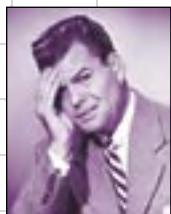


Your Brain on Java—A Learner's Guide

2nd Edition - Covers Java 5.0

Head First Java

Learn how threads
can change your life



Avoid embarrassing
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

Table of Contents (summary)

Intro	xxi
1 Breaking the Surface: <i>a quick dip</i>	1
2 A Trip to Objectville: <i>yes, there will be objects</i>	27
3 Know Your Variables: <i>primitives and references</i>	49
4 How Objects Behave: <i>object state affects method behavior</i>	71
5 Extra-Strength Methods: <i>flow control, operations, and more</i>	95
6 Using the Java Library: <i>so you don't have to write it all yourself</i>	125
7 Better Living in Objectville: <i>planning for the future</i>	165
8 Serious Polymorphism: <i>exploiting abstract classes and interfaces</i>	197
9 Life and Death of an Object: <i>constructors and memory management</i>	235
10 Numbers Matter: <i>math, formatting, wrappers, and statics</i>	273
11 Risky Behavior: <i>exception handling</i>	315
12 A Very Graphic Story: <i>intro to GUI, event handling, and inner classes</i>	353
13 Work on Your Swing: <i>layout managers and components</i>	399
14 Saving Objects: <i>serialization and I/O</i>	429
15 Make a Connection: <i>networking sockets and multithreading</i>	471
16 Data Structures: <i>collections and generics</i>	529
17 Release Your Code: <i>packaging and deployment</i>	581
18 Distributed Computing: <i>RMI with a dash of servlets, EJB, and Jini</i>	607
A Appendix A: <i>Final code kitchen</i>	649
B Appendix B: <i>Top Ten Things that didn't make it into the rest of the book</i>	659
Index	677

Table of Contents (the full version)

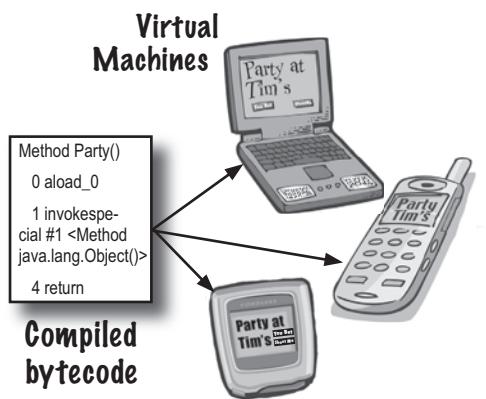
Intro

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?

Who is this book for?	xxii
What your brain is thinking	xxiii
Metacognition	xxv
Bend your brain into submission	xxvii
What you need for this book	xxviii
Technical editors	xxx
Acknowledgements	xxxi

1 Breaking the Surface

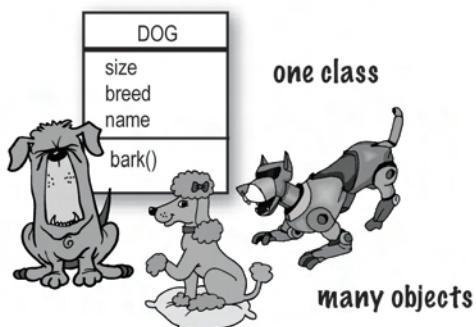
Java takes you to new places. From its humble release to the public as the (wimpy) version 1.02, Java seduced programmers with its friendly syntax, object-oriented features, memory management, and best of all—the promise of portability. We'll take a quick dip and write some code, compile it, and run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



The way Java works	2
Code structure in Java	7
Anatomy of a class	8
The main() method	9
Looping	11
Conditional branching (<i>if</i> tests)	13
Coding the “99 bottles of beer” app	14
Phrase-o-matic	16
Fireside chat: compiler vs. JVM	18
Exercises and puzzles	20

2 A Trip to Objectville

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

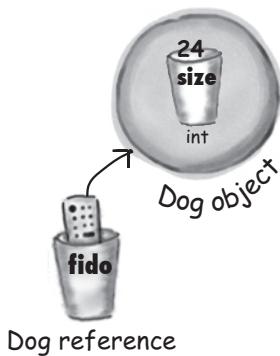


Chair Wars (Brad the OO guy vs. Larry the procedural guy)	28
Inheritance (an introduction)	31
Overriding methods (an introduction)	32
What's in a class? (methods, instance variables)	34
Making your first object	36
Using main()	38
Guessing Game code	39
Exercises and puzzles	42

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.



Declaring a variable (Java cares about <i>type</i>)	50
Primitive types ("I'd like a double with extra foam, please")	51
Java keywords	53
Reference variables (remote control to an object)	54
Object declaration and assignment	55
Objects on the garbage-collectible heap	57
Arrays (a first look)	59
Exercises and puzzles	63

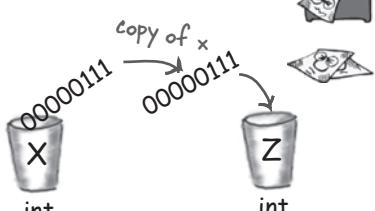
4 How Objects Behave

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

pass-by-value means
pass-by-copy

Methods use object state (bark different)	73
Method arguments and return types	74
Pass-by-value (the variable is <i>always</i> copied)	77
Getters and Setters	79
Encapsulation (do it or risk humiliation)	80
Using references in an array	83
Exercises and puzzles	88

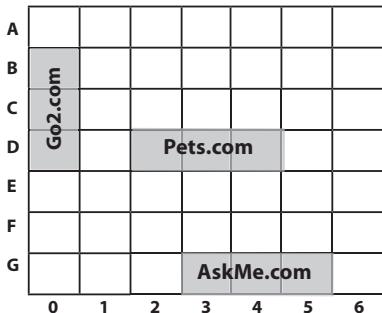




`foo.go(x); void go(int z) { }`

5 Extra-Strength Methods

We're gonna build the
Sink a Dot Com game



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).

Building the Sink a Dot Com game	96
Starting with the Simple Dot Com game (a simpler version)	98
Writing precode (pseudocode for the game)	100
Test code for Simple Dot Com	102
Coding the Simple Dot Com game	103
Final code for Simple Dot Com	106
Generating random numbers with Math.random()	111
Ready-bake code for getting user input from the command-line	112
Looping with <i>for</i> loops	114
Casting primitives from a large size to a smaller size	117
Converting a String to an int with Integer.parseInt()	117
Exercises and puzzles	118

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.

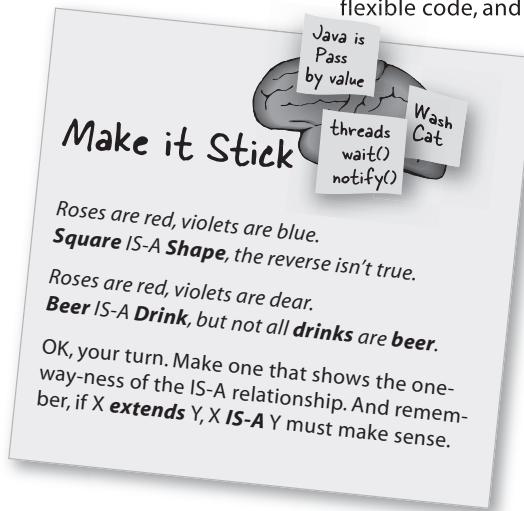
"Good to know there's an ArrayList in the java.util package. But by myself, how would I have figured that out?"

- Julia, 31, hand model



Analyzing the bug in the Simple Dot Com Game	126
ArrayList (taking advantage of the Java API)	132
Fixing the DotCom class code	138
Building the <i>real</i> game (Sink a Dot Com)	140
Precode for the <i>real</i> game	144
Code for the <i>real</i> game	146
<i>boolean</i> expressions	151
Using the library (Java API)	154
Using packages (import statements, fully-qualified names)	155
Using the HTML API docs and reference books	158
Exercises and puzzles	161

7 Better Living in Objectville



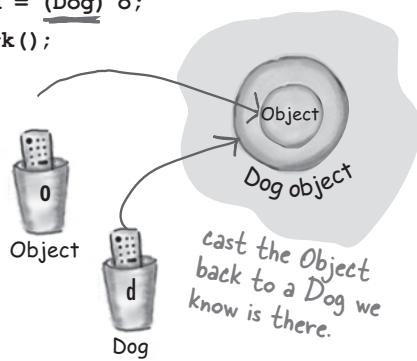
Plan your programs with the future in mind. What if you could write code that someone else could extend, easily? What if you could write code that was flexible, for those pesky last-minute spec changes? When you get on the Polymorphism Plan, you'll learn the 5 steps to better class design, the 3 tricks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

Understanding inheritance (superclass and subclass relationships)	168
Designing an inheritance tree (the Animal simulation)	170
Avoiding duplicate code (using inheritance)	171
Overriding methods	172
IS-A and HAS-A (bathtub girl)	177
What do you inherit from your superclass?	180
What does inheritance really <i>buy</i> you?	182
Polymorphism (using a supertype reference to a subclass object)	183
Rules for overriding (don't touch those arguments and return types!)	190
Method overloading (nothing more than method name re-use)	191
Exercises and puzzles	192

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...

```
Object o = al.get(id);
Dog d = (Dog) o;
d.bark();
```



