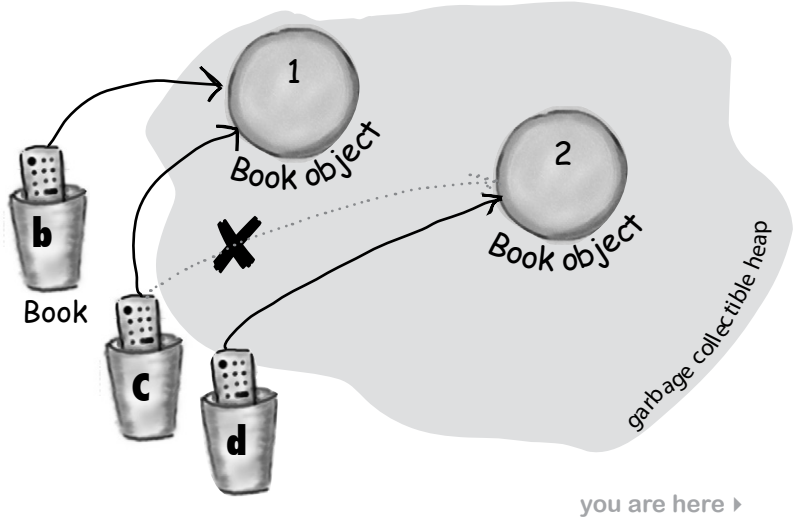
```
c = b;
```

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.

References: 3

Objects: 2



Life and death on the heap

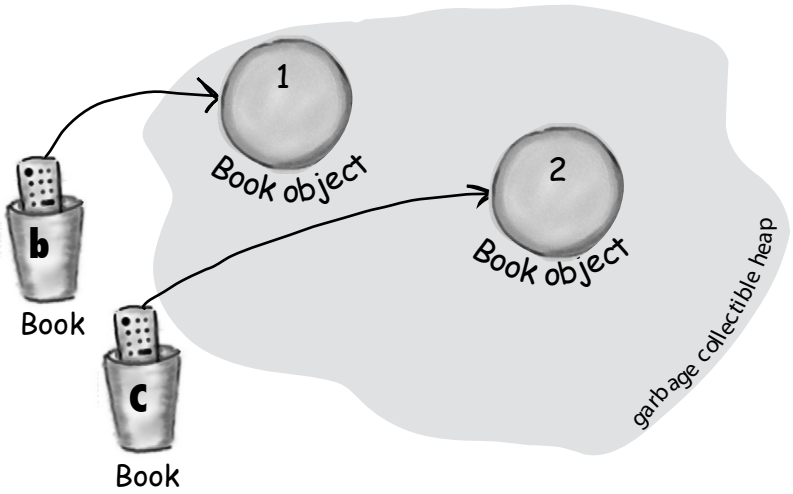
```
Book b = new Book();
Book c = new Book();
```

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.

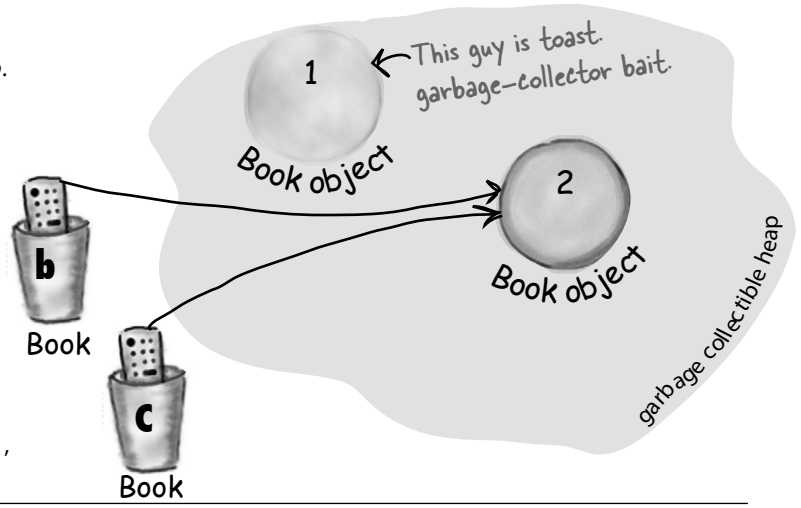
Both b and c refer to the same object. Object 1 is abandoned and eligible for Garbage Collection (GC).

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.

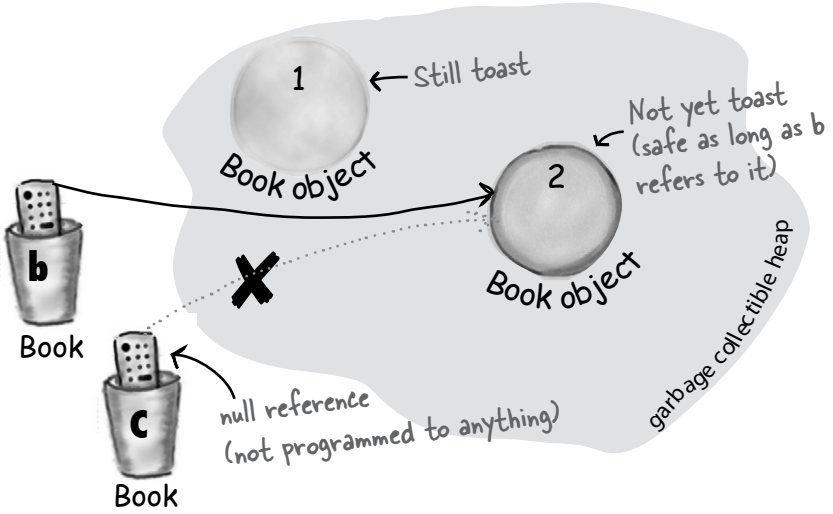
Object 2 still has an active reference (b), and as long as it does, the object is not eligible for GC.

Active References: 1

null References: 1

Reachable Objects: 1

Abandoned Objects: 1



An array is like a tray of cups

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

```
int[] nums;
```

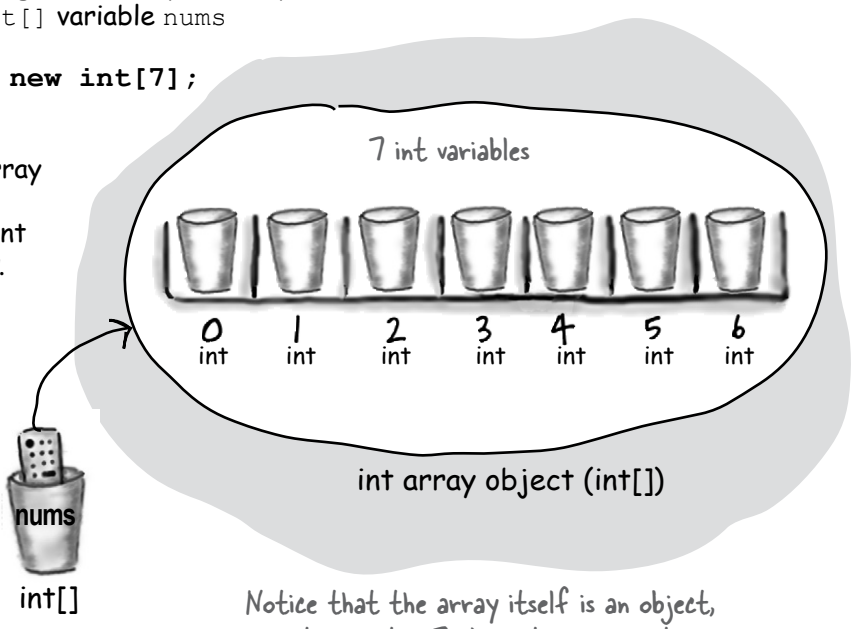
2. Create a new int array with a length of 7, and assign it to the previously-declared `int[]` variable `nums`

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

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

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

7 int variables



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

Arrays are objects too

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 above – *the array is an object, even though it's an array of primitives.*

Arrays are always objects, whether they're declared to hold primitives or object references. But you can have an array object that's declared to *hold* primitive values. In other words, the array object can have *elements* which are primitives, but the array itself is *never* a primitive. Regardless of what the array holds, the array itself is always an object!

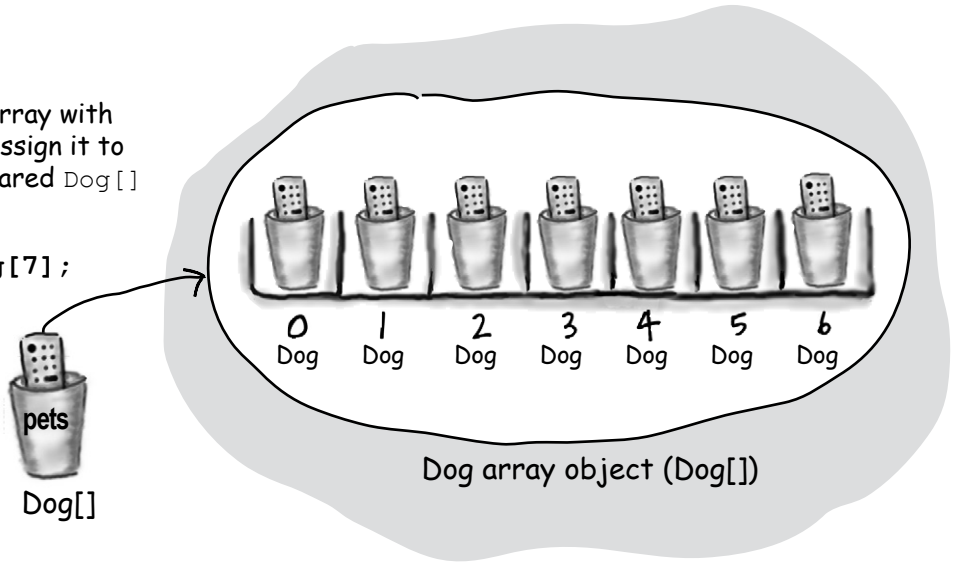
Make an array of Dogs

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];
```



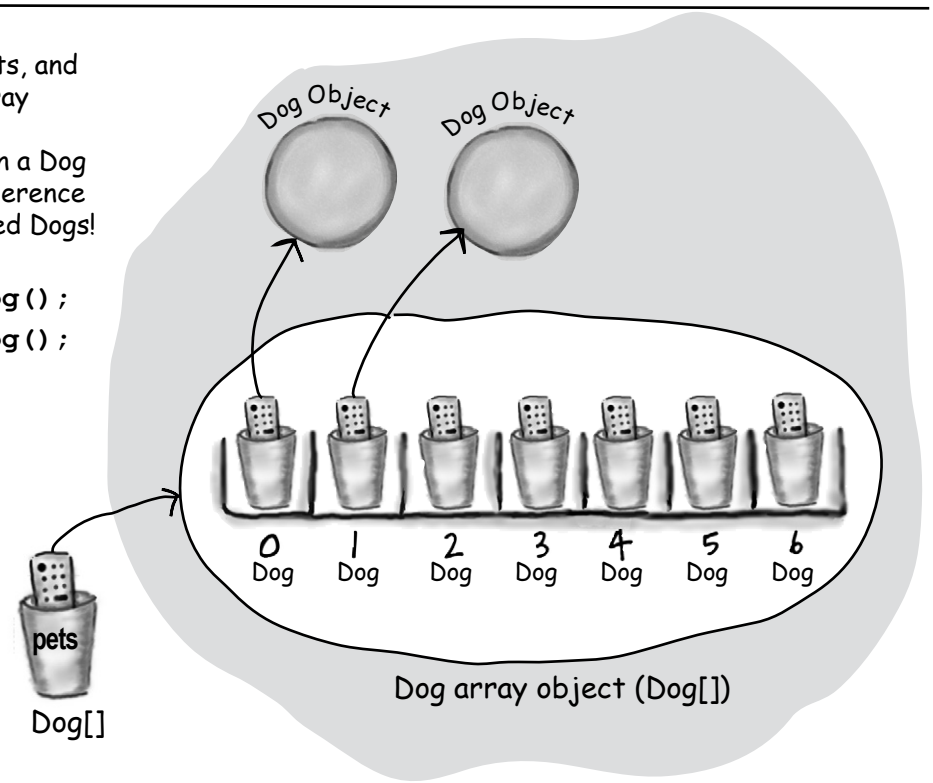
What's missing?

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();
```

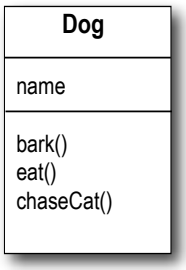


Sharpen your pencil

What is the current value of pets[2]? _____

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

_____



Control your Dog

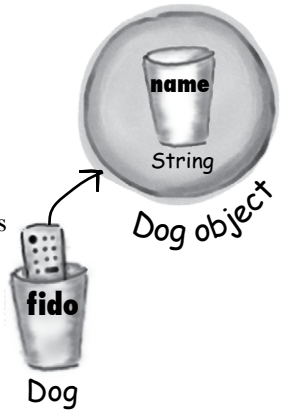
(with a reference variable)

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

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

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

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



What happens if the Dog is in a Dog array?

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

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

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

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

Java cares about type.

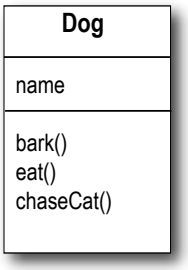
Once you've declared an array, you can't put anything in it except things that are of the declared array type.

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.**

```
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() {  }
}
```

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");
    }
  }
}
```

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  Bikini

% java TestArrays
island = Fiji
island = Cozumel
island = Bermuda
island = Azores
```

```
class TestArrays {

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

Pool Puzzle

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

Output

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

Bonus Question!

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

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

```
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;
  }
}
```

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

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

puzzle: Heap o' Trouble

A Heap o' Trouble

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

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

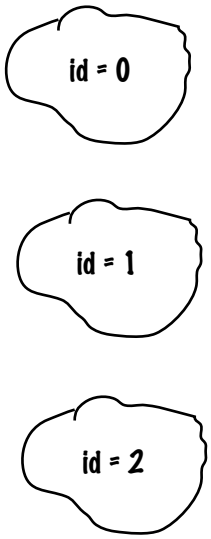
```
class HeapQuiz {
  int id = 0;
  public static void main(String [] args) {
    int x = 0;
    HeapQuiz [ ] hq = new HeapQuiz[5];
    while ( x < 3 ) {
      hq[x] = new HeapQuiz();
      hq[x].id = x;
      x = x + 1;
    }
    hq[3] = hq[1];
    hq[4] = hq[1];
    hq[3] = null;
    hq[4] = hq[0];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

match each reference variable with matching object(s)

You might not have to use every reference.