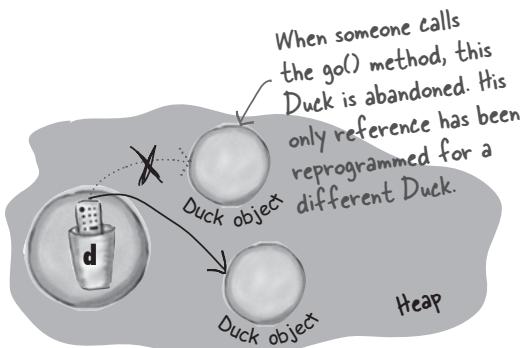
Some classes just should <i>not</i> be instantiated	200
Abstract classes (<i>can't</i> be instantiated)	201
Abstract methods (must be implemented)	203
Polymorphism in action	206
Class Object (the ultimate superclass of <i>everything</i>)	208
Taking objects out of an ArrayList (they come out as type Object)	211
Compiler checks the reference type (before letting you call a method)	213
Get in touch with your inner object	214
Polymorphic references	215
Casting an object reference (moving lower on the inheritance tree)	216
Deadly Diamond of Death (multiple inheritance problem)	223
Using interfaces (the best solution!)	224
Exercises and puzzles	230



9

Life and Death of an Object

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



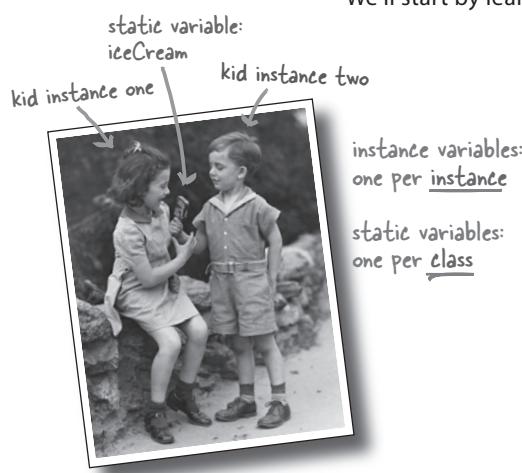
'd' is assigned a new Duck object, leaving the original (first) Duck object abandoned. That first Duck is toast..

The stack and the heap, where objects and variables live	236
Methods on the stack	237
Where <i>local</i> variables live	238
Where <i>instance</i> variables live	239
The miracle of object creation	240
Constructors (the code that runs when you say <i>new</i>)	241
Initializing the state of a new Duck	243
Overloaded constructors	247
Superclass constructors (constructor chaining)	250
Invoking overloaded constructors using <i>this()</i>	256
Life of an object	258
Garbage Collection (and making objects eligible)	260
Exercises and puzzles	266

10

Numbers Matter

Static variables are shared by all instances of a class.

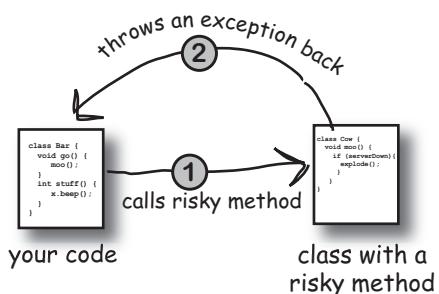


Do the Math. The Java API has methods for absolute value, rounding, min/max, etc. But what about formatting? You might want numbers to print exactly two decimal points, or with commas in all the right places. And you might want to print and manipulate dates, too. And what about parsing a String into 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*.

Math class (do you really need an instance of it?)	274
static methods	275
static variables	277
Constants (static final variables)	282
Math methods (random(), round(), abs(), etc.)	286
Wrapper classes (Integer, Boolean, Character, etc.)	287
Autoboxing	289
Number formatting	294
Date formatting and manipulation	301
Static imports	307
Exercises and puzzles	310

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.



Making a music machine (the BeatBox)	316
What if you need to call risky code?	319
Exceptions say "something bad may have happened..."	320
The compiler guarantees (it <i>checks</i>) that you're aware of the risks	321
Catching exceptions using a <i>try/catch</i> (skateboarder)	322
Flow control in <i>try/catch</i> blocks	326
The <i>finally</i> block (no matter what happens, turn off the oven!)	327
Catching multiple exceptions (the order matters)	329
Declaring an exception (just duck it)	335
Handle or declare law	337
Code Kitchen (making sounds)	339
Exercises and puzzles	348

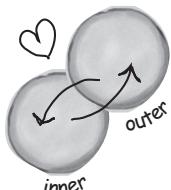
12 A Very Graphic Story

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).



Your first GUI	355
Getting a user event	357
Implement a listener interface	358
Getting a button's ActionEvent	360
Putting graphics on a GUI	363
Fun with paintComponent()	365
The Graphics2D object	366
Putting more than one button on a screen	370
Inner classes to the rescue (make your listener an inner class)	376
Animation (move it, paint it, move it, paint it, move it, paint it...)	382
Code Kitchen (painting graphics with the beat of the music)	386
Exercises and puzzles	394

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.



Swing Components	400
Layout Managers (they control size and placement)	401
Three Layout Managers (border, flow, box)	403
BorderLayout (cares about five regions)	404
FlowLayout (cares about the order and preferred size)	408
BoxLayout (like flow, but can stack components vertically)	411
JTextField (for single-line user input)	413
JTextArea (for multi-line, scrolling text)	414
JCheckBox (is it selected?)	416
JList (a scrollable, selectable list)	417
Code Kitchen (The Big One - building the BeatBox chat client)	418
Exercises and puzzles	424

14

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.

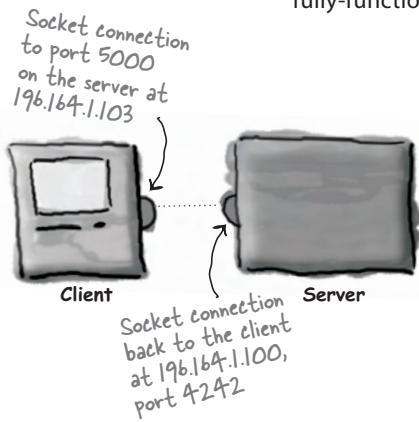
Any questions?



Saving object state	431
Writing a serialized object to a file	432
Java input and output streams (connections and chains)	433
Object serialization	434
Implementing the Serializable interface	437
Using transient variables	439
Deserializing an object	441
Writing to a text file	447
java.io.File	452
Reading from a text file	454
Splitting a String into tokens with split()	458
CodeKitchen	462
Exercises and puzzles	466

15 Make a Connection

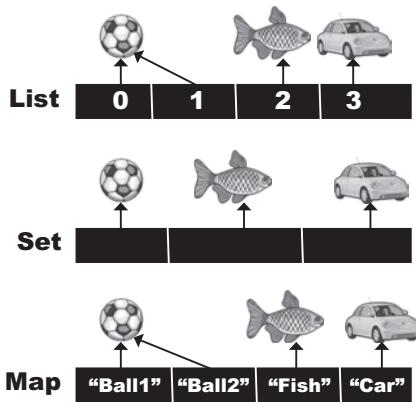
Connect with the outside world. It's easy. All the low-level networking details are taken care of by classes in the `java.net` library. One of Java's best features is that sending and receiving data over a network is really just I/O with a slightly different connection stream at the end of the chain. 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*?



Chat program overview	473
Connecting, sending, and receiving	474
Network sockets	475
TCP ports	476
Reading data from a socket (using <code>BufferedReader</code>)	478
Writing data to a socket (using <code>PrintWriter</code>)	479
Writing the Daily Advice Client program	480
Writing a simple server	483
Daily Advice Server code	484
Writing a chat client	486
Multiple call stacks	490
Launching a new thread (make it, start it)	492
The <code>Runnable</code> interface (the thread's job)	494
Three states of a new <code>Thread</code> object (new, runnable, running)	495
The runnable-running loop	496
Thread scheduler (it's his decision, not yours)	497
Putting a thread to sleep	501
Making and starting two threads	503
Concurrency issues: can this couple be saved?	505
The Ryan and Monica concurrency problem, in code	506
Locking to make things atomic	510
Every object has a lock	511
The dreaded "Lost Update" problem	512
Synchronized methods (using a lock)	514
Deadlock!	516
Multithreaded ChatClient code	518
Ready-bake SimpleChatServer	520
Exercises and puzzles	524

16 Data Structures

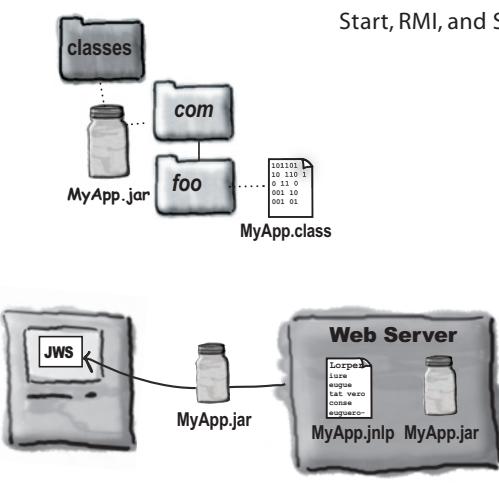
Sorting is a snap in Java. You have all the tools for collecting and manipulating your data without having to write your own sort algorithms. The Java Collections Framework has a data structure that should work for virtually anything you'll ever need to do. Want to keep a list that you can easily keep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



Collections	533
Sorting an ArrayList with Collections.sort()	534
Generics and type-safety	540
Sorting things that implement the Comparable interface	547
Sorting things with a custom Comparator	552
The collection API—lists, sets, and maps	557
Avoiding duplicates with HashSet	559
Overriding hashCode() and equals()	560
HashMap	567
Using wildcards for polymorphism	574
Exercises and puzzles	576

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.

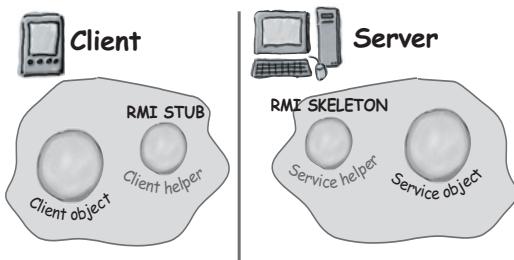


Deployment options	582
Keep your source code and class files separate	584
Making an executable JAR (Java ARchives)	585
Running an executable JAR	586
Put your classes in a package!	587
Packages must have a matching directory structure	589
Compiling and running with packages	590
Compiling with -d	591
Making an executable JAR (with packages)	592
Java Web Start (JWS) for deployment from the web	597
How to make and deploy a JWS application	600
Exercises and puzzles	601

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.