Reference Variables:

HeapQuiz Objects:



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 as tight as Tawny's top, 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 boys, 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 wearing her short Aloha dress. "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 [] ca = new Contact[10];
while ( x < 10 ) {   // make 10 contact objects
  ca[x] = new Contact();
  x = x + 1;
}
// do complicated Contact list updating stuff with ca
```

"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. Kent was next, already imagining coconut cocktails with Tawny, "Bob," he said, "your solution's a bit kludgy don't you think?" Kent smirked, "Take a look at this baby":

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

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

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

Exercise Solutions

Code Magnets:

```
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  Bikini

% java TestArrays
island = Fiji
island = Cozumel
island = Bermuda
island = Azores
```

A

```
class Books {
  String title;
  String author;
}
class BooksTestDrive {
  public static void main(String [] args) {
    Books [] myBooks = new Books[3];
    int x = 0;
    myBooks[0] = new Books();
    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;
    }
  }
}
```

Remember: We have to actually make the Books objects !

B

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

Remember: arrays start with element 0 !

Puzzle Solutions

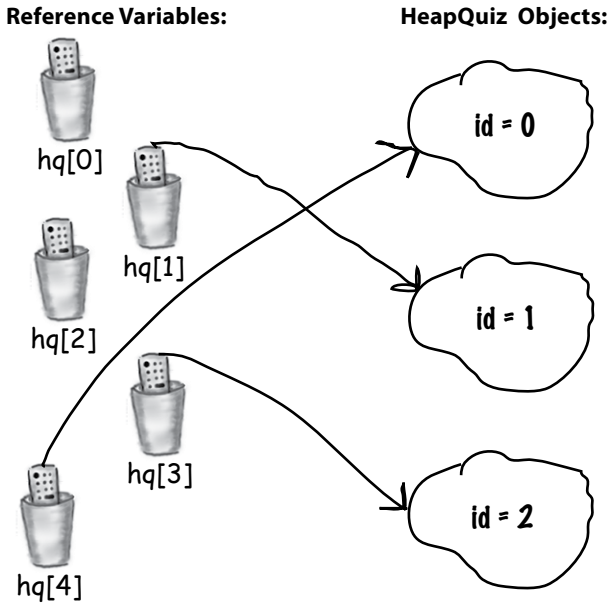
```
class Triangle {
  double area;
  int height;
  int length;
  public static void main(String [] args) {
    int x = 0;
    Triangle [ ] ta = new Triangle[4];
    while ( x < 4 ) {
      ta[x] = new Triangle();
      ta[x].height = (x + 1) * 2;
      ta[x].length = x + 4;
      ta[x].setArea();
      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
```

The case of the pilfered references

Tawny could see that Kent's method had a serious flaw. It's true that he 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 his method created. With each trip through the loop, he 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, Kent'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.)



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?

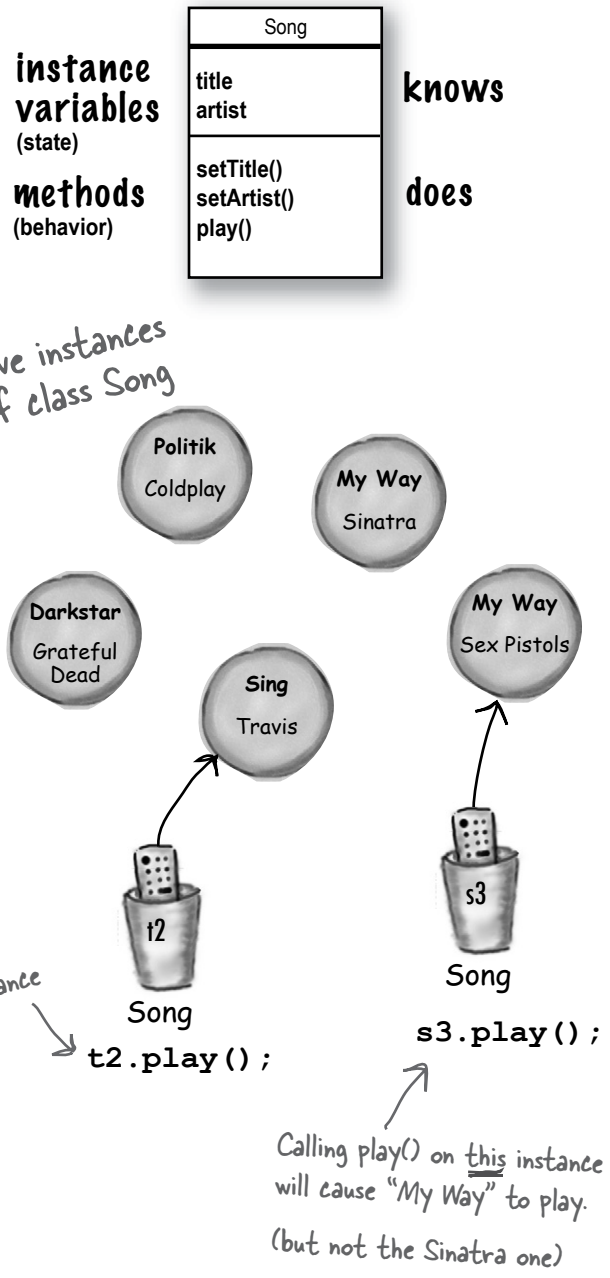
Can every object of that type have different method behavior?

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*. The play() method plays a song, but the instance you call play() on will play the song represented by the value of the *title* instance variable for that instance. So, if you call the play() method on one instance you'll hear the song "Politik", while another instance plays "Darkstar". The method code, however, is the same.

```
void play() {
    soundPlayer.playSound(title);
}
```



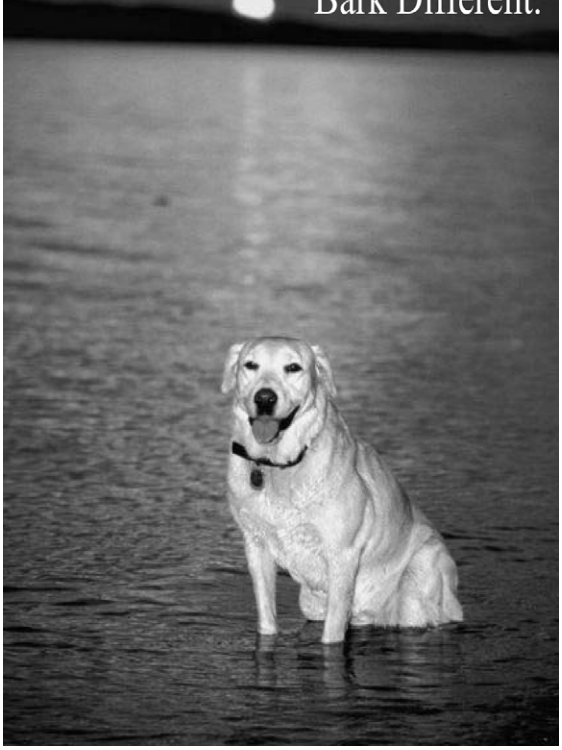

```
Song t2 = new Song();
t2.setArtist("Travis");
t2.setTitle("Sing");
Song s3 = new Song();
s3.setArtist("Sex Pistols");
s3.setTitle("My Way");
```

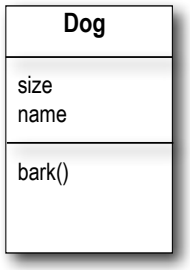
*Yes, another stunningly clear answer!

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:

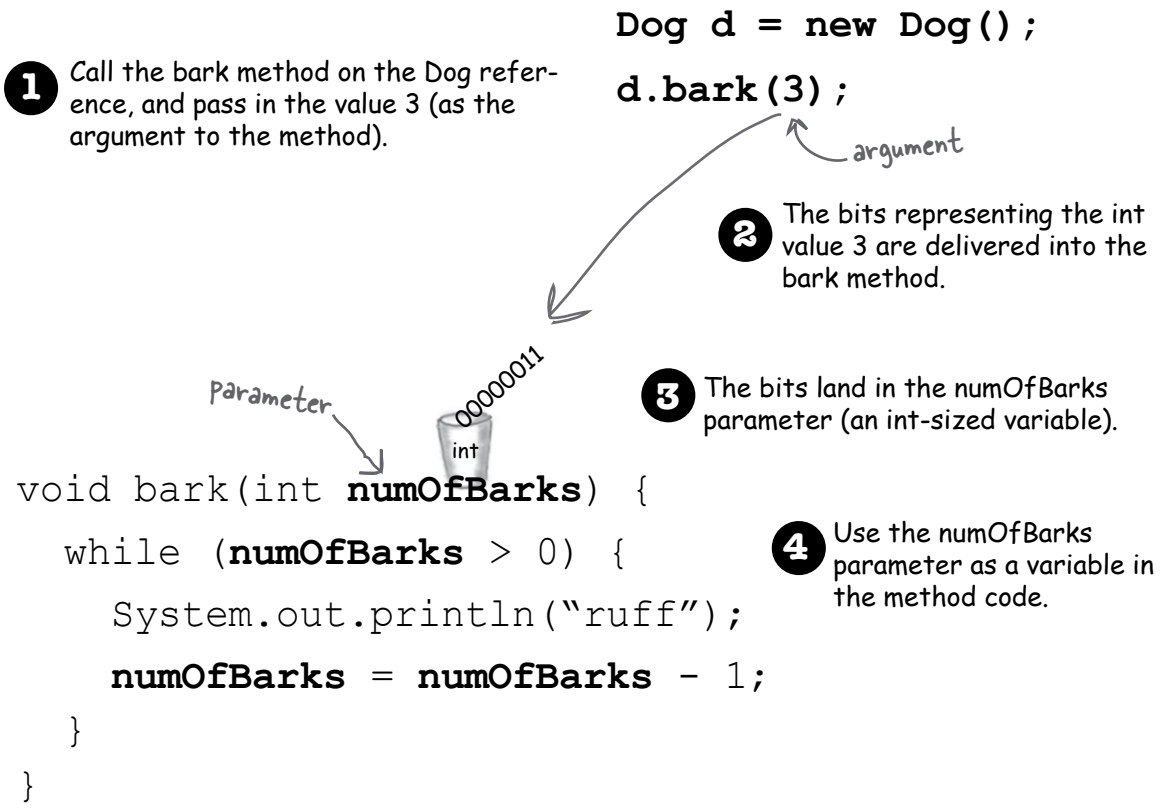
```
d.bark(3);
```

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 method uses parameters. A caller passes arguments.

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.** And that something must be a value of the appropriate type.



You can get things *back* from a method.

Methods can 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() {
}
```

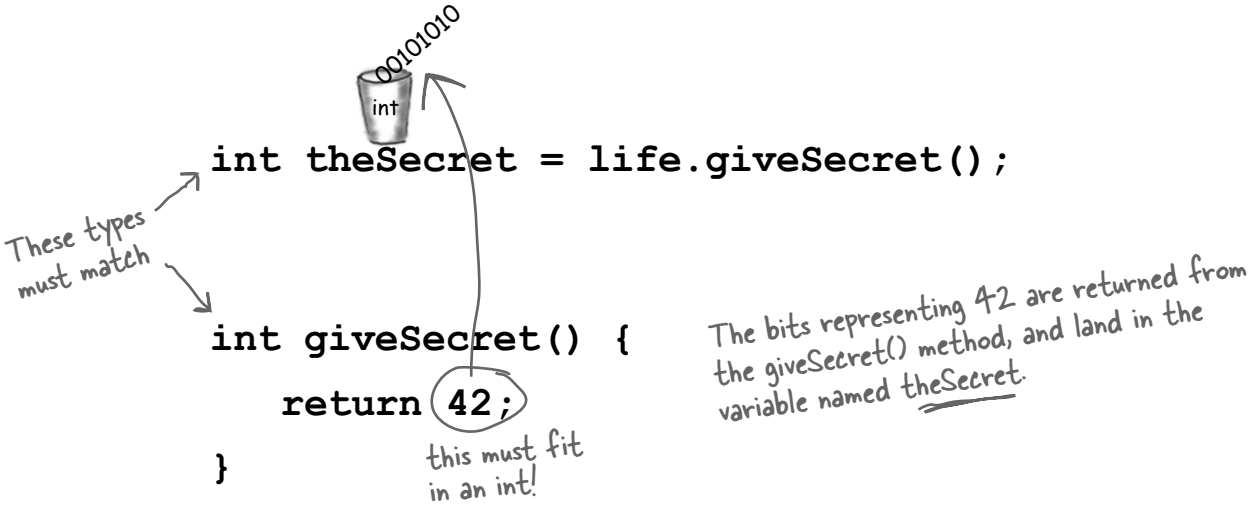
But we can declare a method to give a specific type of value back to the caller, such as:

```
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 chapter 7 and chapter 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.

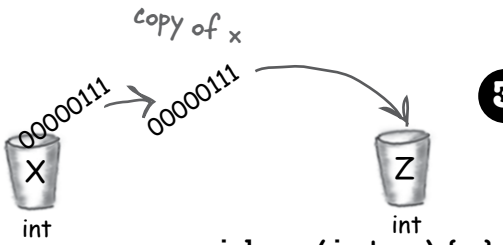
That means pass-by-copy.

```
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){ }
```

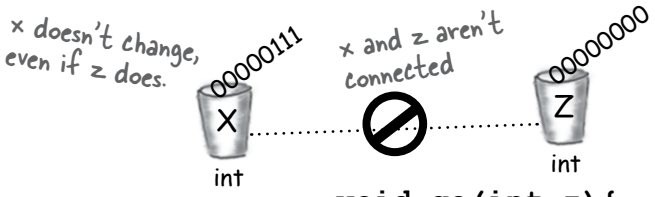
2 Declare a method with an int parameter named z.




```
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?

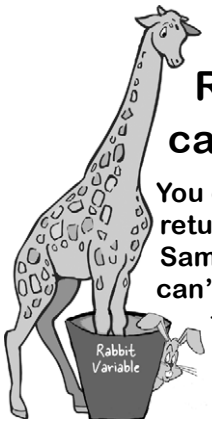
A: Sort of. A method can declare only one return value. BUT... if you want to return, say, three int values, then the declared return type can be an int *array*. Stuff those ints into the array, and pass it on back. It's a little more involved to return multiple values with different types; we'll be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

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.

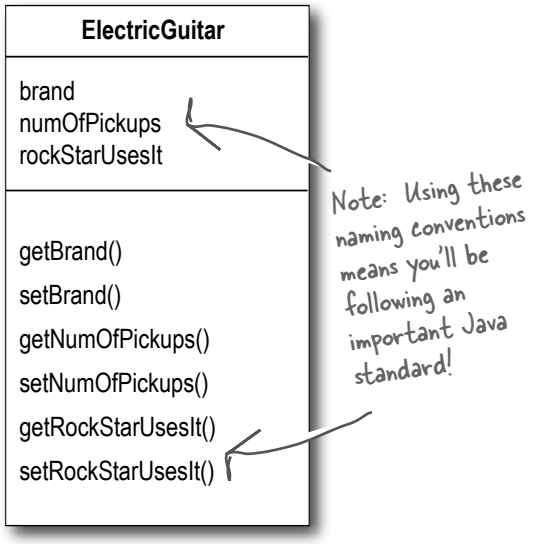
BULLET POINTS

- 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: **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 the 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!

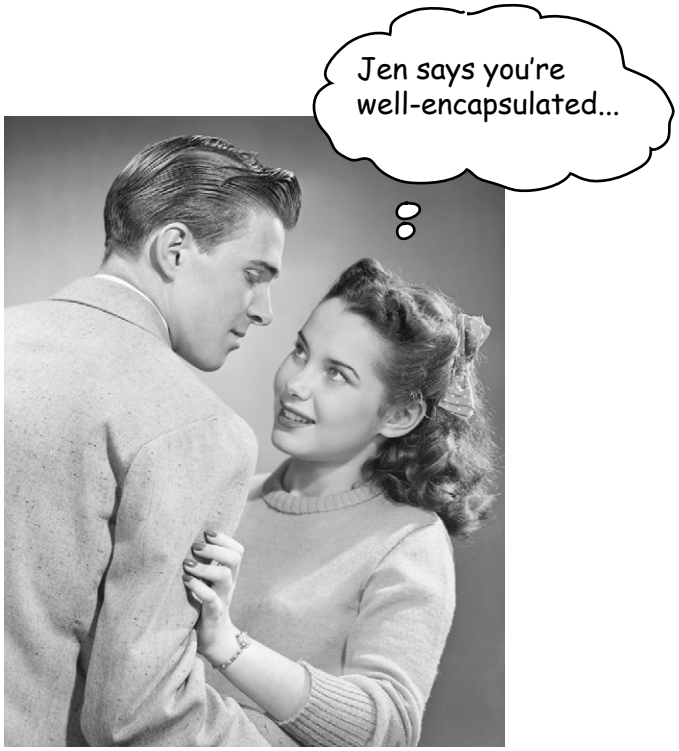
Here we are, just humming along without a care in the world leaving our data out there for *anyone* to see and even touch.

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;
```

yikes! 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: Doesn't take a PhD here. Most instance variable values are coded with certain assumptions about the boundaries of the values. 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. Cell 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 do-able. 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.

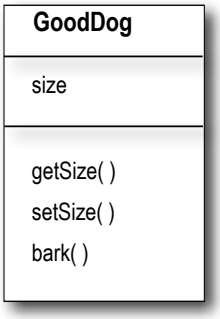
HeadFirst: But sometimes I see setter methods that simply set the value without checking anything. If you have an instance variable that doesn't have a boundary, doesn't that setter method create unnecessary overhead? A performance hit?

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 cool 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:

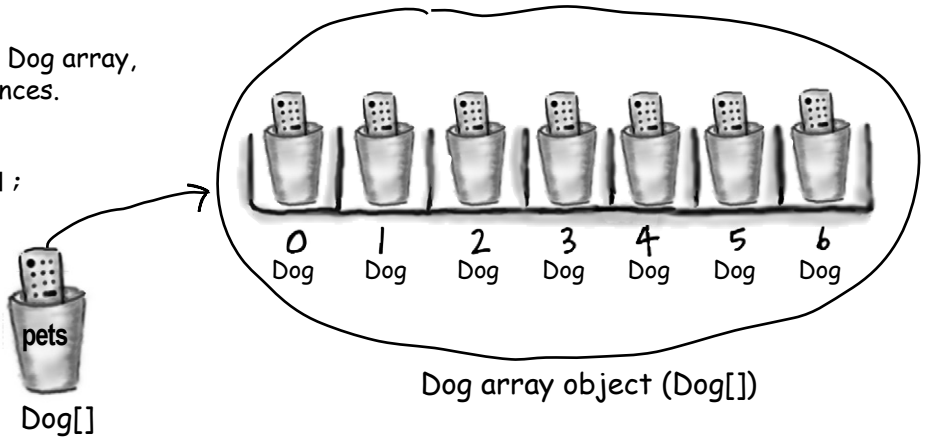
```
int x = 3 + one.getSize();
```

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 7 Dog references.

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



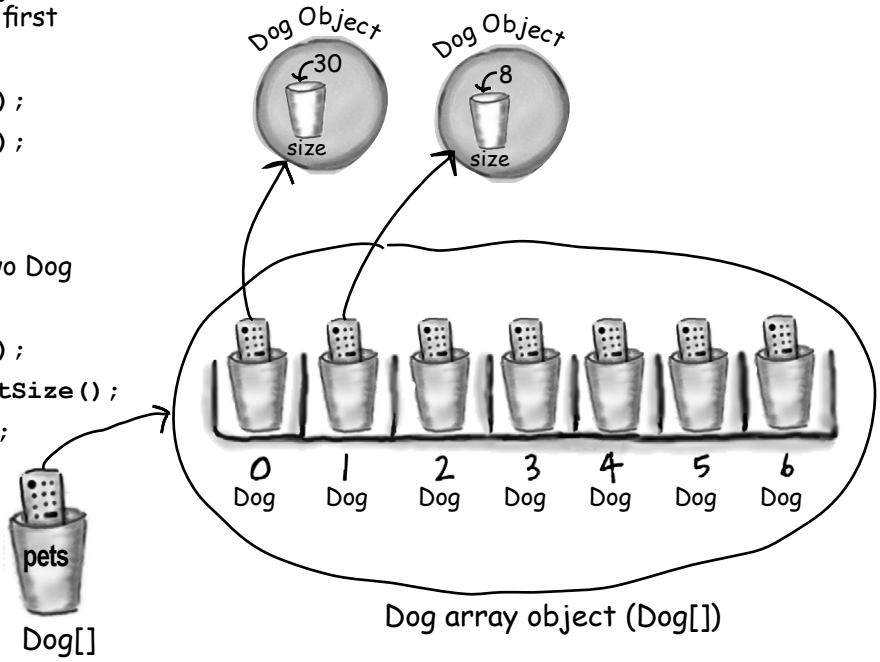
---

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;
   }

}

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());
   }
}
```

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

What will these return??