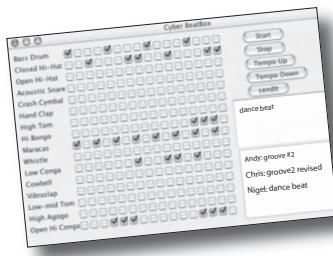


Java Remote Method Invocation (RMI), hands-on, <i>very</i> detailed	614
Servlets (a quick look)	625
Enterprise JavaBeans (EJB), a <i>very</i> quick look	631
Jini, the best trick of all	632
Building the really cool universal service browser	636
The End	648

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.



BeatBoxFinal (client code)	650
MusicServer (server code)	657

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.

Top Ten List	660
--------------	-----

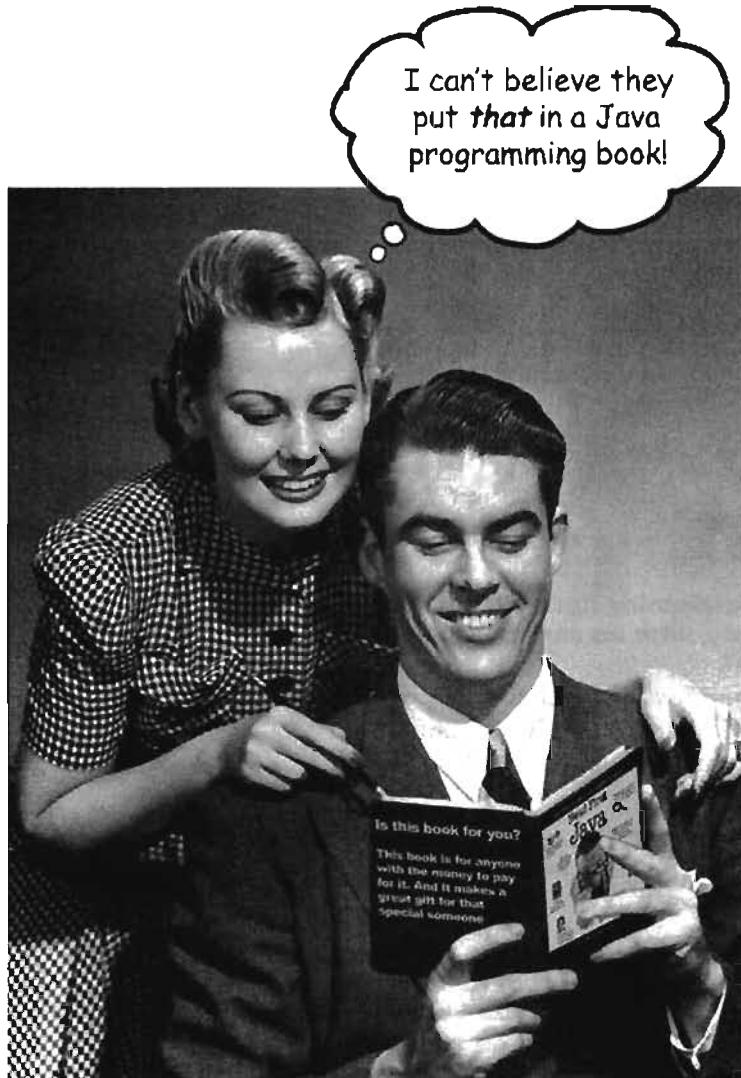
i

Index

677

how to use this book

Intro



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

Who is this book for?

If you can answer "yes" to *all* of these:

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

this book is for you.

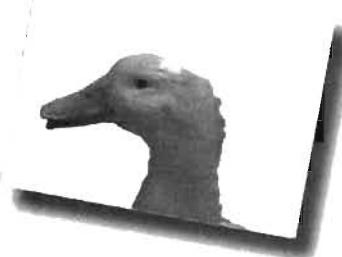
This is NOT a reference book. Head First Java is a book designed for learning, not an encyclopedia of Java facts.

Who should probably back away from this book?

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

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

this book is *not* for you.



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

We know what you're thinking.

"How can *this* be a serious Java programming book?"

"What's with all the graphics?"

"Can I actually *learn* it this way?"

"Do I smell pizza?"



And we know what your brain is thinking.

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

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

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

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

Neurons fire. Emotions crank up. *Chemicals surge*

And that's how your brain knows...

This must be Important! Don't forget it!

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

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

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



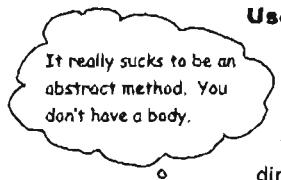
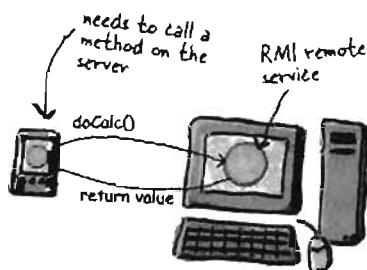
We think of a "Head First Java" reader as a learner.

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

Some of the Head First learning principles:



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



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



No method body!
End it with a connection ↑

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.

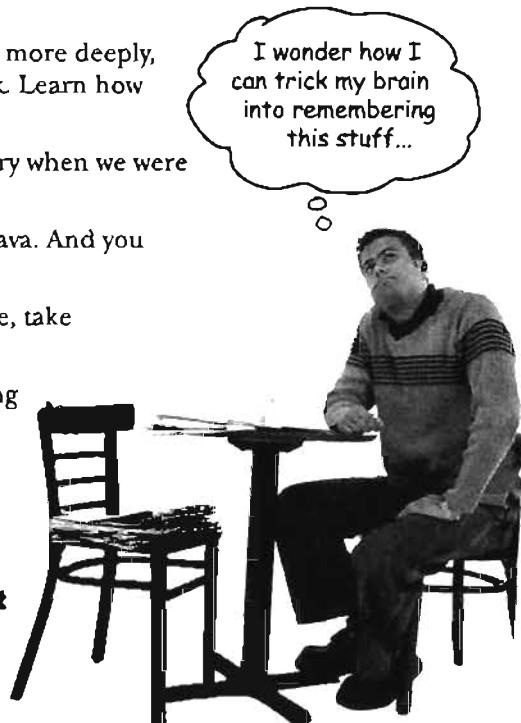
So just how **DO** you get your brain to treat Java like it was a hungry tiger?

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 make sense of how the words and picture relate, and this causes more neurons to fire. More neurons firing = more chances for your brain to *get* that this is something worth paying attention to, and possibly recording.

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

But pictures and conversational style are just the beginning.



Here's what WE did:

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

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

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

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

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

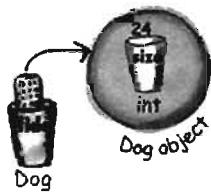
We used **multiple learning styles**, because you might prefer step-by-step procedures, while someone else wants to understand the big picture first, while someone else just wants to see a code example. But regardless of your own learning preference, *everyone* benefits from seeing the same content represented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

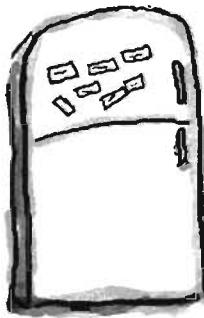
We include content for **both sides of your brain**, because the more of your brain you engage, the more likely you are to learn and remember, and the longer you can stay focused. Since working one side of the brain often means giving the other side a chance to rest, you can be more productive at learning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

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



➊ 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.

➋ 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.

➌ 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.

➍ 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.

➎ 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.

➏ 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.

➐ 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.

➑ 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.

➒ Feel something!

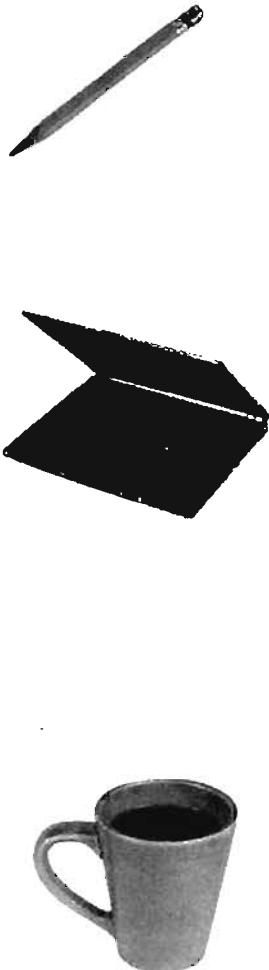
Your brain needs to know that this *matters*. Get involved with the stories. Make up your own captions for the photos. Groaning over a bad joke is *still* better than feeling nothing at all.

➓ 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 headfirstjava.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.



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 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.

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

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

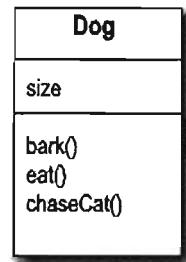
The 'Sharpen Your Pencil' exercises don't have answers.

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)

The code examples are as lean as possible

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

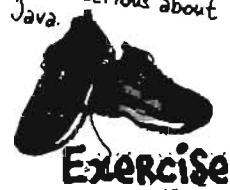
We use a simpler, modified faux-UML



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

Sharpen your pencil

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



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



Technical Editors

"Credit goes to all, but mistakes are the sole responsibility 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 SCPJ 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](http://javaranch.com). He holds the SCPJ, SCJD, SCBCD, SCWCD, and SCDJWS certifications. He has also had the opportunity to serve as a co-author for Whizlabs SCBCD Exam Simulator.