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;
   int b = 12;

   public int add() {
      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.

But that's because the compiler will give you an error if you try to invoke a method without sending 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 declared for the method, and the arguments are assigned (automatically) to the parameters.

Comparing variables (primitives or references)

Sometimes you want to know if two *primitives* are the same. 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. Easy as well, just use the == operator. But sometimes you want to know if two *objects* are equal. And for that, you need the .equals() method. The idea of equality for objects depends on the type of object. For example, if two different String objects have the same characters (say, "expeditious"), 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 (and appendix B), 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.

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.

(Such as two different String objects that both represent the characters in "Fred")

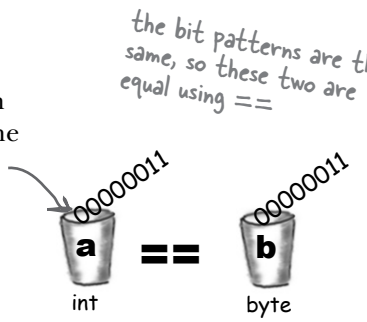
To compare two primitives, use the == operator

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 it doesn't care about the size of the variable, so all the extra zeroes on the left end don't matter).

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

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



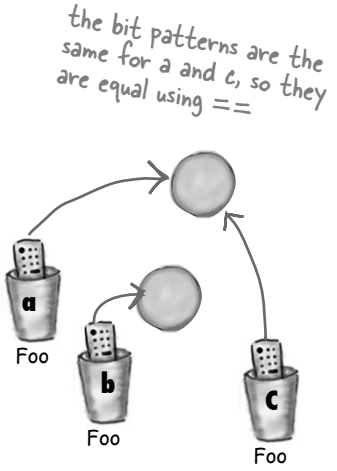
To see if two references are the same (which means they refer to the same object on the heap) use the == operator

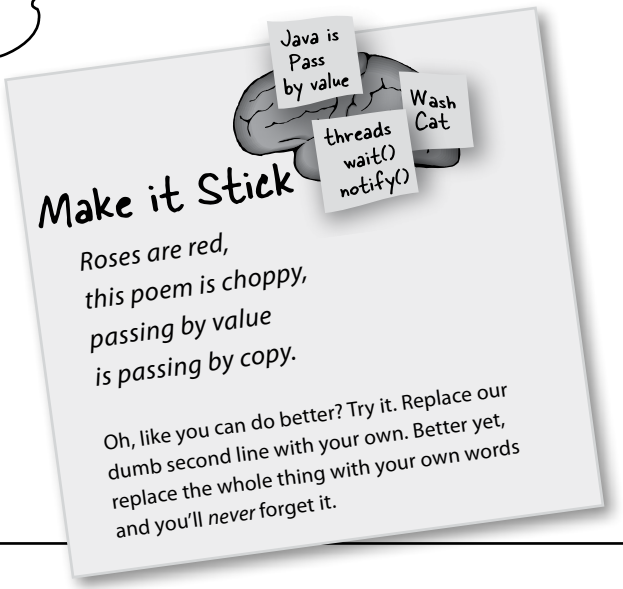
Remember, the == operator cares only about the pattern of bits in the variable. The rules are the same whether the variable is a reference or primitive. So the == operator returns true if two reference variables refer to the same object! In that case, we don't know what the bit pattern is (because it's 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 }
```

a == c is true

a == b is false





Sharpen your pencil

What's legal?

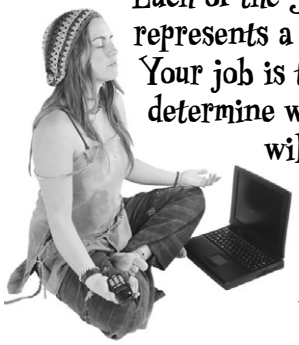
Given the method below, which of the method calls listed on the right are legal?

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);
```

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;
  }
}
```

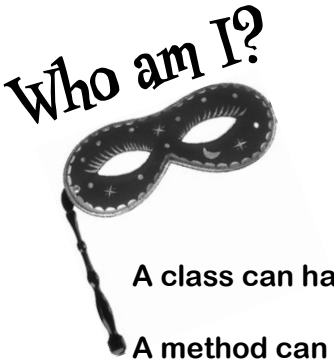
B

```
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);
  }
}
```

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 guy, 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. _____

puzzle: Mixed Messages

A short Java program is listed to your right. Two blocks of the program are missing. Your challenge is to **match the candidate blocks of code** (below), **with the output** that you'd see if the blocks were inserted.

Not all the lines of output will be used, and some of the lines of output might be used more than once. Draw lines connecting the candidate blocks of code with their matching command-line output.

```
public class Mix4 {
  int counter = 0;
  public static void main(String [] args) {
    int count = 0;
    Mix4 [] m4a  =new Mix4[20];
    int x = 0;
    while (          ) {
      m4a[x] = new Mix4();
      m4a[x].counter = m4a[x].counter + 1;
      count = count + 1;
      count = count + m4a[x].maybeNew(x);
      x = x + 1;
    }
    System.out.println(count + " "
                       + m4a[1].counter);
  }

  public int maybeNew(int index) {
    if (              ) {
      Mix4 m4 = new Mix4();
      m4.counter = m4.counter + 1;
      return 1;
    }
    return 0;
  }
}
```

Candidates:

```
x < 9

index < 5
```

```
x < 20

index < 5
```

```
x < 7

index < 7
```

```
x < 19

index < 1
```

Possible output:

```
14 7
```

```
9 5
```

```
19 1
```

```
14 1
```

```
25 1
```

```
7 7
```

```
20 1
```

```
20 5
```

Pool Puzzle

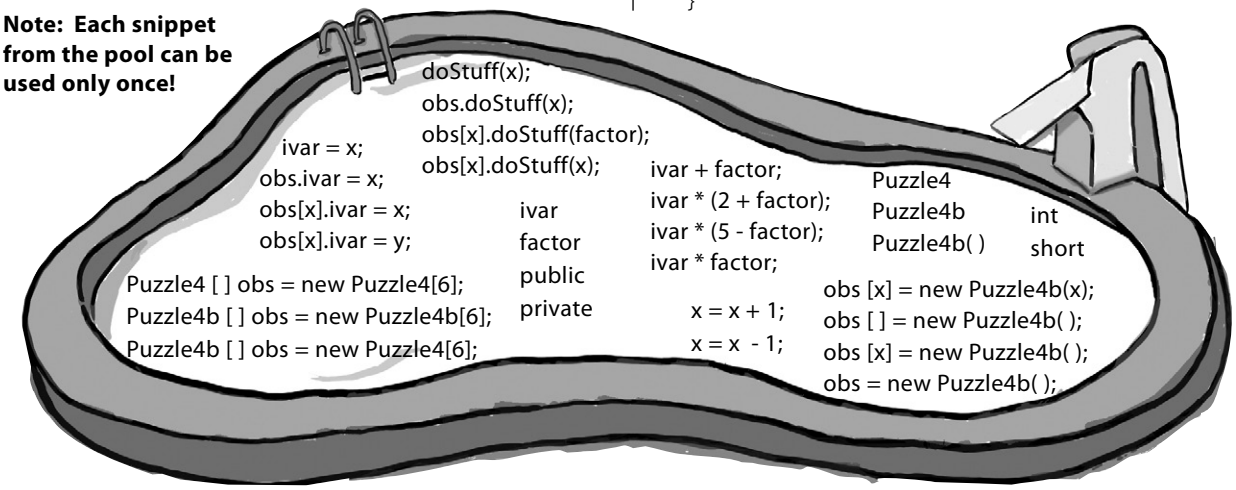
Your ***job*** is to take code snippets from the pool and place them into the blank lines in the code. You may **not** use the same snippet more than once, and you won't need to use all the snippets. Your ***goal*** is to make a class that will compile and run and produce the output listed.

Output

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

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

    _____________________________________

    int y = 1;
    int x = 0;
    int result = 0;
    while (x < 6) {

      _____________________________

      _____________________________

      y = y * 10;

      ___________________

    }
    x = 6;
    while (x > 0) {

      ___________________

      result = result + _____________________

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

class ____________ {
  int ivar;

  _________  ______ doStuff(int __________) {

    if (ivar > 100) {

      return _____________________________

    } else {

      return _____________________________

    }
  }
}
```

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



doStuff(x);
obs.doStuff(x);
obs[x].doStuff(factor);
obs[x].doStuff(x);

ivar = x;
obs.ivar = x;
obs[x].ivar = x;
obs[x].ivar = y;

ivar
factor
public
private

ivar + factor;
ivar * (2 + factor);
ivar * (5 - factor);
ivar * factor;

Puzzle4
Puzzle4b
Puzzle4b( )

int
short

Puzzle4 [ ] obs = new Puzzle4[6];
Puzzle4b [ ] obs = new Puzzle4b[6];
Puzzle4b [ ] obs = new Puzzle4[6];

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

obs [x] = new Puzzle4b(x);
obs [ ] = new Puzzle4b( );
obs [x] = new Puzzle4b( );
obs = new Puzzle4b( );

Fast Times in Stim-City

When Buchanan jammed his twitch-gun into Jai's side, 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 J37NE", 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 J37NE 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 reached for his twitch-gun but Leveler's stunner was already on Buchanan's neck, "Let it go Buchanan", sneered Leveler, "Drop the twitcher 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?

Exercise Solutions

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), 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.

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

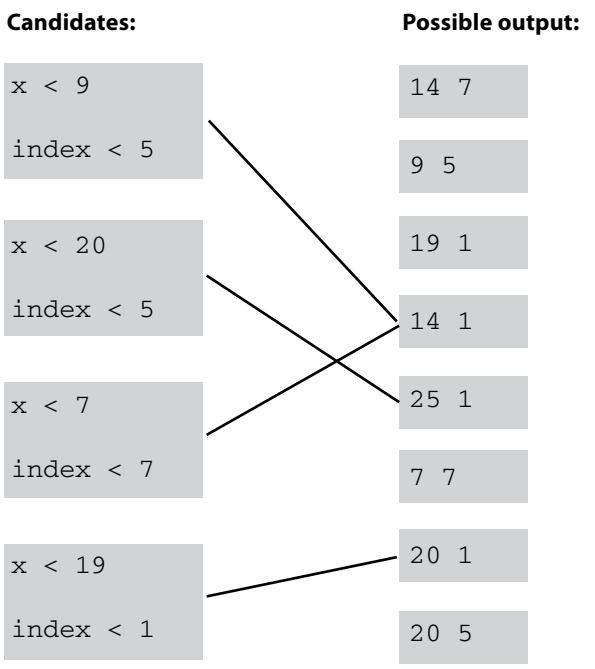
```
public class Puzzle4 {
  public static void main(String [] args) {
    Puzzle4b [ ] obs = new Puzzle4b[6];
    int y = 1;
    int x = 0;
    int result = 0;
    while (x < 6) {
      obs[x] = new Puzzle4b( );
      obs[x] . ivar = y;
      y = y * 10;
      x = x + 1;
    }
    x = 6;
    while (x > 0) {
      x = x - 1;
      result = result + obs[x].doStuff(x);
    }
    System.out.println("result " + result);
  }
}
class Puzzle4b {
  int ivar;
  public int doStuff(int factor) {
    if (ivar > 100) {
      return ivar * factor;
    } else {
      return ivar * (5 - factor);
    }
  }
}
```

Output

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

Answer to the 5-minute mystery...

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.



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**. And **turn a String into an int**, yeah, that would be cool. 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.

Let's build a Battleship-style game: "Sink a Dot Com"

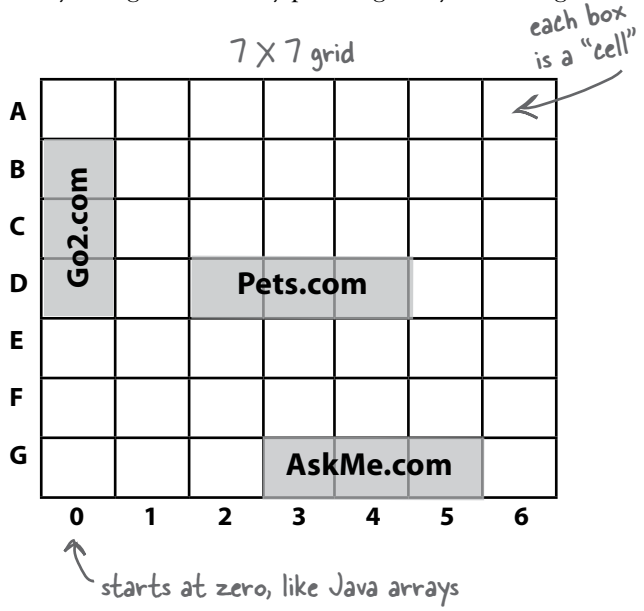
It's you against the computer, but unlike the real Battleship game, in this one you don't place any ships of your own. Instead, your job is to sink the computer's ships in the fewest number of guesses.

Oh, and we aren't sinking ships. We're killing Dot Coms. (Thus establishing business relevancy so you can expense the cost of this book).

Goal: Sink all of the computer's Dot Coms 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 Dot Coms 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 Pets.com" (or whatever the lucky Dot Com of the day is). When you've sent all three Dot Coms to that big 404 in the sky, the game ends by printing out your rating.



You're going to build the Sink a Dot Com game, with a 7 x 7 grid and three Dot Coms. Each Dot Com takes up three cells.

part of a game interaction

```
File Edit Window Help Sell
%java DotComBust
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 Pets.com   : (
kill
Enter a guess  B4
miss
Enter a guess  G3
hit
Enter a guess  G4
hit
Enter a guess  G5
Ouch! You sunk AskMe.com   : (
```

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 Dot Coms
- **B** Game places the three Dot Coms onto a virtual grid

2 Game play begins

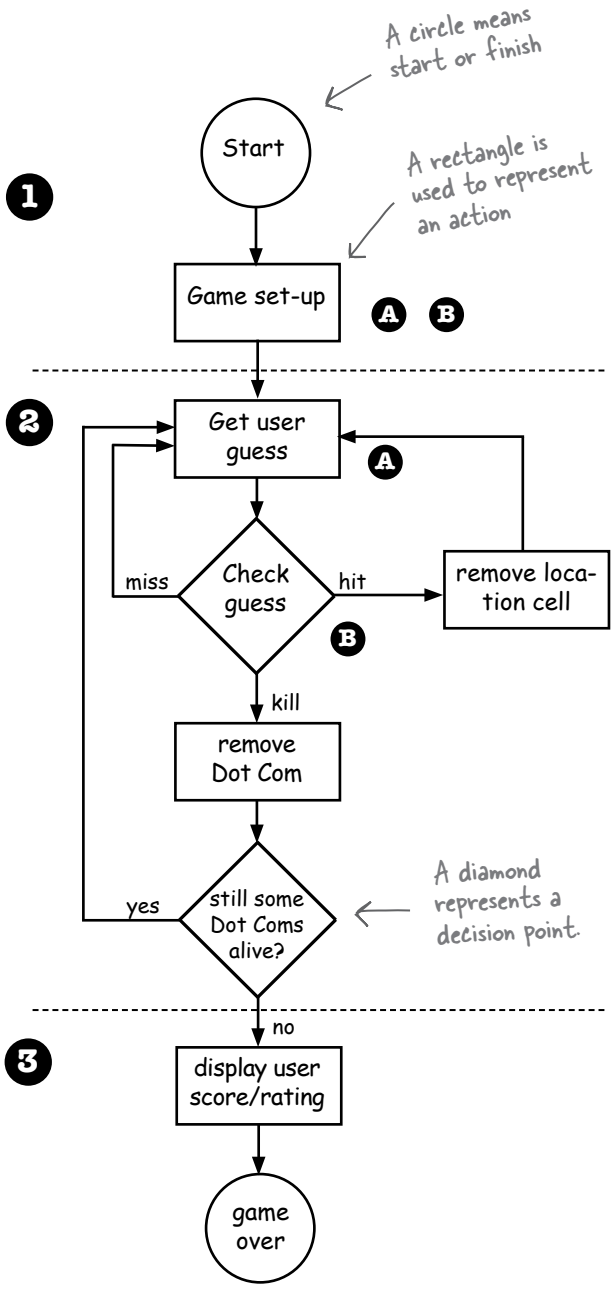
Repeat the following until there are no more Dot Coms:

- **A** Prompt user for a guess ("A2", "C0", etc.)
- **B** Check the user guess against all Dot Coms to look for a hit, miss, or kill. Take appropriate action: if a hit, delete cell (A2, D4, etc.). If a kill, delete Dot Com.

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 Larry; focus first on the ***things*** in the program rather than the ***procedures***.



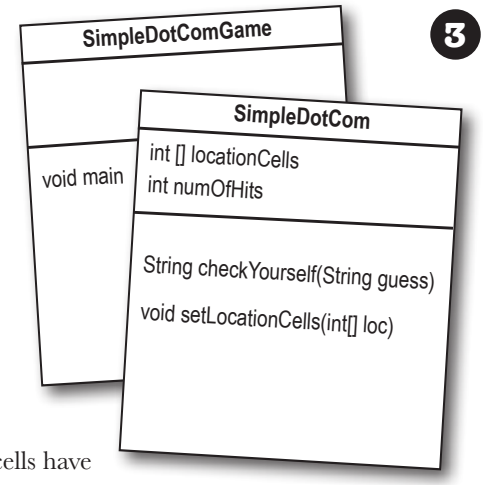
Whoa. A real flow chart.

The "Simple Dot Com Game" a gentler introduction

It looks like we're gonna need at least two classes, a Game class and a DotCom class. But before we build the full monty ***Sink a Dot Com*** game, we'll start with a stripped-down, simplified version, ***Simple Dot Com 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 Dot Com in just a single *row.* And instead of *three* Dot Coms, we use *one*.

The goal is the same, though, so the game still needs to make a DotCom instance, assign it a location somewhere in the row, get user input, and when all of the DotCom'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 DotCom 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 DotCom 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 DotCom 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 DotCom occupies.

1. **Game starts**, and creates ONE DotCom 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 DotCom'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 tells the user how many guesses it took to sink the DotCom.

A complete game interaction

```
File Edit Window Help Destroy
%java SimpleDotComGame
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:

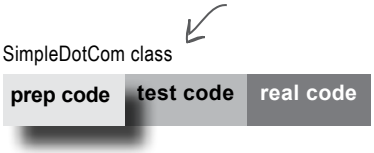
- ☐ Figure out what the class is supposed to *do*.
- ☐ List the **instance variables and methods**.
- ☐ Write **prepcode** for the methods. (You'll see this in just a moment.)
- ☐ Write **test code** for the methods.
- ☐ **Implement** the class.
- ☐ **Test** the methods.
- ☐ **Debug** and **reimplement**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



prep code

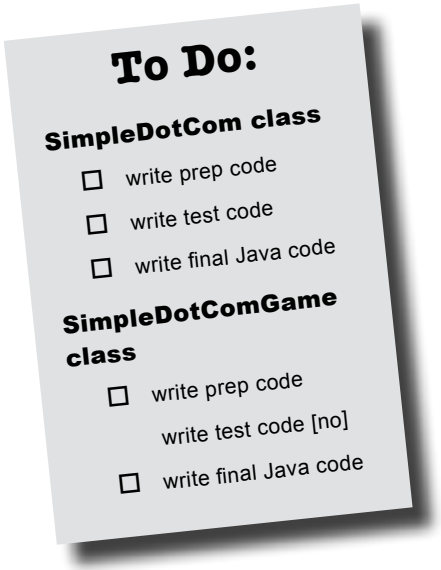
A form of pseudocode, to help you focus on the logic without stressing about syntax.

test code

A class or methods that will test the real code and validate that it's doing the right thing.

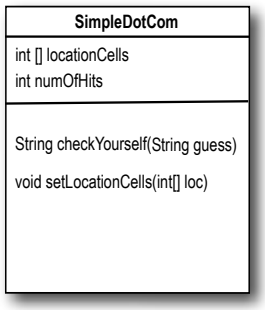
real code

The actual implementation of the class. This is where we write real Java code.



SimpleDotCom class

prep code | test code | real code



You'll get the idea of how prepcode (our version of pseudocode) works as you read through this example. It's sort of half-way between real Java code and a plain English description of the class. Most prepcode includes three parts: instance variable declarations, method declarations, method logic. The most important part of prepcode 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 ***String*** 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(String userGuess)*

GET the user guess as a String parameter

CONVERT the user guess to an ***int***

REPEAT with each of the location cells in the ***int*** array

// COMPARE the user guess to the location cell

IF the user guess matches

INCREMENT the number of hits

// FIND OUT if it was the last location cell:

IF number of hits is 3, **RETURN** "kill" as the result

ELSE it was not a kill, so **RETURN** "hit"

END IF

ELSE the user guess did not match, so **RETURN** "miss"

END IF

END REPEAT

END METHOD

METHOD: *void setLocationCells(int[] cellLocations)*

GET the cell locations as an ***int array*** parameter

ASSIGN the cell locations parameter to the cell locations instance variable

END METHOD

prep code | test code | real code

Writing the method implementations

let's write the real method code now, and get this puppy working.

Before we start coding the methods, though, let's back up and write some code to *test* the methods. That's right, we're writing the test code *before* there's anything to test!

The concept of writing the test code first is one of the practices of Extreme Programming (XP), and it can make it easier (and faster) for you to write your code. We're not necessarily saying you should use XP, but we do like the part about writing tests first. And XP just *sounds* cool.

Extreme Programming (XP)

Extreme Programming(XP) is a newcomer to the software development methodology world. Considered by many to be "the way programmers really want to work", XP emerged in the late 90's and has been adopted by companies ranging from the two-person garage shop to the Ford Motor Company. The thrust of XP is that the customer gets what he wants, when he wants it, even when the spec changes late in the game.

XP is based on a set of proven practices that are all designed to work together, although many folks do pick and choose, and adopt only a portion of XP's rules. These practices include things like:

Make small, but frequent, releases.

Develop in iteration cycles.

Don't put in anything that's not in the spec (no matter how tempted you are to put in functionality "for the future").

Write the test code *first*.

No killer schedules; work regular hours.

Refactor (improve the code) whenever and wherever you notice the opportunity.

Don't release anything until it passes all the tests.

Set realistic schedules, based around small releases.

Keep it simple.

Program in pairs, and move people around so that everybody knows pretty much everything about the code.

Writing test code for the SimpleDotCom class

We need to write test code that can make a SimpleDotCom object and run its methods. For the SimpleDotCom 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 prepcode 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 prepcode:

```
METHOD String checkYourself(String userGuess)
  GET the user guess as a String parameter
  CONVERT the user guess to an int
  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 SimpleDotCom object.
2. Assign it a location (an array of 3 ints, like {2,3,4}).
3. Create a String 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, so that you always prove that your latest code additions don't break previously-tested code.

Test code for the SimpleDotCom class

```
public class SimpleDotComTestDrive {

   public static void main (String[] args) {

      SimpleDotCom dot = new SimpleDotCom();

      int[] locations = {2,3,4};

      dot.setLocationCells(locations);

      String userGuess = "2";

      String result = dot.checkYourself(userGuess);

      String testResult = "failed";

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

           testResult = "passed";

       }

       System.out.println(testResult);

    }

}
```

instantiate a SimpleDotCom object

make an int array for the location of the dot com (3 consecutive ints out of a possible 7).

invoke the setter method on the dot com.

make a fake user guess

invoke the checkYourself() method on the dot com 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

In the next couple of pages we implement the SimpleDotCom 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:

SimpleDotCom class

prep code | test code | real code

The checkYourself() method

There isn't a perfect mapping from prepcode to javacode; you'll see a few adjustments. The prepcode gave us a much better idea of *what* the code needs to do, and now we have to find 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

CONVERT the user guess to an *int*

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"

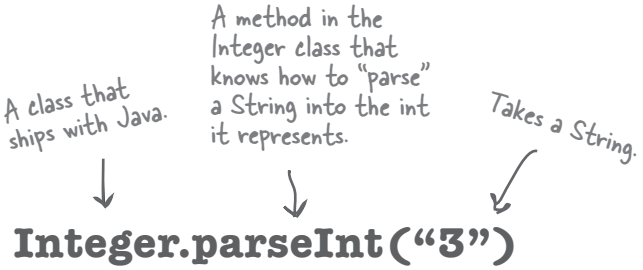
```
public String checkYourself(String stringGuess) {

    int guess = Integer.parseInt(stringGuess);   (1)  <- convert the String to an int

     String result = "miss";   <- make a variable to hold the result we'll return. put "miss" in as the default (i.e. we assume a "miss")

    for (int cell : locationCells) {   (2)  <- repeat with each cell in the locationCells array (each cell location of the object)

       if (guess == cell) {   <- compare the user guess to this element (cell) in the array

           result = "hit";   <- we got a hit!

           numOfHits++;   (3)

           break;   (4)  <- get out of the loop, no need to test the other cells

       } // end if

    } // end for

    if (numOfHits == locationCells.length) {

       result = "kill";   <- we're out of the loop, but let's see if we're now 'dead' (hit 3 times) and change the result String to "Kill"

    } // end if

    System.out.println(result);   <- display the result for the user ("Miss", unless it was changed to "Hit" or "Kill")

    return result;

} // end method   <- return the result back to the calling method
```

prep code | test code | real code

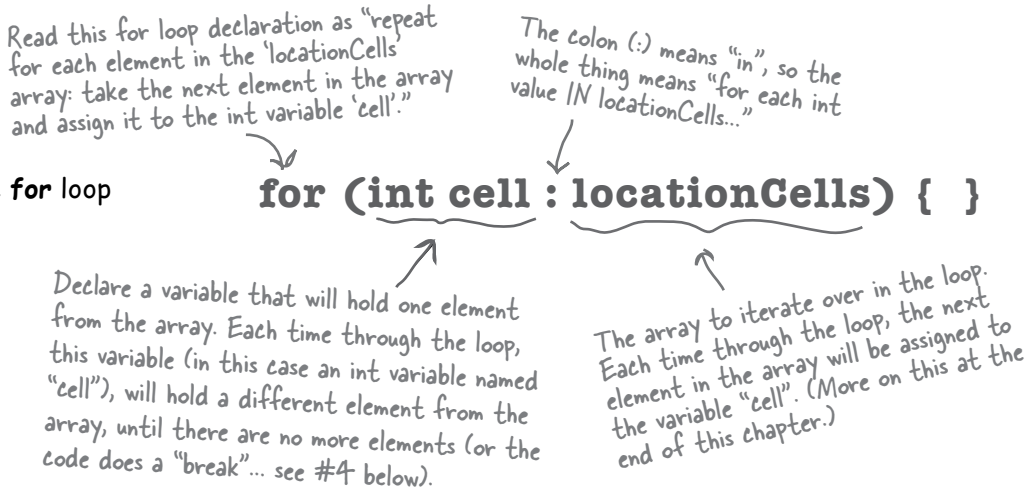
Just the new stuff

The things we haven't seen before are on this page. Stop worrying! The rest of the details are at the end of the chapter. This is just enough to let you keep going.

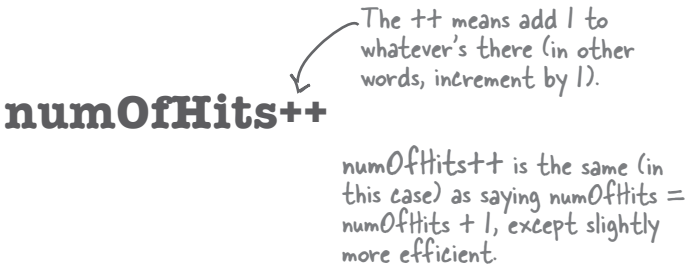
1 Converting a String to an int



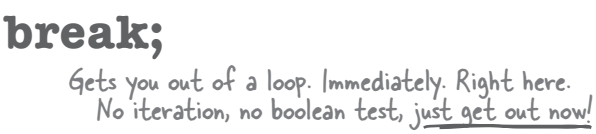
2 The for loop



3 The post-increment operator



4 break statement



there are no Dumb Questions

Q: What happens in Integer.parseInt() if the thing you pass isn't a number? And does it recognize spelled-out numbers, like "three"?

A: Integer.parseInt() works only on Strings that represent the ascii values for digits (0,1,2,3,4,5,6,7,8,9). If you try to parse something like "two" or "blurp", the code will blow up at runtime. (By *blow up*, we actually mean *throw an exception*, but we don't talk about exceptions until the Exceptions chapter. So for now, *blow up* is close enough.)

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... beginning with Java 5.0 (Tiger), 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.

Final code for SimpleDotCom and SimpleDotComTester

```
public class SimpleDotComTestDrive {

   public static void main (String[] args) {
      SimpleDotCom dot = new SimpleDotCom();
      int[] locations = {2,3,4};
      dot.setLocationCells(locations);
      String userGuess = "2";
      String result = dot.checkYourself(userGuess);
    }
}
```

```
public class SimpleDotCom {

   int[] locationCells;
   int numOfHits = 0;

   public void setLocationCells(int[] locs) {
      locationCells = locs;
   }

   public String checkYourself(String stringGuess) {
      int guess = Integer.parseInt(stringGuess);
      String result = "miss";
      for (int cell : locationCells) {
         if (guess == cell) {
            result = "hit";
            numOfHits++;
            break;
         }
      } // out of the loop

      if (numOfHits ==
          locationCells.length) {
          result = "kill";
      }
      System.out.println(result);
         return result;
    } // close method
} // close class
```

There's a little bug lurking here. It compiles and runs, but sometimes... 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 SimpleDotCom 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 SimpleDotComTestDrive
hit
passed
```

prep code | test code | real code

Sharpen your pencil

We built the test class, and the SimpleDotCom 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 prepcode 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 dot com is still alive :

GET user input from the command line

The SimpleDotComGame needs to do this:

1. Make the single SimpleDotCom 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 dot com is dead .
6. Tell the user how many guesses it took.

A complete game interaction

```
File Edit Window Help Runaway
%java SimpleDotComGame
enter a number  2
hit
enter a number  3
hit
enter a number  4
miss
enter a number  1
kill
You took 4 guesses
```

Prepcode for the SimpleDotComGame class

Everything happens in main()

There are some things you'll have to take on faith. For example, we have one line of prepcode 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 prepcode, 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*, set it to 0.

MAKE a new SimpleDotCom 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 SimpleDotCom instance

DECLARE a boolean variable representing the state of the game, named *isAlive*. **SET** it to true

WHILE the dot com is still alive (isAlive == true) :

GET user input from the command line

// CHECK the user guess

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

INCREMENT *numOfGuesses* variable

// CHECK for dot com death

IF result is "kill"

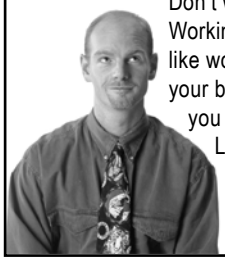
SET *isAlive* to false (which means we won't enter the loop again)

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

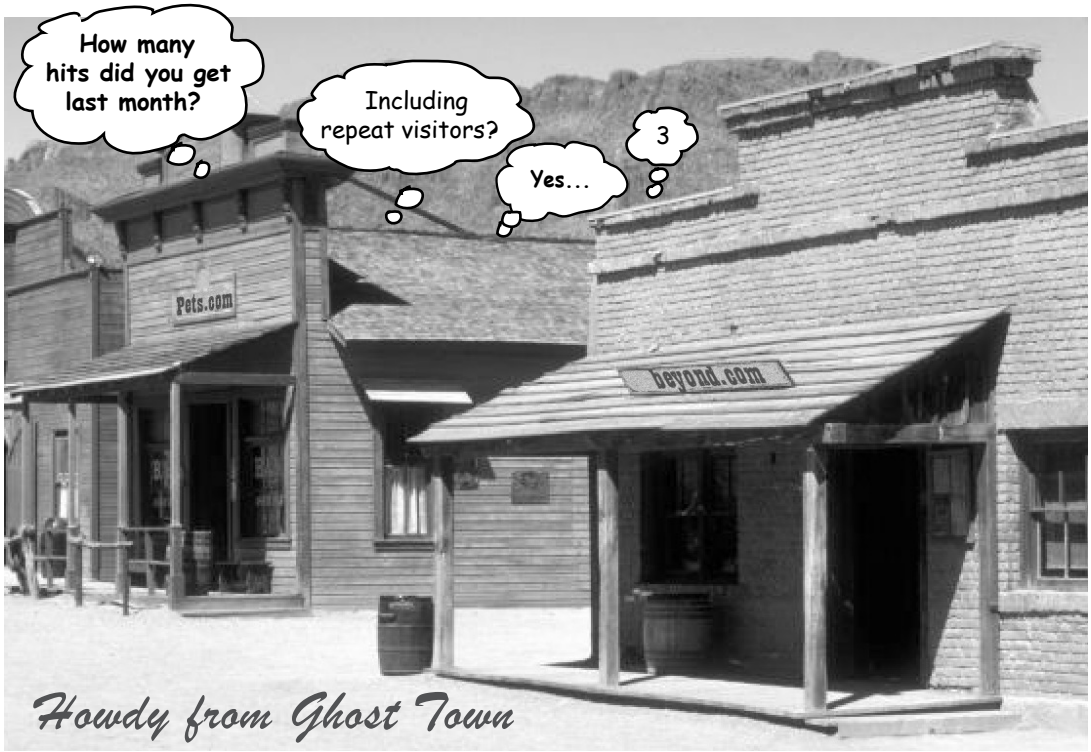
- Your Java program should start with a high-level design.
- Typically you'll write three things when you create a new class:

  prepcode

  testcode

  real (Java) code

- Prepcode should describe *what* to do, not *how* to do it. Implementation comes later.
- Use the prepcode to help design the test code.
- 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.
- Use the pre/post *increment* operator to add 1 to a variable (x++;)
- Use the pre/post *decrement* to subtract 1 from a variable (x--;)
- Use `Integer.parseInt()` to get the int value of a String.
- `Integer.parseInt()` works only if the String represents a digit ("0","1","2", etc.)
- Use *break* to leave a loop early (i.e. even if the boolean test condition is still true).



SimpleDotComGame class

prep code | test code | real code

The game's main() method

Just as you did with the SimpleDotCom 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.