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

credit Other people to blame:

At O'Reilly:

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

Our intrepid beta testers and reviewer team:

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

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

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

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



Rodney J.
Woodruff



James Cubeta Terri Cubeta



Ira Becker



John Nyquist



Chris Jones

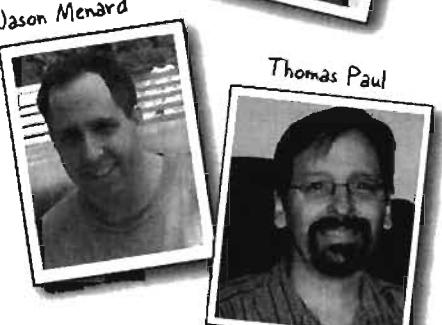
Some of our Java
expert reviewers...



Jeff Cumps



Johannes de Jong



Jason Menard

Thomas Paul



Marilyn de
Queiroz

still more acknowledgements

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:

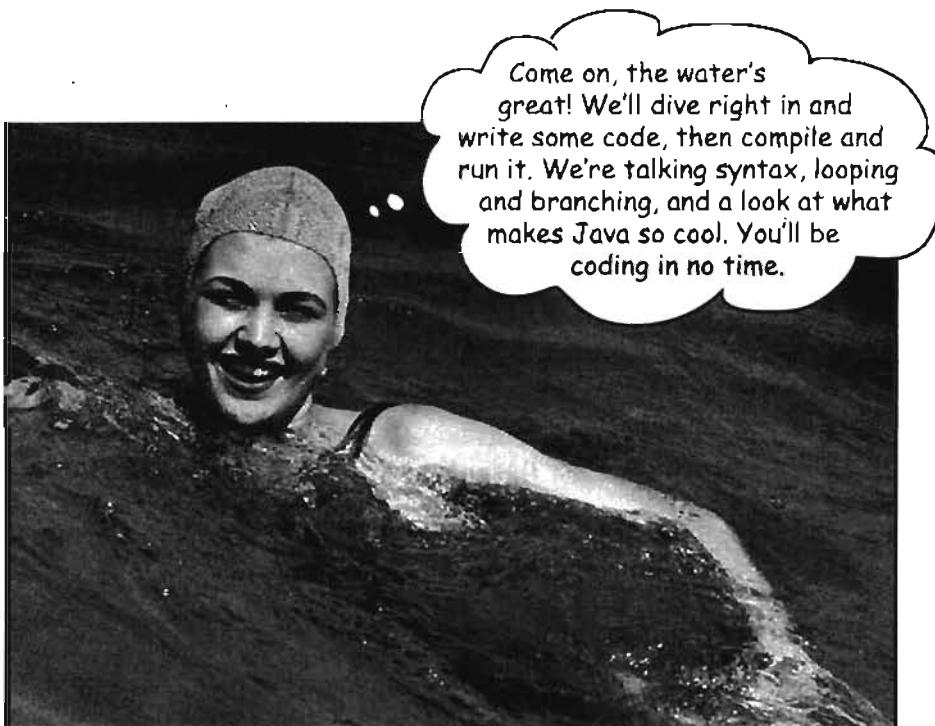
Paul Wheaton, the javaranch Trail Boss for supporting thousands of Java learners.
Solveig Haugland, mistress of J2EE and author of "Dating Design Patterns".
Authors Dori Smith and Tom Negrino (backupbrain.com), for helping us navigate the tech book world.
Our Head First partners in crime, Eric Freeman 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



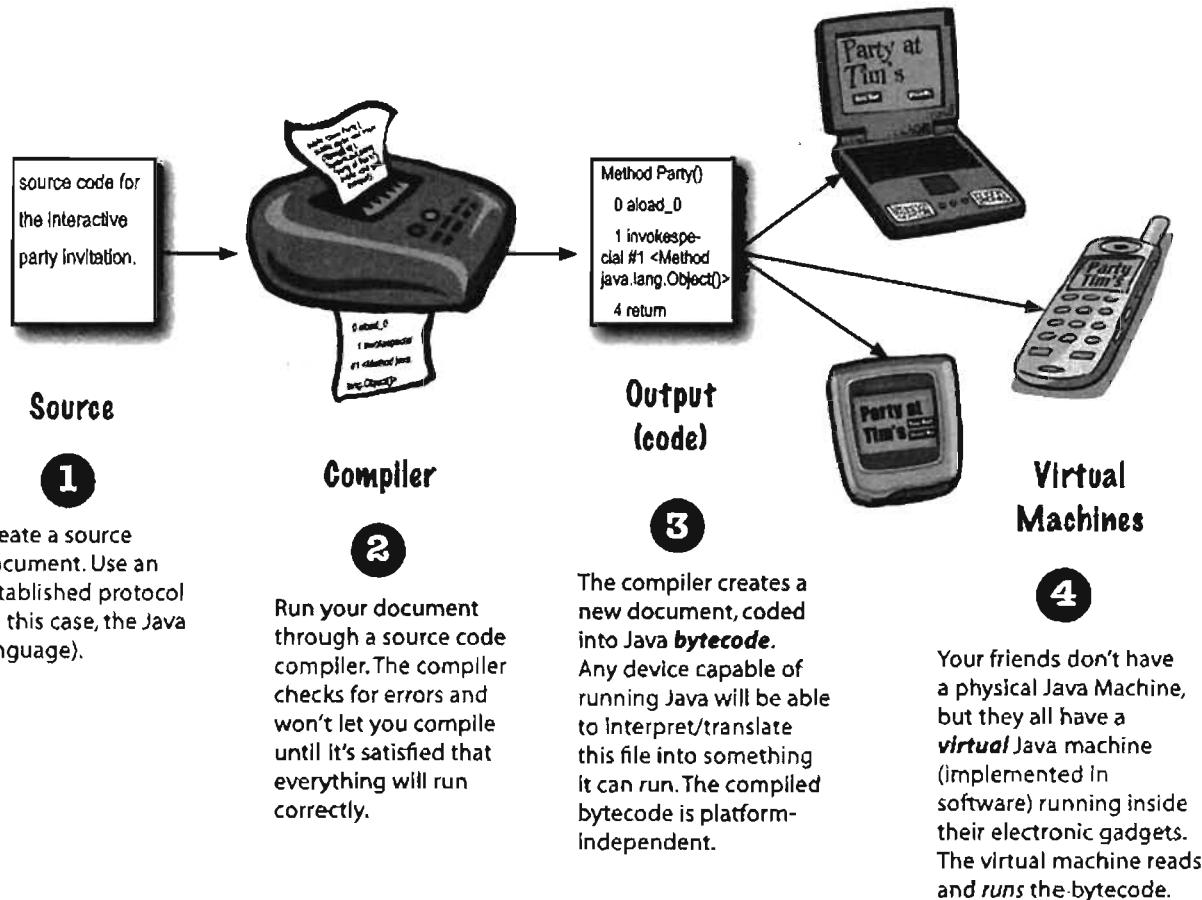
Come on, the water's great! We'll dive right in and write some code, then compile and run it. We're talking syntax, looping and branching, and a look at what makes Java so cool. You'll be coding in no time.

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



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(b);
        p.add(c);
    } // more code here...
}
```

Source

1

Type your source code.

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

```
File Edit Window Help Please
% 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()
0aload_0
1invokespecial #1 <Method
java.lang.Object()
4return

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

Output (code)

3

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

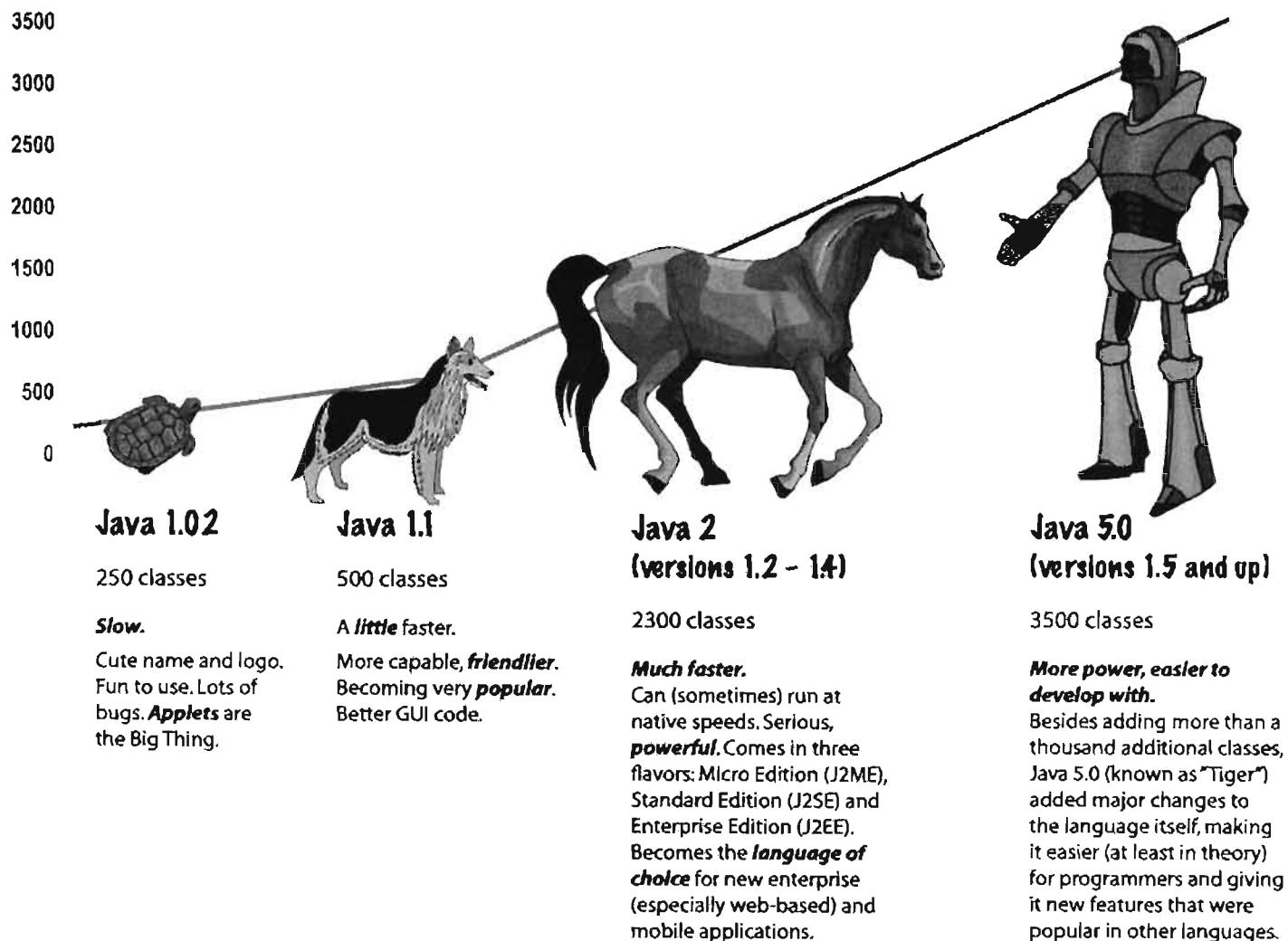
```
File Edit Window Help Swear
% java Party
ooo
Party at Tim's!
You Bet Shoot Me
```

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.)

Classes in the Java standard library

Sharpen your pencil



**Look how easy it
is to write Java.**

```
int size = 27;  
String name = "Fido";  
Dog myDog = new Dog(name, size);  
x = size - 5;  
if (x < 15) myDog.bark(8);  
  
while (x > 3) {  
    myDog.play();  
}  
  
int[] numList = {2,4,6,8};  
System.out.print("Hello");  
System.out.print("Dog: " + name);  
String num = "8";  
int z = Integer.parseInt(num);  
  
try {  
    readTheFile("myFile.txt");  
}  
catch(FileNotFoundException ex) {  
    System.out.print("File not found.");  
}
```

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 was 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...

why Java is cool

Sharpen your pencil answers

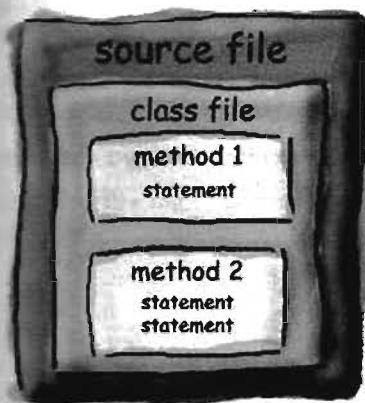
Look how easy it
is to write Java.

```
int size = 27;  
String name = "Fido";  
Dog myDog = new Dog(name, size);  
x = size - 5;  
if (x < 15) myDog.bark(8);  
  
while (x > 3) {  
    myDog.play();  
}  
  
int[] numList = {2,4,6,8};  
System.out.print("Hello");  
System.out.print("Dog: " + name);  
String num = "8";  
int z = Integer.parseInt(num);  
  
try {  
    readTheFile("myFile.txt");  
}  
catch(FileNotFoundException ex) {  
    System.out.print("File not found.");  
}
```

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

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

Code structure in Java



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

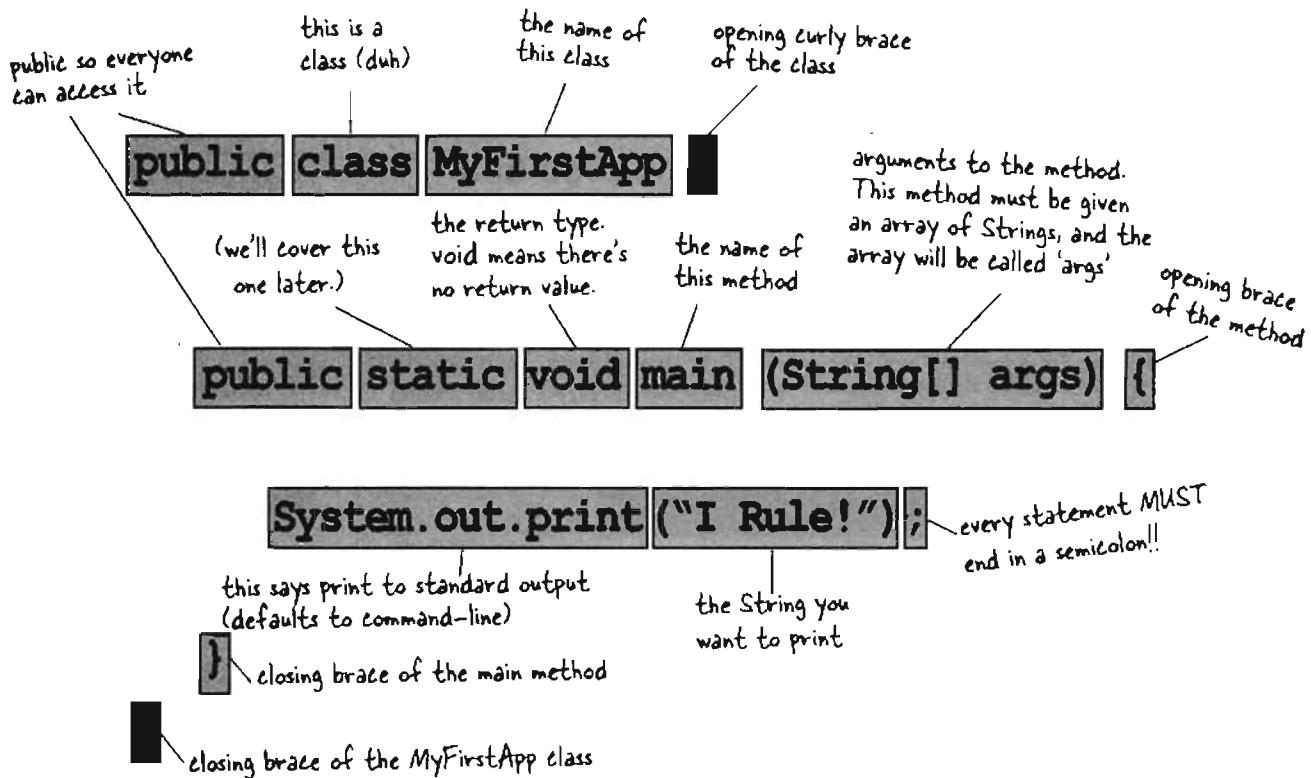
a Java class

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

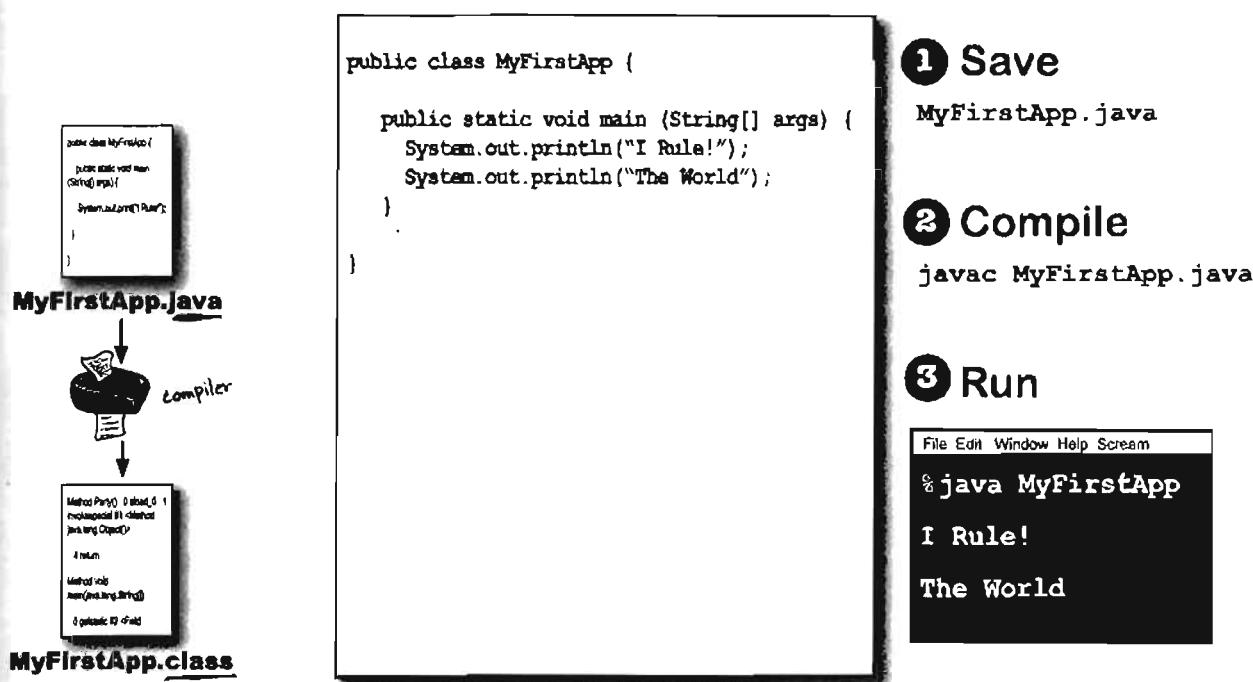
In Java, everything goes in a **class**. You'll type your source code file (with a `.java` extension), then compile it into a new class file (with a `.class` extension). When you run your program, you're really running a **class**.

Running a program means telling the Java Virtual Machine (JVM) to "Load the `Hello` 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.



statements, looping, branching

What can you say in the main method?

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

Your code can tell the JVM to:

➊ do something

Statements: declarations, assignments, method calls, etc.