```
public static void main(String[] args) {
    int numOfGuesses = 0;
    GameHelper helper = new GameHelper();
    SimpleDotCom theDotCom = new SimpleDotCom();
    int randomNum = (int) (Math.random() * 5);
    int[] locations = {randomNum, randomNum+1, randomNum+2};
    theDotCom.setLocationCells(locations);
    boolean isAlive = true;
    while(isAlive == true) {
       String guess = helper.getUserInput("enter a number");
       String result = theDotCom.checkYourself(guess);
       numOfGuesses++;
       if (result.equals("kill")) {
          isAlive = false;
          System.out.println("You took " + numOfGuesses + " guesses");
       } // close if
    } // close while
} // close main
```

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 the dot com object

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

give the dot com its locations (the array)

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

get user input String

ask the dot com to check the guess; save the returned result in a String

increment guess count

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

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

MAKE a SimpleDotCom object

COMPUTE a random number between 0 and 4

MAKE an int array with the 3 cell locations, and

INVOKE setLocationCells on the dot com object

DECLARE a boolean isAlive

WHILE the dot com is still alive

GET user input

// CHECK it

INVOKE checkYourself() on dot com

INCREMENT numOfGuesses

IF result is "kill"

SET gameAlive 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 parens). Math.random returns a double, so we have to cast it to be an int (we want a nice whole number between 0 and 4). In this case, the cast lops 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 method of the Math class.

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.

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

We declare a String variable to hold the user input String 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 a String.

GameHelper class (Ready-bake)

prep code | test code | real code

One last class: GameHelper

We made the *dot com* 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 way too many topics best left for later. (Later, as in chapter 14.)

Just copy* the code below and compile it into a class named GameHelper. Drop all three classes (SimpleDotCom, SimpleDotComGame, GameHelper) into the same directory, and make it your working directory.

Whenever you see the Ready-bake Code 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

```
import java.io.*;
public class GameHelper {
   public String getUserInput(String prompt) {
      String inputLine = null;
      System.out.print(prompt + "  ");
      try {
        BufferedReader is = new BufferedReader(
        new InputStreamReader(System.in));
       inputLine = is.readLine();
       if (inputLine.length() == 0 )  return null;
      } catch (IOException e) {
       System.out.println("IOException: " + e);
      }
      return inputLine;
   }
}
```

*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 wickedlysmart.com.

Let's play

Here's what happens when we run it and enter the numbers 1,2,3,4,5,6. Lookin' good.

A complete game interaction
(your mileage may vary)

```
File  Edit  Window  Help  Smile
%java SimpleDotComGame
enter a number  1
miss
enter a number  2
miss
enter a number  3
miss
enter a number  4
hit
enter a number  5
hit
enter a number  6
kill
You took 6 guesses
```

What's this? A *bug*?

Gasp!

Here's what happens when we enter 1,1,1.

A different game interaction
(yikes)

```
File  Edit  Window  Help  Faint
%java SimpleDotComGame
enter a number  1
hit
enter a number  1
hit
enter a number  1
kill
You took 3 guesses
```

Sharpen your pencil

It's a cliff-hanger!

Will we ***find*** the bug?

Will we ***fix*** the bug?

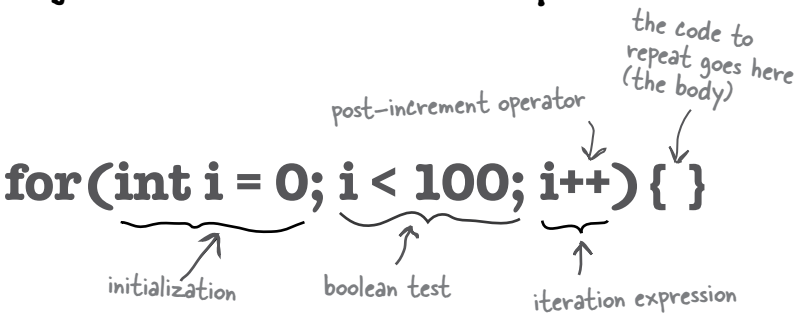
Stay tuned for the next chapter, where we answer these questions and more...

And in the meantime, see if you can come up with ideas for what went wrong and how to fix it.

More about for loops

We've covered all the game code for *this* chapter (but we'll pick it up again to finish the deluxe version of the game in the next chapter). We didn't want to interrupt your work with some of the details and background info, so we put it back here. We'll start with the details of for loops, and if you're a C++ programmer, you can just skim these last few pages...

Regular (non-enhanced) for loops



What it means in plain English: "Repeat 100 times."

How the compiler sees it:

* create a variable *i* and set it to 0.
* repeat while *i* is less than 100.
* at the end of each loop iteration, add 1 to *i*

Part One: *initialization*

Use this part to declare and initialize a variable to use within the loop body. You'll most often use this variable as a counter. You can actually initialize more than one variable here, but we'll get to that later in the book.

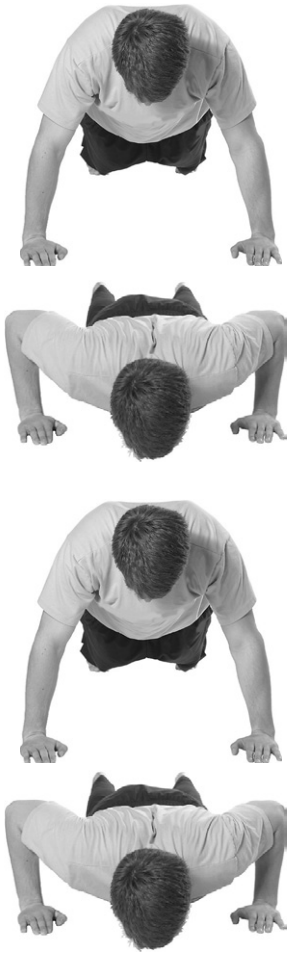
Part Two: *boolean test*

This is where the conditional test goes. Whatever's in there, it *must* resolve to a boolean value (you know, ***true*** or ***false***). You can have a test, like (x >= 4), or you can even invoke a method that returns a boolean.

Part Three: *iteration expression*

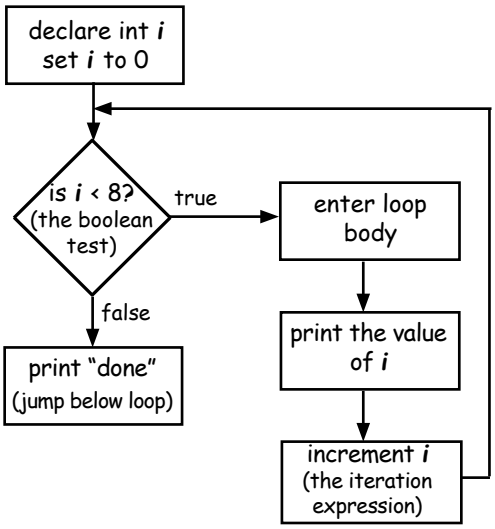
In this part, put one or more things you want to happen with each trip through the loop. Keep in mind that this stuff happens at the *end* of each loop.

repeat for 100 reps:



Trips through a loop

```
for (int i = 0; i < 8; i++) {
   System.out.println(i);
}
System.out.println("done");
```



Difference between for and while

A *while* loop has only the boolean test; it doesn't have a built-in initialization or iteration expression. A *while* loop is good when you don't know how many times to loop and just want to keep going while some condition is true. But if you *know* how many times to loop (e.g. the length of an array, 7 times, etc.), a *for* loop is cleaner. Here's the loop above rewritten using *while:*

```
int i = 0;
while (i < 8) {
   System.out.println(i);
   i++;
}
System.out.println("done");
```

we have to declare and initialize the counter

we have to increment the counter

output:

```
File Edit Window Help Repeat
%java Test
0
1
2
3
4
5
6
7
done
```

++ −−

Pre and Post Increment/Decrement Operator

The shortcut for adding or subtracting 1 from a variable.

```
x++;
```

is the same as:

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

They both mean the same thing in this context:

"add 1 to the current value of x" or "***increment*** x by 1"

And:

```
x--;
```

is the same as:

```
x = x - 1;
```

Of course that's never the whole story. The placement of the operator (either before or after the variable) can affect the result. Putting the operator *before* the variable (for example, ++x), means, "*first*, increment x by 1, and *then* use this new value of x." This only matters when the ++x is part of some larger expression rather than just in a single statement.

```
int x = 0;      int z = ++x;
```

produces: x is 1, z is 1

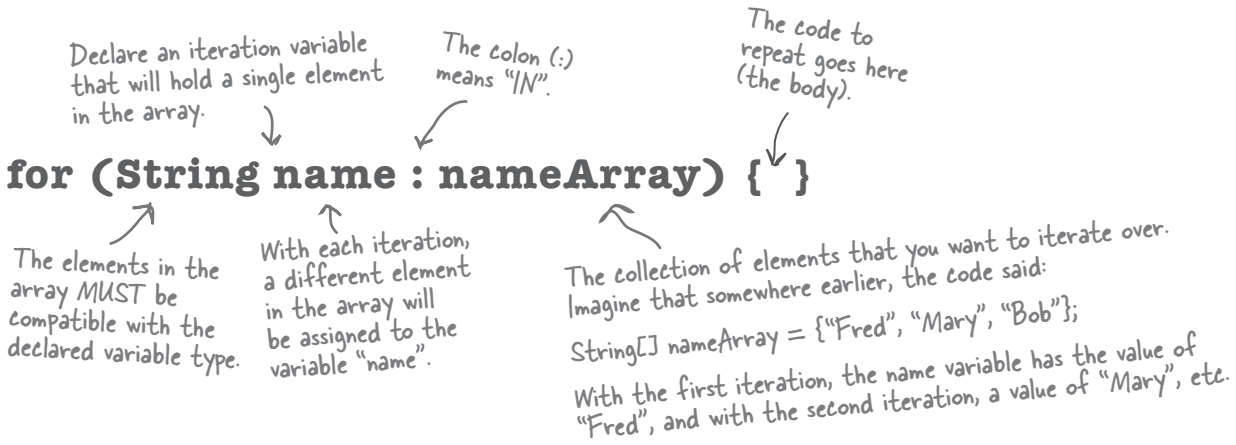
But putting the ++ *after* the x give you a different result:

```
int x = 0;      int z = x++;
```

produces: x is 1, but ***z is 0***! z gets the value of x and *then* x is incremented.

The enhanced for loop

Beginning with Java 5.0 (Tiger), the Java language has a second kind of *for* loop called the *enhanced for*, that makes it easier to iterate over all the elements in an array or other kinds of collections (you'll learn about *other* collections in the next chapter). That's really all that the enhanced for gives you—a simpler way to walk through all the elements in the collection, but since it's the most common use of a *for* loop, it was worth adding it to the language. We'll revisit the *enhanced for loop* in the next chapter, when we talk about collections that *aren't* arrays.



What it means in plain English: "For each element in nameArray, assign the element to the 'name' variable, and run the body of the loop."

How the compiler sees it:

* Create a String variable called *name* and set it to null.
* Assign the first value in *nameArray* to name.
* Run the body of the loop (the code block bounded by curly braces).
* Assign the next value in *nameArray* to name.
* Repeat while *there are still elements in the array.*

Note: depending on the programming language they've used in the past, some people refer to the enhanced for as the "for each" or the "for in" loop, because that's how it reads: "for EACH thing IN the collection..."

Part One: *iteration variable declaration*

Use this part to declare and initialize a variable to use within the loop body. With each iteration of the loop, this variable will hold a different element from the collection. The type of this variable must be compatible with the elements in the array! For example, you can't declare an *int* iteration variable to use with a *String[]* array.

Part Two: *the actual collection*

This must be a reference to an array or other collection. Again, don't worry about the *other* non-array kinds of collections yet—you'll see them in the next chapter.

Converting a String to an int

```
int guess = Integer.parseInt(stringGuess);
```

The user types his guess at the command-line, when the game prompts him. That guess comes in as a String ("2", "0", etc.) , and the game passes that String into the checkYourself() method.

But the cell locations are simply ints in an array, and you can't compare an int to a String.

For example, ***this won't work:***

String num = "2";

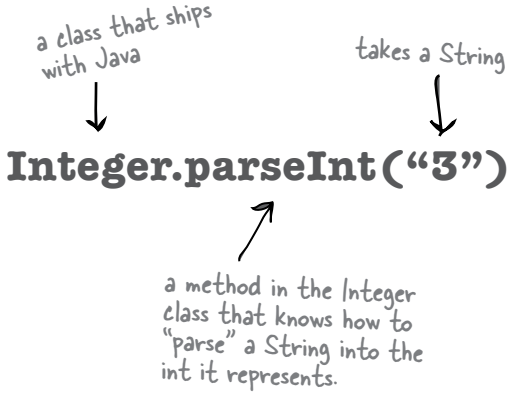
int x = 2;

if (x == num) // horrible explosion!

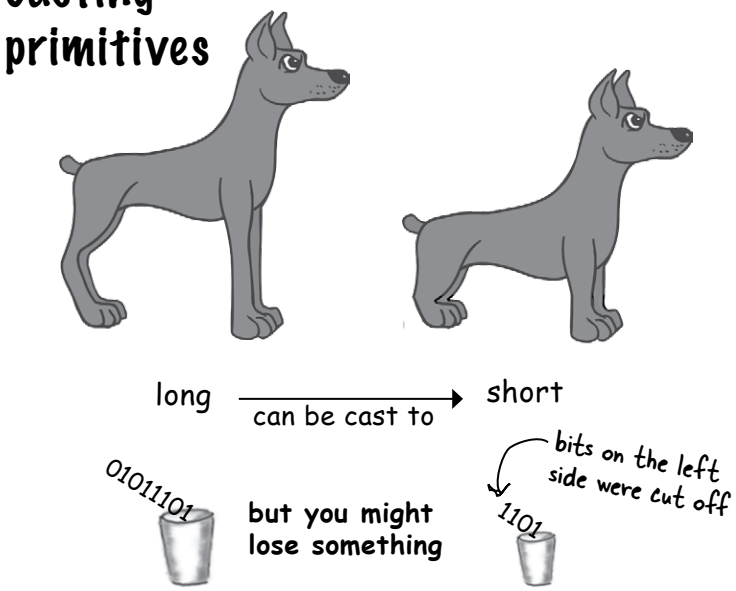
Trying to compile that makes the compiler laugh and mock you:

```
operator == cannot be applied to
      int,java.lang.String
     if (x == num) { }
               ^
```

So to get around the whole apples and oranges thing, we have to make the *String* "2" into the *int* 2. Built into the Java class library is a class called Integer (that's right, an Integer *class*, not the int *primitive*), and one of its jobs is to take Strings that *represent* numbers and convert them into *actual* numbers.



Casting primitives



In chapter 3 we talked about the sizes of the various primitives, and how you can't shove a big thing directly into a small thing:

```
long y = 42;
int x = y;    // won't compile
```

A *long* is bigger than an *int* and the compiler can't be sure where that *long* has been. It might have been out drinking with the other longs, and taking on really big values. To force the compiler to jam the value of a bigger primitive variable into a smaller one, you can use the **cast** operator. It looks like this:

```
long y = 42;     // so far so good
int x = (int) y; // x = 42 cool!
```

Putting in the cast tells the compiler to take the value of y, chop it down to int size, and set x equal to whatever is left. If the value of y was bigger than the maximum value of x, then what's left will be a weird (but calculable*) number:

```
long y = 40002;
// 40002 exceeds the 16-bit limit of a short
short x = (short) y;  // x now equals -25534!
```

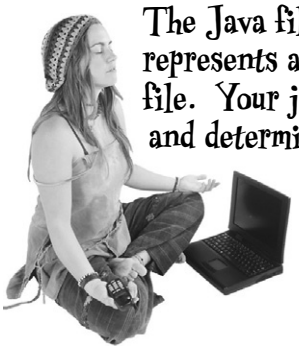
Still, the point is that the compiler lets you do it. And let's say you have a floating point number, and you just want to get at the whole number (*int*) part of it:

```
float f = 3.14f;
int x = (int) f;   //  x will equal 3
```

And don't even *think* about casting anything to a boolean or vice versa—just walk away.

**It involves sign bits, binary, 'two's complement' and other geekery, all of which are discussed at the beginning of appendix B.*

BE the JVM



The Java file on this page represents a complete source file. Your job is to play JVM and determine what would be the output when the program runs?

```
class Output {

  public static void main(String [] args) {
    Output o = new Output();
    o.go();
  }

  void go() {
    int y = 7;
    for(int x = 1; x < 8; x++) {
      y++;
      if (x > 4) {
         System.out.print(++y + " ");
      }
      if (y > 14) {
        System.out.println(" x = " + x);
        break;
      }
    }
  }
}
```

```
File  Edit  Window  Help  OM
% java Output
12 14
```

-or-

```
File  Edit  Window  Help  Incense
% java Output
12 14 x = 6
```

-or-

```
File  Edit  Window  Help  Believe
% java Output
13 15 x = 6
```

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!

```
x++;
```

```
if (x == 1) {
```

```
System.out.println(x + " " + y);
```

```
class MultiFor {
```

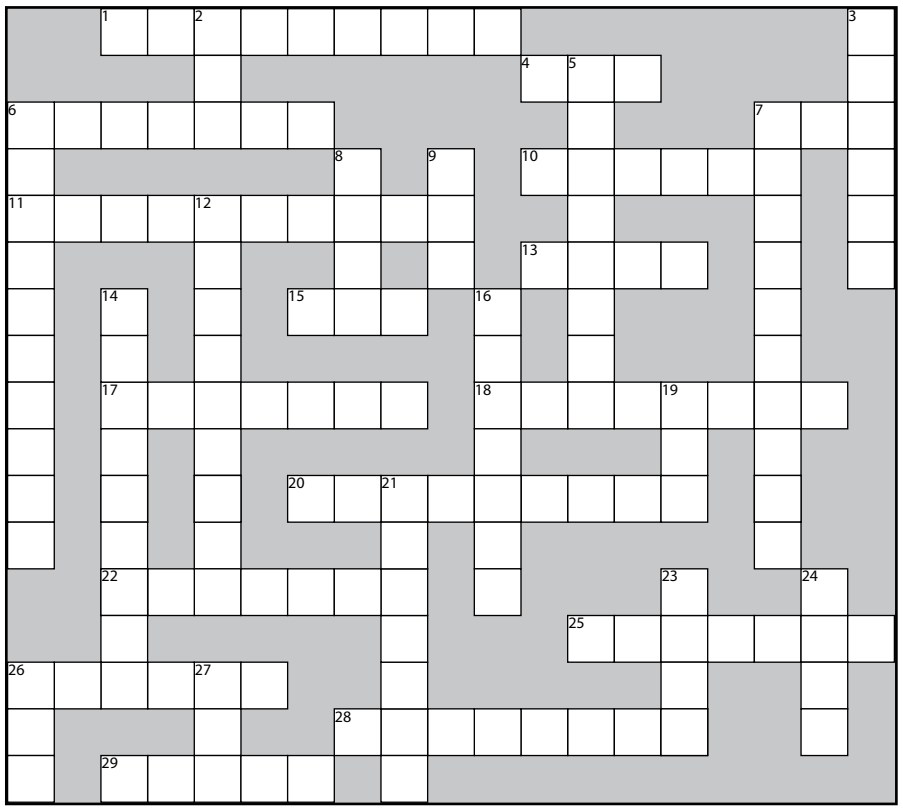
```
for(int  y = 4; y > 2; y--) {
```

```
for(int x = 0; x < 4; x++) {
```

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

```
File  Edit  Window  Help  Raid
% java MultiFor
0 4
0 3
1 4
1 3
3 4
3 3
```

puzzle: JavaCross



JavaCross

How does a crossword puzzle help you learn Java? Well, all of the words **are** Java related. In addition, the clues provide metaphors, puns, and the like. These mental twists and turns burn alternate routes to Java knowledge, right into your brain!

Across

1. Fancy computer word for build
4. Multi-part loop
6. Test first
7. 32 bits
10. Method's answer
11. Prepcode-esque
13. Change
15. The big toolkit
17. An array unit
18. Instance or local
20. Automatic toolkit
22. Looks like a primitive, but..
25. Un-castable
26. Math method
28. Converter method
29. Leave early

Down

2. Increment type
3. Class's workhorse
5. Pre is a type of _____
6. For's iteration _____
7. Establish first value
8. While or For
9. Update an instance variable
12. Towards blastoff
14. A cycle
16. Talkative package
19. Method messenger (abbrev.)
21. As if
23. Add after
24. Pi house
26. Compile it and ____
27. ++ quantity

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 MixFor5 {
  public static void main(String [] args) {
    int x = 0;
    int y = 30;
    for (int outer = 0; outer < 3; outer++) {
      for(int inner = 4; inner > 1; inner--) {

        [          ]   <-- candidate code goes here

        y = y - 2;
        if (x == 6) {
          break;
        }
        x = x + 3;
      }
      y = y - 2;
    }
    System.out.println(x + " " + y);
  }
}
```

Candidates:

```
x = x + 3;
```

```
x = x + 6;
```

```
x = x + 2;
```

```
x++;
```

```
x--;
```

```
x = x + 0;
```

Possible output:

```
45 6
```

```
36 6
```

```
54 6
```

```
60 10
```

```
18 6
```

```
6 14
```

```
12 14
```

match each candidate with one of the possible outputs

Exercise Solutions

Be the JVM:

```
class Output {

  public static void main(String [] args) {
    Output o = new Output();
    o.go();
  }
  void go() {
    int y = 7;
    for(int x = 1; x < 8; x++) {
      y++;
      if (x > 4) {
         System.out.print(++y + " ");
      }
      if (y > 14) {
        System.out.println(" x = " + x);
        break;
      }
    }
  }
}
```

Did you remember to factor in the break statement? How did that affect the output?

```
File  Edit  Window  Help  MotorcycleMaintenance

% java Output
13 15 x = 6
```

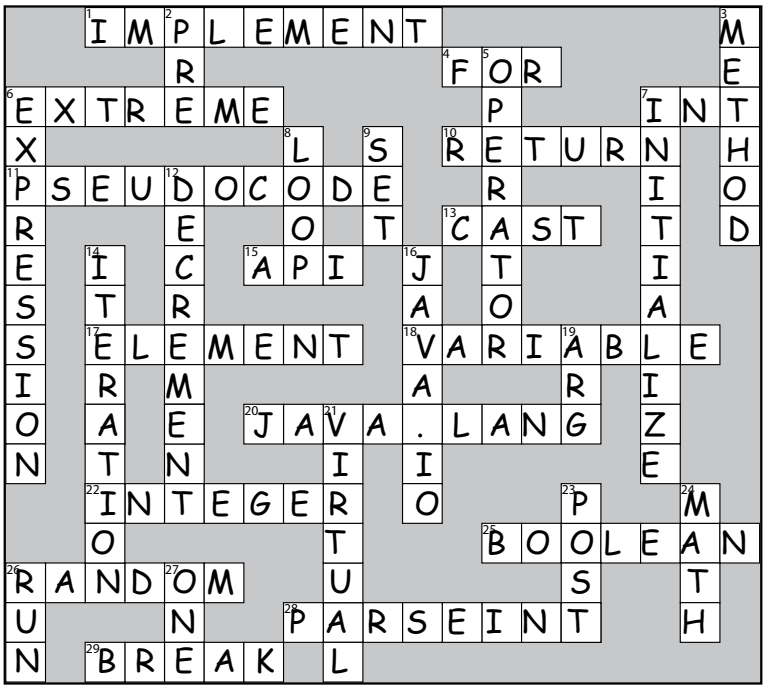
Code Magnets:

```
class MultiFor {

  public static void main(String [] args) {

    for(int x = 0; x < 4; x++) {

      for(int y = 4; y > 2; y--) {
        System.out.println(x + " " + y);
      }

      if (x == 1) {
        x++;
      }
    }
  }
}
```

What would happen if this code block came before the 'y' for loop?

```
File  Edit  Window  Help  Monopole

% java MultiFor
0 4
0 3
1 4
1 3
3 4
3 3
```

Puzzle Solutions



Candidates:

```
x = x + 3;
x = x + 6;
x = x + 2;
x++;
x--;
x = x + 0;
```

Possible output:

```
45 6
36 6
54 6
60 10
18 6
6 14
12 14
```

Using the Java Library

Java ships with hundreds of pre-built classes. You don't have to reinvent the wheel if you know how to find what you need in the Java library, known as the **Java API**. *You've got better things to do*. If you're going to write code, you might as well write *only* the parts that are truly custom for your application. You know those programmers who walk out the door each night at 5 PM? The ones who don't even show *up* until 10 AM? **They use the Java API.** And about eight pages from now, so will you. The core Java library is a giant pile of classes just waiting for you to use like building blocks, to assemble your own program out of largely pre-built code. The Ready-bake Java we use in this book is code you don't have to create from scratch, but you still have to type it. The Java API is full of code you don't even have to *type*. All you need to do is learn to use it.

In our last chapter, we left you with the cliff-hanger. A bug.

How it's supposed to look

Here's what happens when we run it and enter the numbers 1,2,3,4,5,6. Lookin' good.

A complete game interaction
(your mileage may vary)

```
File  Edit  Window  Help  Smile
%java SimpleDotComGame
enter a number  1
miss
enter a number  2
miss
enter a number  3
miss
enter a number  4
hit
enter a number  5
hit
enter a number  6
kill
You took 6 guesses
```

How the bug looks

Here's what happens when we enter 2,2,2.

A different game interaction
(yikes)

```
File  Edit  Window  Help  Faint
%java SimpleDotComGame
enter a number  2
hit
enter a number  2
hit
enter a number  2
kill
You took 3 guesses
```

In the current version, once you get a hit, you can simply repeat that hit two more times for the kill!

So what happened?

```
public String checkYourself(String stringGuess) {

    int guess = Integer.parseInt(stringGuess);

    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
```

Convert the String to an int.

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 thing in the array.

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.

Here's where it goes wrong. We counted a hit every time the user guessed a cell location, *even if that location had already been hit!*

We need a way to know that when a user makes a hit, he hasn't previously hit that cell. If he has, then we don't want to count it as a hit.

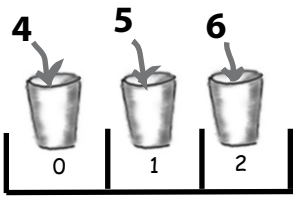
How do we fix it?

We need a way to know whether a cell has already been hit. Let's run through some possibilities, but first, we'll look at what we know so far...

We have a virtual row of 7 cells, and a DotCom will occupy three consecutive cells somewhere in that row. This virtual row shows a DotCom placed at cell locations 4,5 and 6.

The virtual row, with the 3 cell locations for the DotCom object.

The DotCom has an instance variable—an int array—that holds that DotCom object's cell locations.



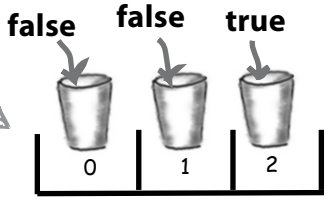
locationCells
(instance variable of the DotCom)

The array instance variable that holds the DotCom's cell locations. This DotCom holds the 3 values of 4, 5, and 6. Those are the numbers the user needs to guess.

1 Option one

We could make a second array, and each time the user makes a hit, we store that hit in the second array, and then check that array each time we get a hit, to see if that cell has been hit before.

A 'true' in a particular index in the array means that the cell location at that same index in the OTHER array (locationCells) has been hit.



hitCells array
(this would be a new boolean array instance variable of the DotCom)

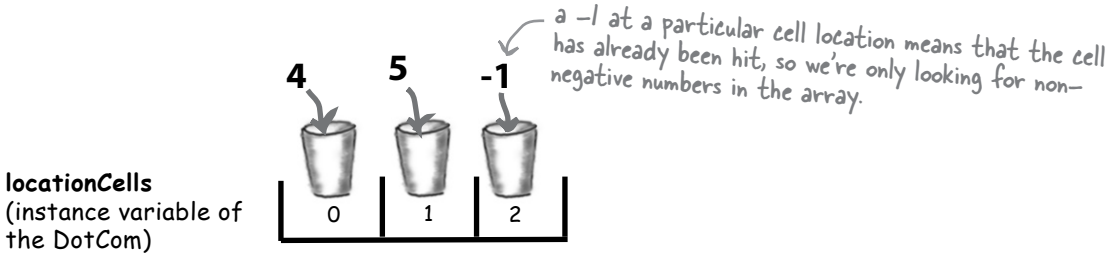
This array holds three values representing the 'state' of each cell in the DotCom's location cells array. For example, if the cell at index 2 is hit, then set index 2 in the "hitCells" array to 'true'.

Option one is too clunky

Option one seems like more work than you'd expect. It means that each time the user makes a hit, you have to change the state of the *second* array (the 'hitCells' array), oh -- but first you have to CHECK the 'hitCells' array to see if that cell has already been hit anyway. It would work, but there's got to be something better...

② Option two

We could just keep the one original array, but change the value of any hit cells to -1. That way, we only have ONE array to check and manipulate



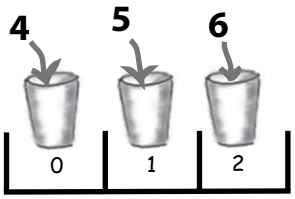
Option two is a little better, but still pretty clunky

Option two is a little less clunky than option one, but it's not very efficient. You'd still have to loop through all three slots (index positions) in the array, even if one or more are already invalid because they've been 'hit' (and have a -1 value). There has to be something better...

③ Option three

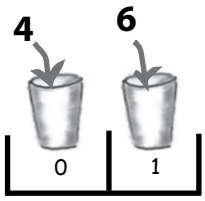
We delete each cell location as it gets hit, and then modify the array to be smaller. Except arrays can't change their size, so we have to make a **new** array and copy the remaining cells from the old array into the new smaller array.

locationCells array BEFORE any cells have been hit



The array starts out with a size of 3, and we loop through all 3 cells (positions in the array) to look for a match between the user guess and the cell value (4,5, 6).

locationCells array AFTER cell '5', which was at index 1 in the array, has been hit



When cell '5' is hit, we make a new, smaller array with only the remaining cell locations, and assign it to the original locationCells reference.

Option three would be much better if the array could shrink, so that we wouldn't have to make a new smaller array, copy the remaining values in, and reassign the reference.

The original prepcode for part of the checkYourself() method:

REPEAT with each of the location cells in the ***int*** array
  // COMPARE the user guess to the location cell
  IF the user guess matches
    INCREMENT the number of hits
    // FIND OUT if it was the last location cell:
    IF number of hits is 3, **RETURN** "kill"
    ELSE it was not a kill, so **RETURN** "hit"
    END IF
  ELSE user guess did not match, so **RETURN** "miss"
  END IF
END REPEAT

Life would be good if only we could change it to:

REPEAT with each of the ***remaining*** location cells
  // COMPARE the user guess to the location cell
  IF the user guess matches
    REMOVE this cell from the array
    // FIND OUT if it was the last location cell:
    IF the array is now empty, **RETURN** "kill"
    ELSE it was not a kill, so **RETURN** "hit"
    END IF
  ELSE user guess did not match, so **RETURN** "miss"
  END IF
END REPEAT


If only I could find an array that could **shrink** when you **remove** something. And one that you didn't have to loop through to check each element, but instead you could just **ask it if it contains** what you're looking for. And it would let you **get** things out of it, without having to know exactly which slot the things are in. That would be dreamy. But I know it's just a fantasy...

Wake up and smell the library

As if by magic, there really *is* such a thing.

But it's not an *array*, it's an *ArrayList*.

A class in the core Java library (the API).