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

➋ do something again and again

Loops: for and while

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

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

➌ do something under this condition

Branching: if/else tests

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



Syntax Fun

★ Each statement must end in a semicolon.

x = x + 1;

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

x = 22;
// this line disturbs me

★ Most white space doesn't matter.

x = 3 ;

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

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

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

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

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



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
}
```

Java basics

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");  
    }  
}
```

```
* 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
```

this is the output

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

Sharpen your pencil

Given the output:

```
$ java DooBee
DooBeeDooBeeDo
```

Fill in the missing code:

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

serious Java app

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 backend 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

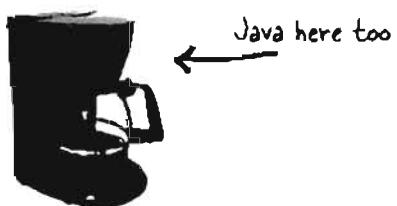
Bob's alarm clock rings at 8:30 Monday morning, just like every other weekday. But Bob had a wild weekend, and reaches for the SNOOZE button. And that's when the action starts, and the Java-enabled appliances come to life.



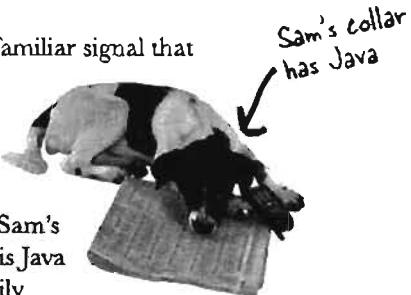
First, the alarm clock sends a message to the coffee maker* "Hey, the geek's sleeping in again, delay the coffee 12 minutes."



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

let's write a program



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", "Bir"};
```

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 net-
worked portal

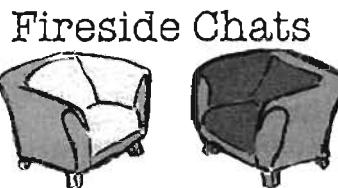
3. We need three random numbers. Java ships out-of-the-box, off-the-shelf, shrink-wrapped, and core competent with a set of math methods (for now, think of them as functions). The `random()` method returns a random number between 0 and not-quite-1, so we have to multiply it by the number of elements (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... voilà! *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 whatever, but the file doesn't *do* anything unless I'm there to run it.

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

exercise: Code Magnets



Code Magnets

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

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

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

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

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

```
    int x = 3;
```

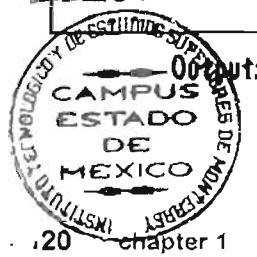
```
    x = x - 1;  
    System.out.print("-");
```

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

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

254560

BIBLIOTECA



chapter 1



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

puzzle: crossword

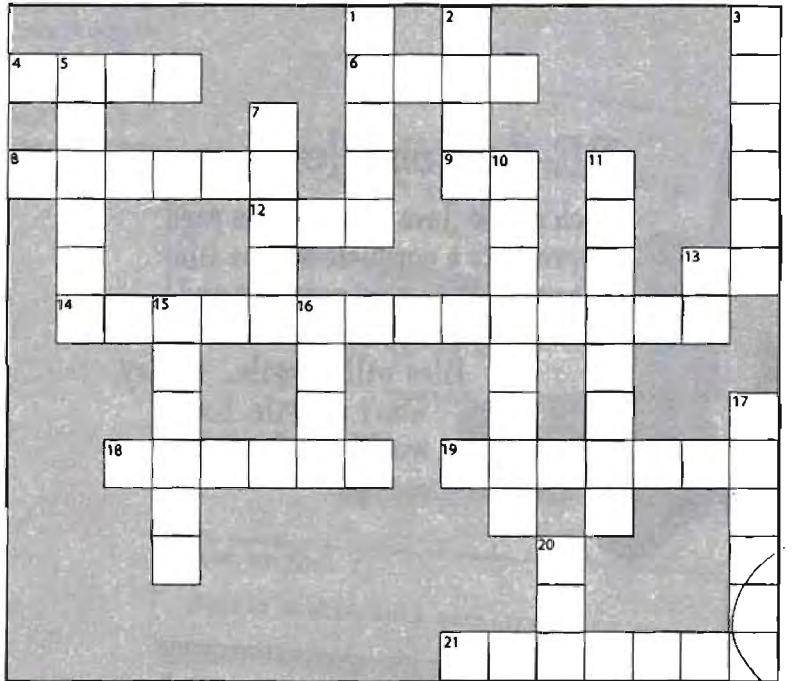


JavaCross 7.0

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 ) {
            
            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;

Possible output:

22 46

y = y + x;

11 34 59

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

02 14 26 38

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

02 14 36 48

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

11 21 32 42 53

00 11 23 36 410

02 14 25 36 47

puzzle: Pool Puzzle



Pool Puzzle

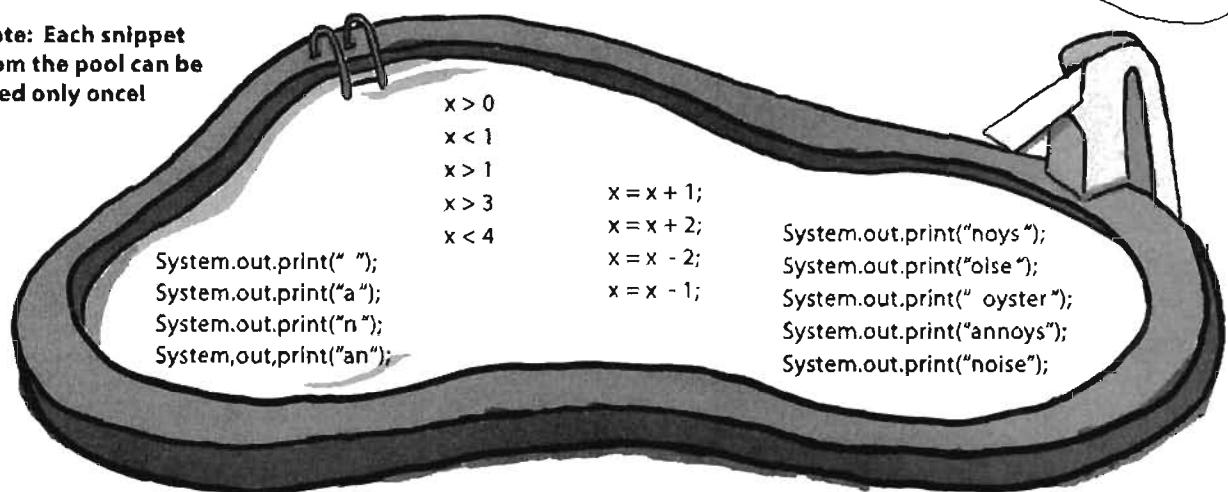


Your **job** is to take code snippets from the pool and place them into the blank lines in the code. You may **not** use the same snippet more than once, and you won't need to use all the snippets. Your **goal** is to make a class that will compile and run and produce the output listed. Don't be fooled—this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

Output

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

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



```
class PoolPuzzleOne {  
    public static void main(String [] args) {  
        int x = 0;  
  
        while ( _____ ) {  
  
            if ( x < 1 ) {  
                _____  
            }  
            _____  
  
            if ( _____ ) {  
                _____  
            }  
            _____  
            if ( x == 1 ) {  
                _____  
            }  
            if ( _____ ) {  
                _____  
            }  
            System.out.println("");  
            _____  
        }  
    }  
}
```

dive In A Quick Dip

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 Post
java Shuffle1
a-b c-d
```

```
class Exerciselb {
```

```
    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!
```

```
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!
```

```
class Exerciselb {
```

```
    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 in-
} side a method. It can't just be
} hanging out inside the class.
```

puzzle answers



```

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

            x = x + 1;
        }
    }
}

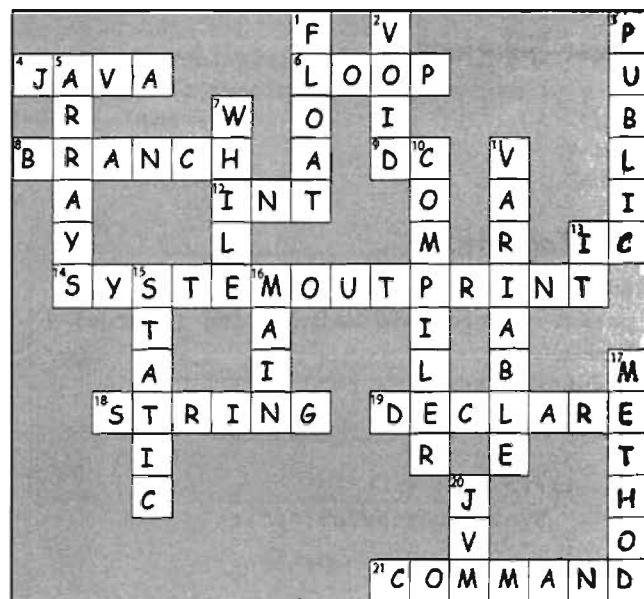
```