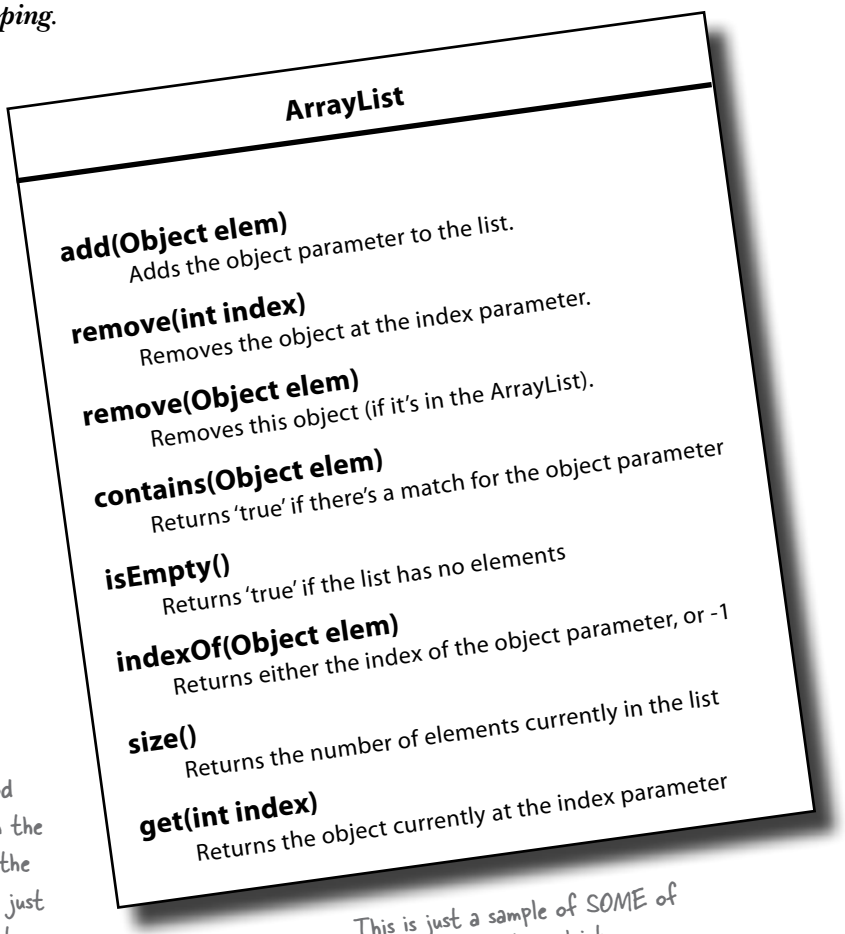
The Java Standard Edition (which is what you have unless you're working on the Micro Edition for small devices and believe me, *you'd know*) ships with hundreds of pre-built classes. Just like our Ready-Bake code except that these built-in classes are already compiled.

That means no typing.

Just use 'em.

One of a gazillion classes in the Java library.

You can use it in your code as if you wrote it yourself.



This is just a sample of SOME of the methods in ArrayList.

(Note: the add(Object elem) method actually looks a little stranger than the one we've shown here... we'll get to the real one later in the book. For now, just think of it as an add() method that takes the object you want to add.)

Some things you can do with ArrayList

Don't worry about this new <Egg> angle-bracket syntax right now; it just means "make this a list of Egg objects".

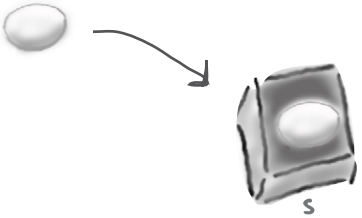
① Make one

```
ArrayList<Egg> myList = new ArrayList<Egg>();
```

A new ArrayList object is created on the heap. It's little because it's empty.

② Put something in it

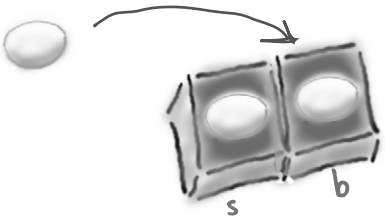
```
Egg s = new Egg();

myList.add(s);
```



Now the ArrayList grows a "box" to hold the Egg object.

③ Put another thing in it

```
Egg b = new Egg();

myList.add(b);
```



The ArrayList grows again to hold the second Egg object.

④ Find out how many things are in it

```
int theSize = myList.size();
```

The ArrayList is holding 2 objects so the size() method returns 2

⑤ Find out if it contains something

```
boolean isIn = myList.contains(s);
```

The ArrayList DOES contain the Egg object referenced by 's', so contains() returns true

⑥ Find out where something is (i.e. its index)

```
int idx = myList.indexOf(b);
```

ArrayList is zero-based (means first index is 0) and since the object referenced by 'b' was the second thing in the list, indexOf() returns 1

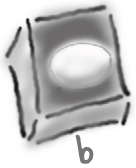
⑦ Find out if it's empty

```
boolean empty = myList.isEmpty();
```

it's definitely NOT empty, so isEmpty() returns false

⑧ Remove something from it

```
myList.remove(s);
```



Hey look — it shrank!

Fill in the rest of the table below by looking at the ArrayList code on the left and putting in what you think the code might be if it were using a regular array instead. We don't expect you to get all of them exactly right, so just make your best guess.

ArrayList	regular array
`ArrayList<String> myList = new ArrayList<String>();`	String [] myList = new String[2];
`String a = new String("whoohoo");`	String a = new String("whoohoo");
`myList.add(a);`	
`String b = new String("Frog");`	String b = new String("Frog");
`myList.add(b);`	
`int theSize = myList.size();`	
`Object o = myList.get(1);`	
`myList.remove(1);`	
`boolean isIn = myList.contains(b);`	

there are no Dumb Questions

Q: So ArrayList is cool, but how would I know it exists?

A: The question is really, "How do I *know* what's in the API?" and that's the key to your success as a Java programmer. Not to mention your key to being as lazy as possible while still managing to build software. You might be amazed at how much time you can save when somebody else has already done most of the heavy lifting, and all you have to do is step in and create the fun part.

But we digress... the short answer is that you spend some time learning what's in the core API. The long answer is at the end of this chapter, where you'll learn *how* to do that.

Q: But that's a pretty big issue. Not only do I need to know that the Java library comes with ArrayList, but more importantly I have to know that ArrayList is the thing that can do what I want! So how do I go from a need-to-do-something to a-way-to-do-it using the API?

A: Now you're really at the heart of it. By the time you've finished this book, you'll have a good grasp of the language, and the rest of your learning curve really is about knowing how to get from a problem to a solution, with you writing the least amount of code. If you can be patient for a few more pages, we start talking about it at the end of this chapter.

Java Exposed

This week's interview: ArrayList, on arrays

HeadFirst: So, ArrayLists are like arrays, right?

ArrayList: In their dreams! ***I*** am an *object* thank you very much.

HeadFirst: If I'm not mistaken, arrays are objects too. They live on the heap right there with all the other objects.

ArrayList: Sure arrays go on the heap, ***duh***, but an array is still a wanna-be ArrayList. A poser. Objects have state *and* behavior, right? We're clear on that. But have you actually tried calling a method on an array?

HeadFirst: Now that you mention it, can't say I have. But what method would I call, anyway? I only care about calling methods on the stuff I put *in* the array, not the array itself. And I can use array syntax when I want to put things in and take things out of the array.

ArrayList: Is that so? You mean to tell me you actually *removed* something from an array? (Sheesh, where do they *train* you guys? McJava's?)

HeadFirst: Of *course* I take something out of the array. I say Dog d = dogArray[1] and I get the Dog object at index 1 out of the array.

ArrayList: Allright, I'll try to speak slowly so you can follow along. You were *not*, I repeat *not*, removing that Dog from the array. All you did was make a copy of the *reference to the Dog* and assign it to another Dog variable.

HeadFirst: Oh, I see what you're saying. No I didn't actually remove the Dog object from the array. It's still there. But I can just set its reference to null, I guess.

ArrayList: But I'm a first-class object, so I have methods and I can actually, you know, *do* things like remove the Dog's reference from myself, not just set it to null. And I can change my size, *dynamically* (look it up). Just try to get an *array* to do that!

HeadFirst: Gee, hate to bring this up, but the rumor is that you're nothing more than a glorified but less-efficient array. That in fact you're just a wrapper for an array, adding extra methods for things like resizing that I would have had to write myself. And while we're at it, *you can't even hold primitives*! Isn't that a big limitation?

ArrayList: I can't *believe* you buy into that urban legend. No, I am *not* just a less-efficient array. I will admit that there are a few *extremely* rare situations where an array might be just a tad, I repeat, *tad* bit faster for certain things. But is it worth the *miniscule* performance gain to give up all this *power*? Still, look at all this *flexibility*. And as for the primitives, of *course* you can put a primitive in an ArrayList, as long as it's wrapped in a primitive wrapper class (you'll see a lot more on that in chapter 10). And as of Java 5.0, that wrapping (and unwrapping when you take the primitive out again) happens automatically. And allright, I'll *acknowledge* that yes, if you're using an ArrayList of *primitives*, it probably is faster with an array, because of all the wrapping and unwrapping, but still... who really uses primitives *these* days?

Oh, look at the time! *I'm late for Pilates.* We'll have to do this again sometime.

Comparing ArrayList to a regular array

ArrayList	regular array
`ArrayList<String> myList = new ArrayList<String>();`	`String [] myList = new String[2];`
`String a = new String("whoohoo");`	`String a = new String("whoohoo");`
`myList.add(a);`	`myList[0] = a;`
`String b = new String("Frog");`	`String b = new String("Frog");`
`myList.add(b);`	`myList[1] = b;`
`int theSize = myList.size();`	`int theSize = myList.length;`
`Object o = myList.get(1);`	`String o = myList[1];`
`myList.remove(1);`	`myList[1] = null;`

`boolean isIn = myList.contains(b);`	`boolean isIn = false;`
	`for (String item : myList) {`
	`if (b.equals(item)) {`
	`isIn = true;`
	`break;`
	`}`
	`}`

Notice how with ArrayList, you're working with an object of type ArrayList, so you're just invoking regular old methods on a regular old object, using the regular old dot operator.

With an *array*, you use *special array syntax* (like myList[0] = foo) that you won't use anywhere else except with arrays. Even though an array *is* an object, it lives in its own special world and you can't invoke any methods on it, although you can access its one and only instance variable, *length*.

Comparing ArrayList to a regular array

① A plain old array has to know its size at the time it's created.

But for ArrayList, you just make an object of type ArrayList. Every time. It never needs to know how big it should be, because it grows and shrinks as objects are added or removed.

```
new String[2]
```

Needs a size.

```
new ArrayList<String>()
```

No size required (although you can give it a size if you want to).

② To put an object in a regular array, you must assign it to a specific location.

(An index from 0 to one less than the length of the array.)

```
myList[1] = b;
```

Needs an index.

If that index is outside the boundaries of the array (like, the array was declared with a size of 2, and now you're trying to assign something to index 3), it blows up at runtime.

With ArrayList, you can specify an index using the *add(anInt, anObject)* method, or you can just keep saying *add(anObject)* and the ArrayList will keep growing to make room for the new thing.

```
myList.add(b);
```

No index.

③ Arrays use array syntax that's not used anywhere else in Java.

But ArrayLists are plain old Java objects, so they have no special syntax.

```
myList[1]
```

The array brackets [ ] are special syntax used only for arrays.

④ ArrayLists in Java 5.0 are parameterized.

We just said that unlike arrays, ArrayLists have no special syntax. But they *do* use something special that was added to Java 5.0 Tiger—***parameterized types***.

```
ArrayList<String>
```

The <String> in angle brackets is a "type parameter". ArrayList<String> means simply "a list of Strings", as opposed to ArrayList<Dog> which means, "a list of Dogs".

Prior to Java 5.0, there was no way to declare the *type* of things that would go in the ArrayList, so to the compiler, all ArrayLists were simply heterogenous collections of objects. But now, using the <typeGoesHere> syntax, we can declare and create an ArrayList that knows (and restricts) the types of objects it can hold. We'll look at the details of parameterized types in ArrayLists in the Collections chapter, so for now, don't think too much about the angle bracket <> syntax you see when we use ArrayLists. Just know that it's a way to force the compiler to allow only a specific type of object *(the type in angle brackets)* in the ArrayList.

Let's fix the DotCom code.

Remember, this is how the buggy version looks:

We've renamed the class DotCom now (instead of SimpleDotCom), for the new advanced version, but this is the same code you saw in the last chapter.

```
public class DotCom {

   int[] locationCells;
   int numOfHits = 0;

   public void setLocationCells(int[] locs) {
      locationCells = locs;
   }

   public String checkYourself(String stringGuess) {
      int guess = Integer.parseInt(stringGuess);
      String result = "miss";

      for (int cell : locationCells) {
         if (guess == cell) {

            result = "hit";
            numOfHits++;

            break;
         }
      } // out of the loop

      if (numOfHits == locationCells.length) {
          result = "kill";
      }
      System.out.println(result);
         return result;
    } // close method
} // close class
```

Where it all went wrong. We counted each guess as a hit, without checking whether that cell had already been hit.

prep code | test code | real code

New and improved DotCom class


```
import java.util.ArrayList;

public class DotCom {

   private ArrayList<String> locationCells;
   // private int numOfHits;
   // don't need that now

   public void setLocationCells(ArrayList<String> loc) {
      locationCells = loc;
   }


   public String checkYourself(String userInput) {

      String result = "miss";

         int index = locationCells.indexOf(userInput);

         if (index >= 0) {

               locationCells.remove(index);


               if (locationCells.isEmpty()) {
                  result = "kill";
               } else {
                  result = "hit";
               } // close if

          } // close outer if

        return result;
   } // close method
} // close class
```

Ignore this line for now; we talk about it at the end of the chapter.

Change the int array to an ArrayList that holds Strings.

New and improved argument name.

Find out if the user guess is in the ArrayList, by asking for its index. If it's not in the list, then indexOf() returns a -1.

If index is greater than or equal to zero, the user guess is definitely in the list, so remove it.

If the list is empty, this was the killing blow!

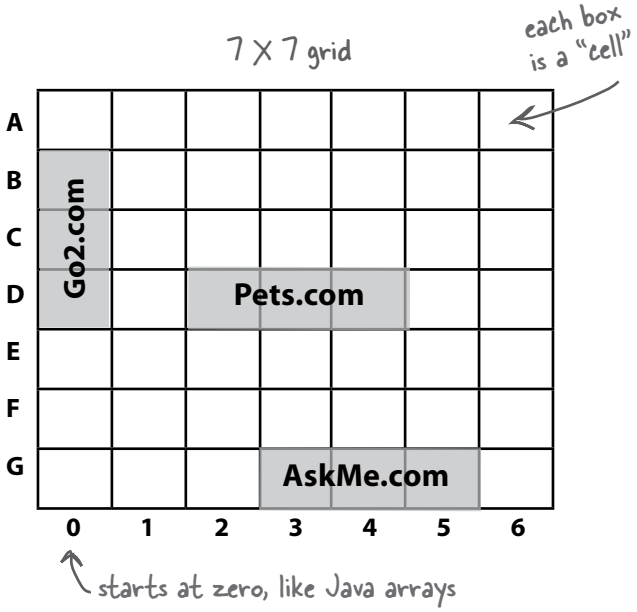
Let's build the REAL game: "Sink a Dot Com"

We've been working on the 'simple' version, but now let's build the real one. Instead of a single row, we'll use a grid. And instead of one DotCom, we'll use three.

Goal: Sink all of the computer's Dot Coms in the fewest number of guesses. You're given a rating level based on how well you perform.

Setup: When the game program is launched, the computer places three Dot Coms, randomly, on the **virtual 7 x 7 grid**. When that's complete, the game asks for your first guess.

How you play: We haven't learned to build a GUI yet, so this version works at the command-line. The computer will prompt you to enter a guess (a cell), which you'll type at the command-line (as "A3", "C5", etc.). In response to your guess, you'll see a result at the command-line, either "hit", "miss", or "You sunk Pets.com" (or whatever the lucky Dot Com of the day is). When you've sent all three Dot Coms to that big 404 in the sky, the game ends by printing out your rating.



You're going to build the Sink a Dot Com game, with a 7 x 7 grid and three Dot Coms. Each Dot Com takes up three cells.

part of a game interaction

```
File Edit Window Help Sell
%java DotComBust
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 Pets.com   : (
kill
Enter a guess  B4
miss
Enter a guess  G3
hit
Enter a guess  G4
hit
Enter a guess  G5
Ouch! You sunk AskMe.com   : (
```

What needs to change?

We have three classes that need to change: the DotCom class (which is now called DotCom instead of SimpleDotCom), the game class (DotComBust) and the game helper class (which we won't worry about now).