cmd

```

% 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:	Possible output:
y = x - y;	22 46
y = y + x;	11 34 59
y = y + 2; if(y > 4) { y = y - 1; }	02 14 26 38 02 14 36 48
x = x + 1; y = y + x;	00 11 21 32 42
if (y < 5) { x = x + 1; if (y < 3) { x = x - 1; } } y = y + 2;	11 21 32 42 53 00 11 23 36 410 02 14 25 36 47

2 classes and objects

A Trip to Objectville



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

once upon a time in Objectville

Chair Wars (or How Objects Can Change Your Life)

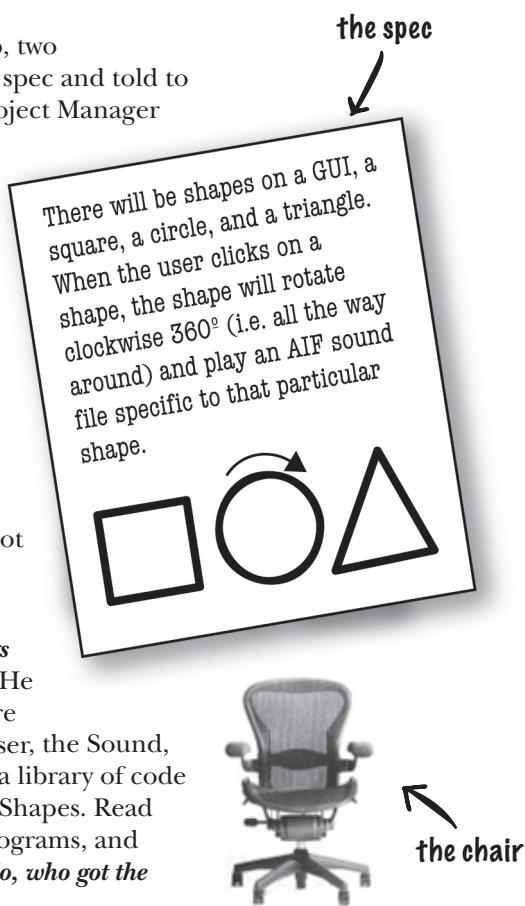


nce upon a time in a software shop, two programmers were given the same spec and told to “build it”. The Really Annoying Project Manager forced the two coders to compete,

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?*”



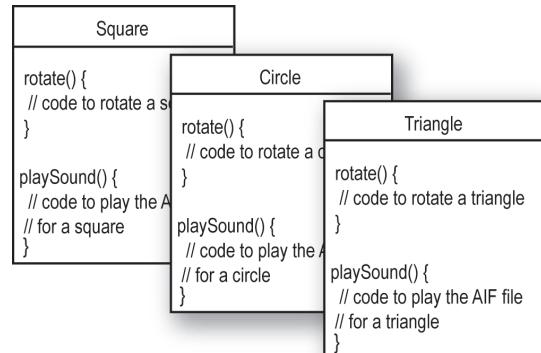
In Larry's cube

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

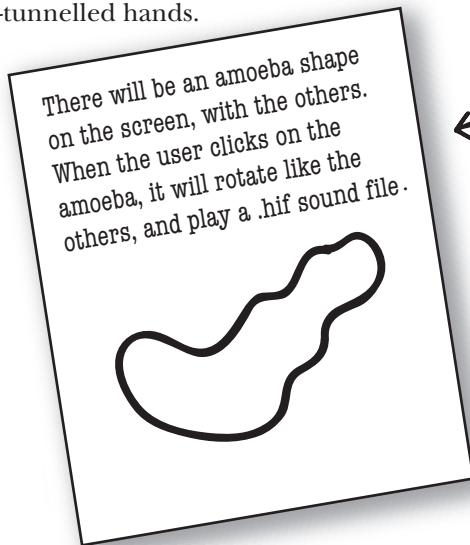


Larry thought he'd nailed it. He could almost feel the rolled steel of the Aeron beneath his...

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.

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

once upon a time in Objectville

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..."

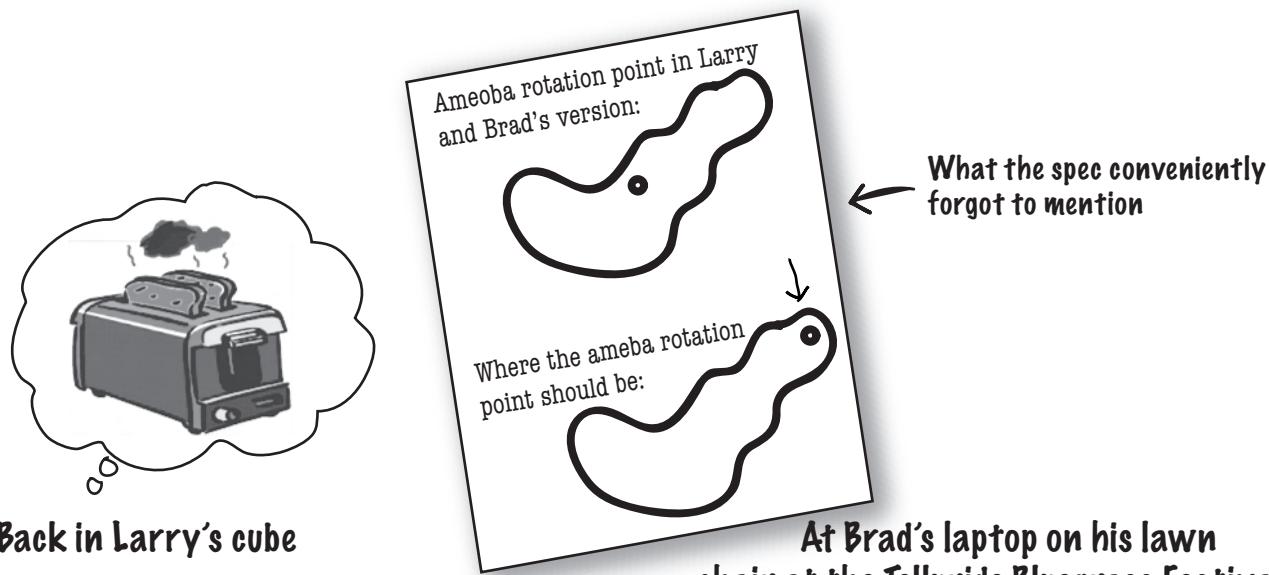
Turns out, both programmers had written their rotate code like this:

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?*"



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.

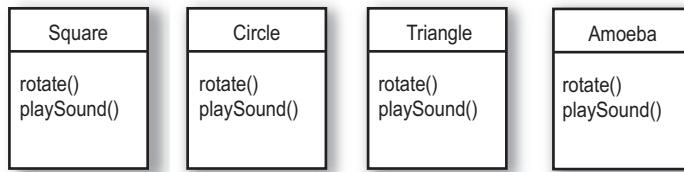
BRAD: It's a *method*, not a *procedure*. And they're *classes*, not *things*.

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)

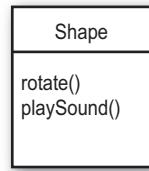


1

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.

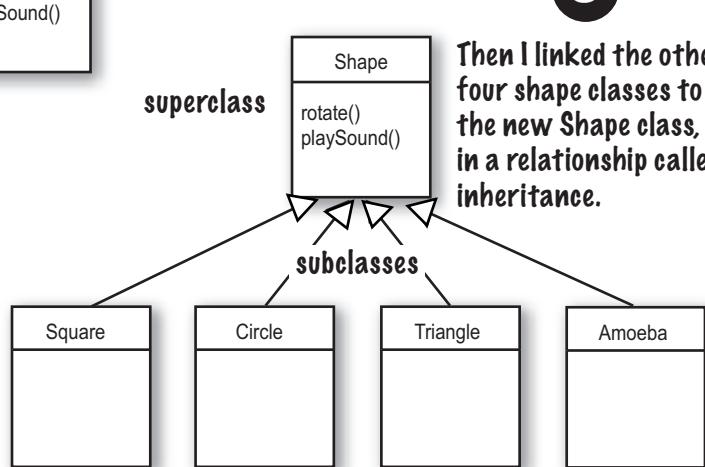


3

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*.



once upon a time in Objectville

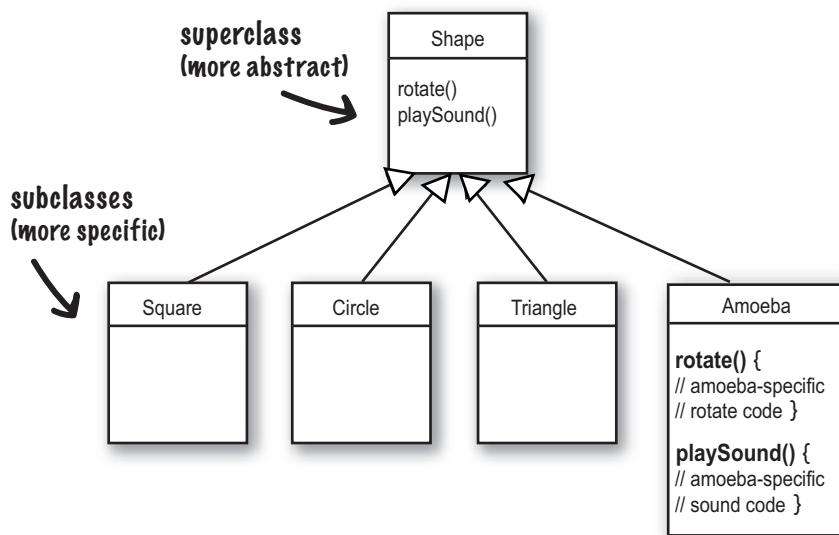
What about the Amoeba rotate()?

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.



4

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

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

Overriding methods

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**.

I know how a Shape is supposed to behave. Your job is to tell me **what** to do, and my job is to make it happen. Don’t you worry your little programmer head about **how** I do it.

I can take care of myself. I know how an Amoeba is supposed to rotate and play a sound.



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



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

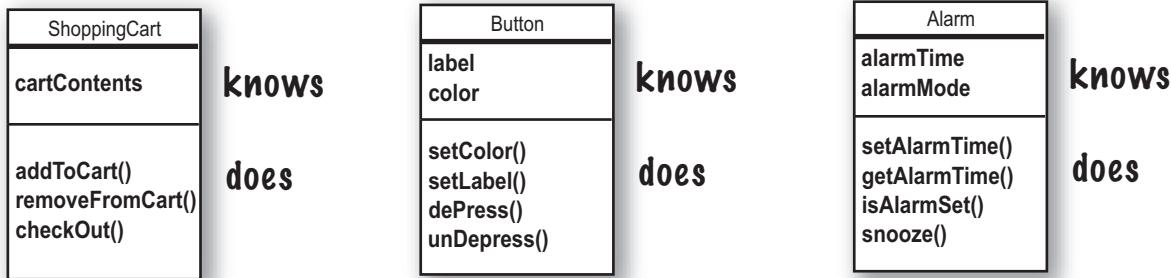


If you're stuck on an exercise, try talking about it out loud. Speaking (and hearing) activates a different part of your brain. Although it works best if you have another person to discuss it with, pets work too. That's how our dog learned polymorphism.

thinking about objects

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

instance variables
(state)
methods
(behavior)

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**.

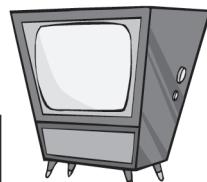
Things an object can *do* are called **methods**. When you design a class, you think about the data an object will need to know about itself, and you also design the methods that operate on that data. It's common for an object to have methods that read or write the values of the instance variables. For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

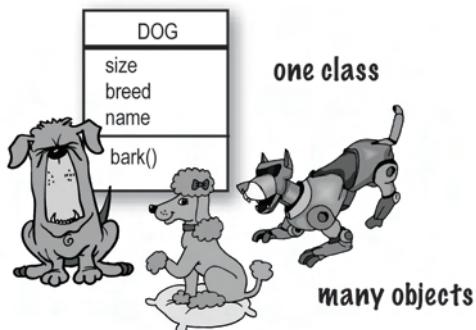
Television



instance variables

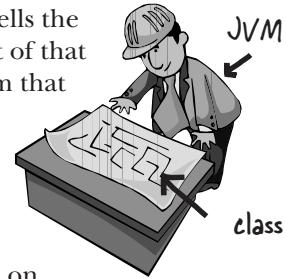
methods

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



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

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



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 objects

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.

1 Write your class