A DotCom class

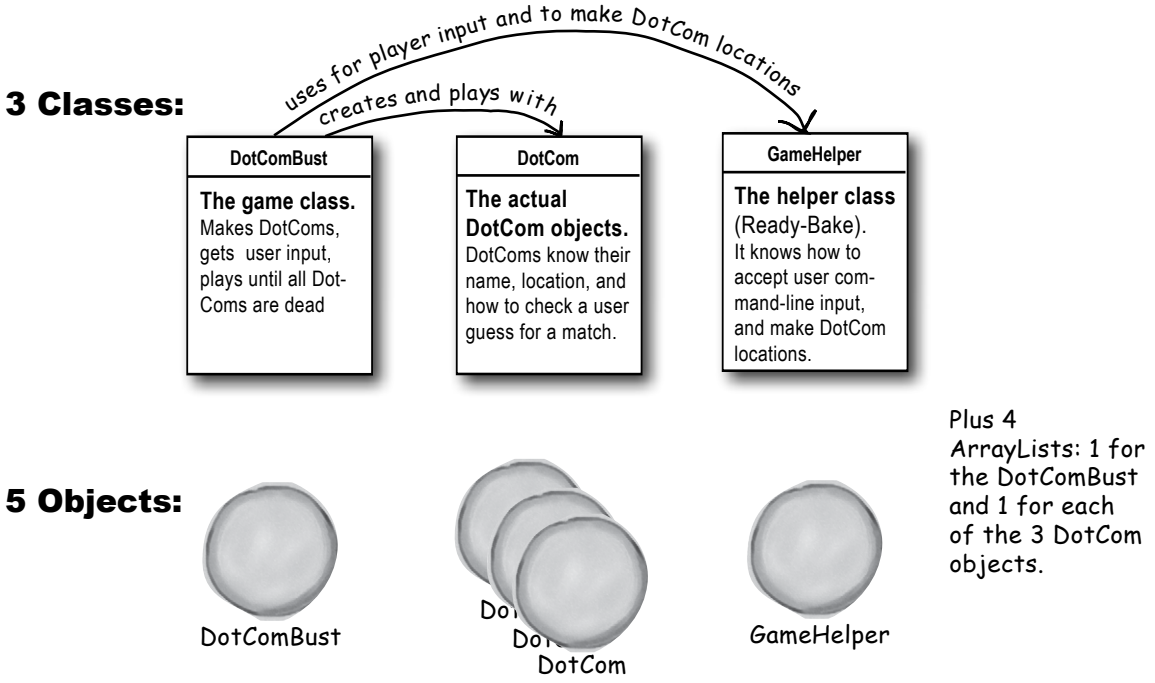
- **Add a *name* variable**
  to hold the name of the DotCom ("Pets.com", "Go2.com", etc.) so each DotCom can print its name when it's killed (see the output screen on the opposite page).

B DotComBust class (the game)

- **Create *three* DotComs instead of one.**
- **Give each of the three DotComs a *name*.**
  Call a setter method on each DotCom instance, so that the DotCom can assign the name to its name instance variable.

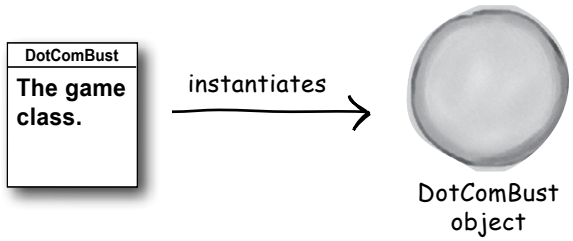
DotComBust class continued...

- **Put the DotComs on a grid rather than just a single row, and do it for all three DotComs.**
  This step is now way more complex than before, if we're going to place the DotComs randomly. Since we're not here to mess with the math, we put the algorithm for giving the DotComs a location into the GameHelper (Ready-bake) class.
- **Check each user guess *with all three DotComs*, instead of just one.**
- **Keep playing the game** (i.e accepting user guesses and checking them with the remaining DotComs) ***until there are no more live DotComs.***
- **Get out of main.** We kept the simple one in main just to... keep it simple. But that's not what we want for the *real* game.



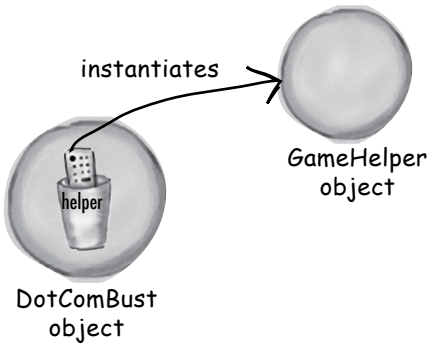
Who does what in the DotComBust game (and when)

1



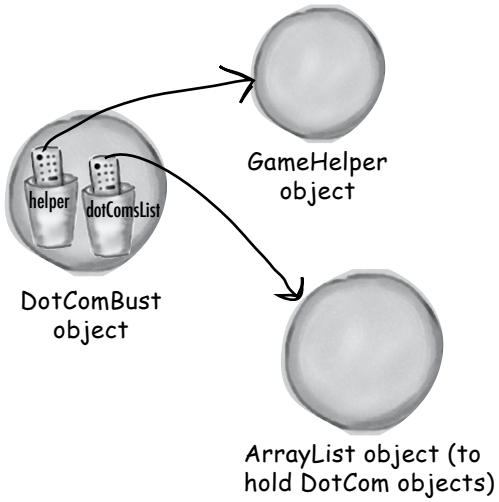
The main() method in the DotComBust class instantiates the DotComBust object that does all the game stuff.

2

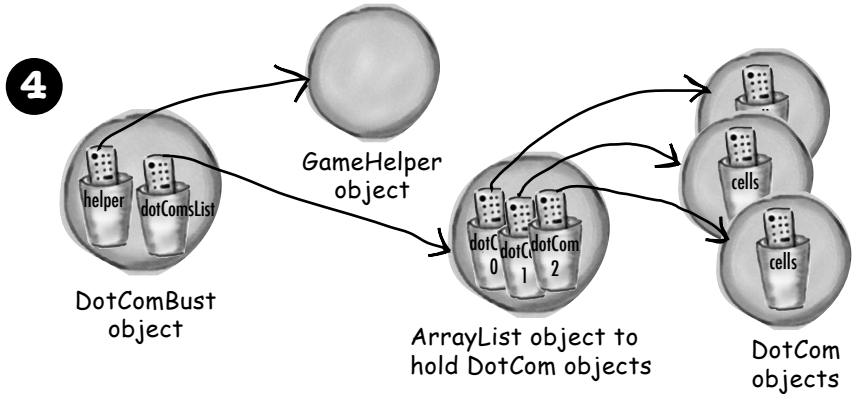


The DotComBust (game) object instantiates an instance of GameHelper, the object that will help the game do its work.

3

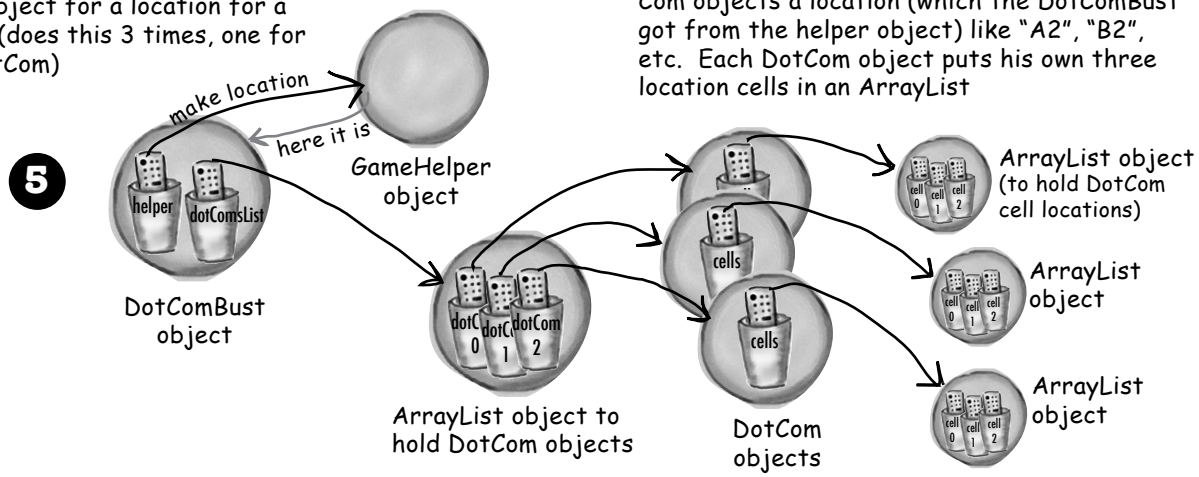


The DotComBust object instantiates an ArrayList that will hold the 3 DotCom objects.



The DotComBust object creates three DotCom objects (and puts them in the ArrayList)

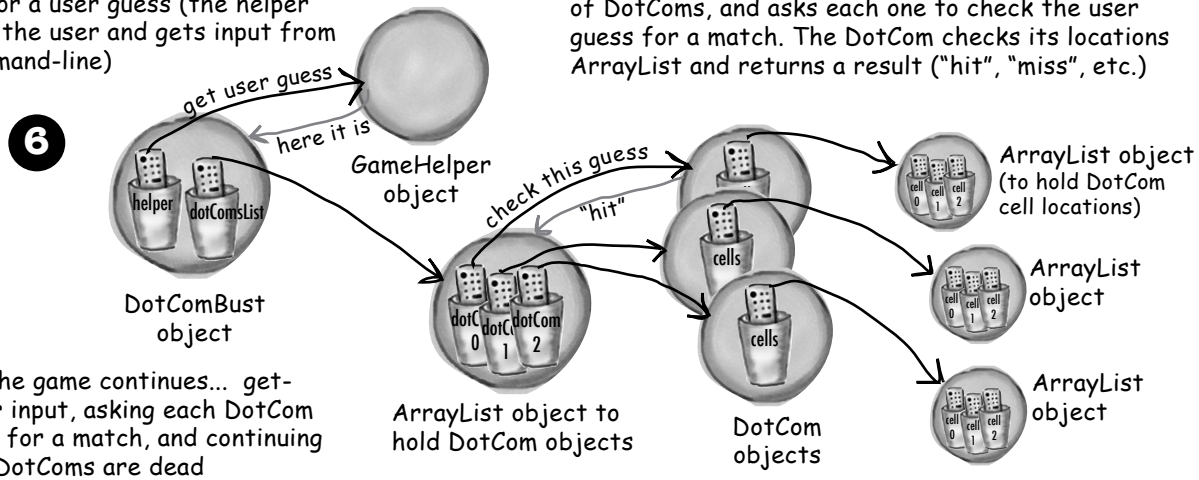
The DotComBust object asks the helper object for a location for a DotCom (does this 3 times, one for each DotCom)



The DotComBust object gives each of the DotCom objects a location (which the DotComBust got from the helper object) like "A2", "B2", etc. Each DotCom object puts his own three location cells in an ArrayList

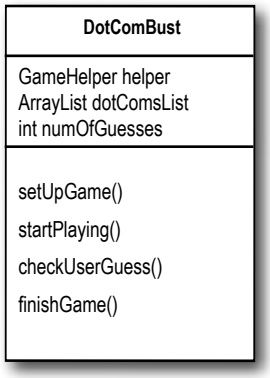
---

The DotComBust object asks the helper object for a user guess (the helper prompts the user and gets input from the command-line)



The DotComBust object loops through the list of DotComs, and asks each one to check the user guess for a match. The DotCom checks its locations ArrayList and returns a result ("hit", "miss", etc.)

And so the game continues... getting user input, asking each DotCom to check for a match, and continuing until all DotComs are dead



Prep code for the real DotComBust class

The DotComBust class has three main jobs: set up the game, play the game until the DotComs are dead, and end the game. Although we could map those three jobs directly into three methods, we split the middle job (play the game) into *two* methods, to keep the granularity smaller. Smaller methods (meaning smaller chunks of functionality) help us test, debug, and modify the code more easily.

Variable Declarations

DECLARE and instantiate the ***GameHelper*** instance variable, named ***helper***.

DECLARE and instantiate an ***ArrayList*** to hold the list of DotComs (initially three) Call it *dotComsList*.

DECLARE an int variable to hold the number of user guesses (so that we can give the user a score at the end of the game). Name it ***numOfGuesses*** and set it to 0.

Method Declarations

DECLARE a ***setUpGame()*** method to create and initialize the DotCom objects with names and locations. Display brief instructions to the user.

DECLARE a ***startPlaying()*** method that asks the player for guesses and calls the checkUserGuess() method until all the DotCom objects are removed from play.

DECLARE a ***checkUserGuess()*** method that loops through all remaining DotCom objects and calls each DotCom object's checkYourself() method.

DECLARE a ***finishGame()*** method that prints a message about the user's performance, based on how many guesses it took to sink all of the DotCom objects.

Method Implementations

METHOD: ***void setUpGame()***

// make three DotCom objects and name them

CREATE three DotCom objects.

SET a name for each DotCom.

ADD the DotComs to the ***dotComsList*** (the ArrayList).

REPEAT with each of the DotCom objects in the ***dotComsList*** array

CALL the ***placeDotCom()*** method on the helper object, to get a randomly-selected location for this DotCom (three cells, vertically or horizontally aligned, on a 7 X 7 grid).

SET the location for each DotCom based on the result of the ***placeDotCom()*** call.

END REPEAT

END METHOD

prep code | test code | real code

Method implementations continued:

METHOD: ***void startPlaying()***

REPEAT while any DotComs exist
GET user input by calling the helper ***getUserInput()*** method
EVALUATE the user's guess by ***checkUserGuess()*** method
END REPEAT
END METHOD

METHOD: ***void checkUserGuess(String userGuess)***

// find out if there's a hit (and kill) on any DotCom
INCREMENT the number of user guesses in the ***numOfGuesses*** variable
SET the local ***result*** variable (a ***String***) to "miss", assuming that the user's guess will be a miss.
REPEAT with each of the DotObjects in the ***dotComsList*** array
EVALUATE the user's guess by calling the DotCom object's ***checkYourself()*** method
SET the result variable to "hit" or "kill" if appropriate
IF the result is "kill", **REMOVE** the DotCom from the ***dotComsList***
END REPEAT
DISPLAY the ***result*** value to the user
END METHOD

METHOD: ***void finishGame()***

DISPLAY a generic "game over" message, then:
IF number of user guesses is small,
DISPLAY a congratulations message
ELSE
DISPLAY an insulting one
END IF
END METHOD

Sharpen your pencil

How should we go from prep code to the final code? First we start with test code, and then test and build up our methods bit by bit. We won't keep showing you test code in this book, so now it's up to you to think about what you'd need to know to test these methods. And which method do you test and write first? See if you can work out some prep code for a set of tests. Prep code or even bullet points are good enough for this exercise, but if you want to try to write the *real* test code (in Java), knock yourself out.

the DotComBust code (the game)

prep code | test code | real code

```
import java.util.*;
public class DotComBust {

  private GameHelper helper = new GameHelper();
  private ArrayList<DotCom> dotComsList = new ArrayList<DotCom>();
  private int numOfGuesses = 0;

  private void setUpGame() {
    // first make some dot coms and give them locations
    DotCom one = new DotCom();
    one.setName("Pets.com");
    DotCom two = new DotCom();
    two.setName("eToys.com");
    DotCom three = new DotCom();
    three.setName("Go2.com");
    dotComsList.add(one);
    dotComsList.add(two);
    dotComsList.add(three);

    System.out.println("Your goal is to sink three dot coms.");
    System.out.println("Pets.com, eToys.com, Go2.com");
    System.out.println("Try to sink them all in the fewest number of guesses");

    for (DotCom dotComToSet : dotComsList) {
      ArrayList<String> newLocation = helper.placeDotCom(3);
      dotComToSet.setLocationCells(newLocation);
    } // close for loop
  } // close setUpGame method

  private void startPlaying() {
    while(!dotComsList.isEmpty()) {
      String userGuess = helper.getUserInput("Enter a guess");
      checkUserGuess(userGuess);
    } // close while
    finishGame();
  } // close startPlaying method
```

Sharpen your pencil

Annotate the code yourself!

Match the annotations at the bottom of each page with the numbers in the code. Write the number in the slot in front of the corresponding annotation.

You'll use each annotation just once, and you'll need all of the annotations.

declare and initialize the variables we'll need

get user input

ask the helper for a DotCom location

repeat with each DotCom in the list

print brief instructions for user

call the setter method on this DotCom to give it the location you just got from the helper

call our own checkUserGuess method

call our own finishGame method

make three DotCom objects, give 'em names, and stick 'em in the ArrayList

as long as the DotCom list is NOT empty

prep code | test code | real code

Whatever you do, DON'T turn the page!

Not until you've finished this exercise.

Our version is on the next page.

```
  private void checkUserGuess(String userGuess) {
      numOfGuesses++; (11)
      String result  = "miss"; (12)
      for (DotCom dotComToTest : dotComsList) { (13)
         result = dotComToTest.checkYourself(userGuess); (14)
         if (result.equals("hit")) {
             break; (15)
         }
         if (result.equals("kill")) {
             dotComsList.remove(dotComToTest); (16)
             break;
         }
      } // close for
      System.out.println(result); (17)
   } // close method

   private void finishGame() {
      System.out.println("All Dot Coms are dead! Your stock is now worthless.");
      if (numOfGuesses <= 18) {
         System.out.println("It only took you " + numOfGuesses + " guesses."); (18)
         System.out.println(" You got out before your options sank.");
      } else {
         System.out.println("Took you long enough. "+ numOfGuesses + " guesses.");
         System.out.println("Fish are dancing with your options.");
      }
   } // close method

   public static void main (String[] args) {
      DotComBust game = new DotComBust(); (19)
      game.setUpGame();  (20)
      game.startPlaying(); (21)
   } // close method
}
```

repeat with all DotComs in the list

this guy's dead, so take him out of the DotComs list then get out of the loop

print a message telling the user how he did in the game

print the result for the user

tell the game object to set up the game

increment the number of guesses the user has made

assume it's a 'miss', unless told otherwise

get out of the loop early, no point in testing the others

tell the game object to start the main game play loop (keeps asking for user input and checking the guess)

ask the DotCom to check the user guess, looking for a hit (or kill)

create the game object