```
class Dog {  
    int size;  
    String breed;  
    String name;  
  
    void bark() {  
        System.out.println("Ruff! Ruff!");  
    }  
}
```

instance variables
a method



The Dot Operator (.)

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

// make a new object

Dog d = new Dog();

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

d.bark();

// set its size using the
// dot operator

d.size = 40;

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

```
class DogTestDrive {  
    public static void main (String[] args) {  
        Dog d = new Dog(); ← make a Dog object  
        d.size = 40; ← use the dot operator(.)  
        d.bark(); ← to set the size of the Dog  
        and to call its bark() method  
    }  
}
```

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  
    }  
}
```

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



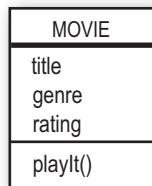
```

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;
    }
}

```



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	<table border="1"> <tr><td>title</td></tr> <tr><td>genre</td></tr> <tr><td>rating</td></tr> </table>	title	genre	rating
title				
genre				
rating				
object 2	<table border="1"> <tr><td>title</td></tr> <tr><td>genre</td></tr> <tr><td>rating</td></tr> </table>	title	genre	rating
title				
genre				
rating				
object 3	<table border="1"> <tr><td>title</td></tr> <tr><td>genre</td></tr> <tr><td>rating</td></tr> </table>	title	genre	rating
title				
genre				
rating				

get the heck out of main

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:

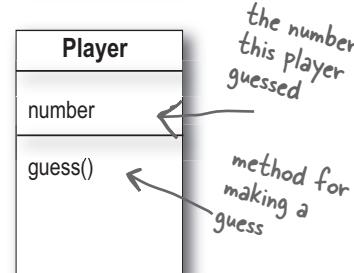
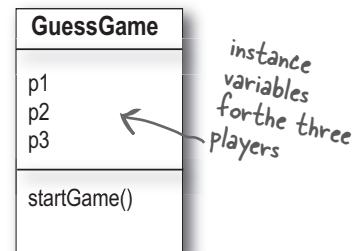
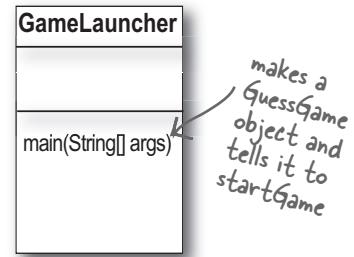
The guessing game involves a 'game' object and three 'player' objects. The game generates a random number between 0 and 9, and the three player objects try to guess it. (We didn't say it was a really *exciting* game.)

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

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

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

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

            guessp1 = p1.number;
            System.out.println("Player one guessed " + guessp1);
            guessp2 = p2.number;
            System.out.println("Player two guessed " + guessp2);
            guessp3 = p3.number;
            System.out.println("Player three guessed " + guessp3);

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

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

Guessing Game

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();  
    }  
}
```



Java takes out the Garbage

Each time an object is created in Java, it goes into an area of memory known as **The Heap**.

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

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

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

there are no Dumb Questions

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

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

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

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

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

A: A Java program is a pile of classes (or at least one class). In a Java application, one of the classes must have a main method, used to start-up the program. So as a programmer, you write one or more classes. And those classes are what you deliver. If the end-user doesn't have a JVM, then you'll also need to include that with your application's classes, so that they can run your program. There are a number of 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.

exercise: Be the Compiler



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();  
        }  
    }  
}
```

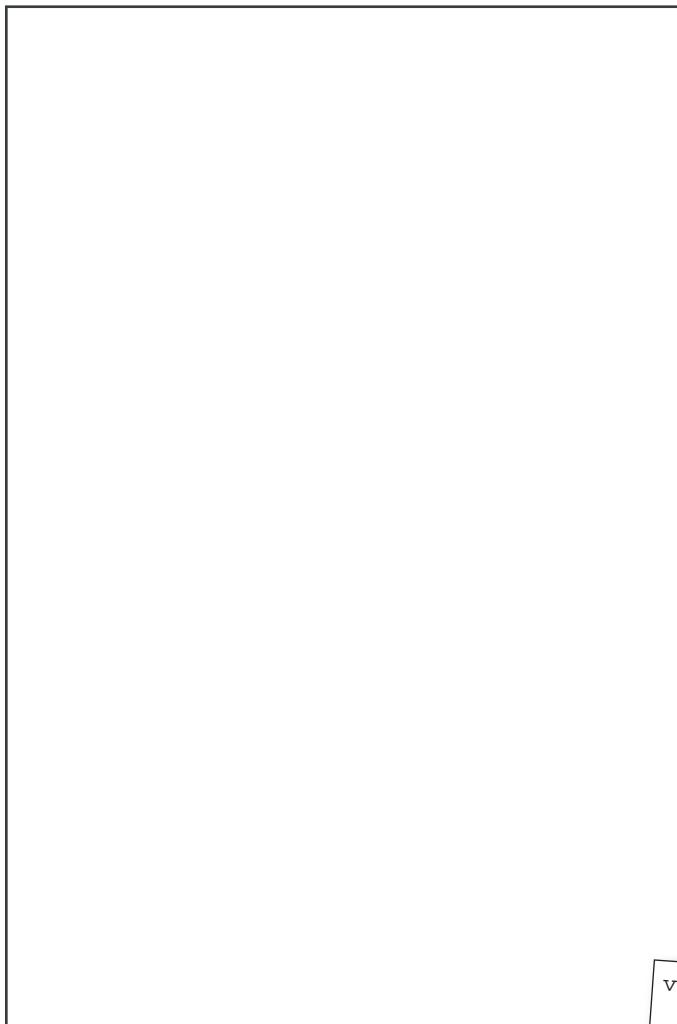
B

```
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.



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

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");
}
```

puzzle: Pool Puzzle



Pool Puzzle



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

Output

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

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

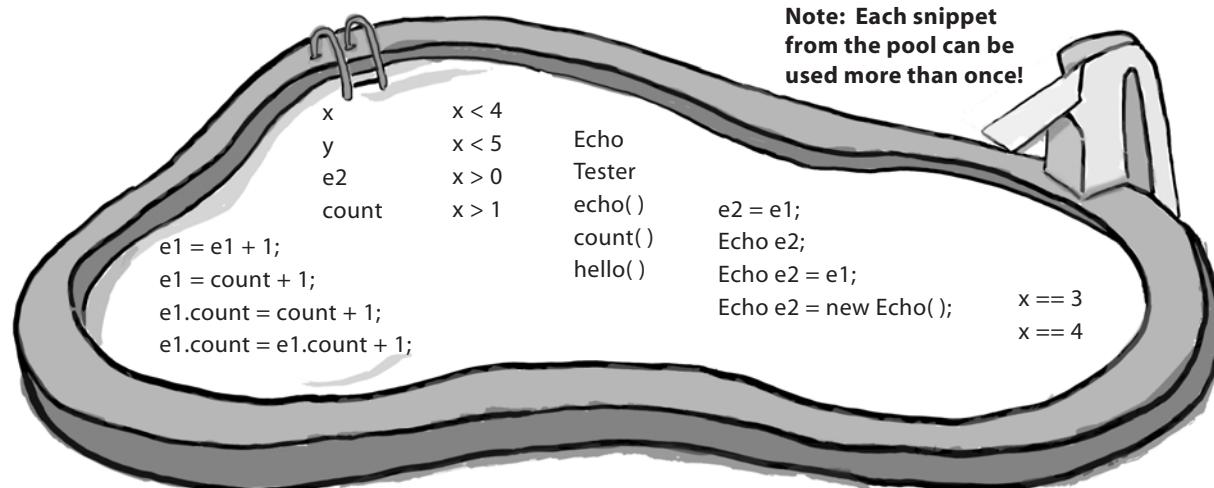
        int x = 0;
        while ( _____ ) {
            e1.hello();

            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... ");
    }
}
```

Bonus Question !

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





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



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 !**

```
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... ");
    }
}

```

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.

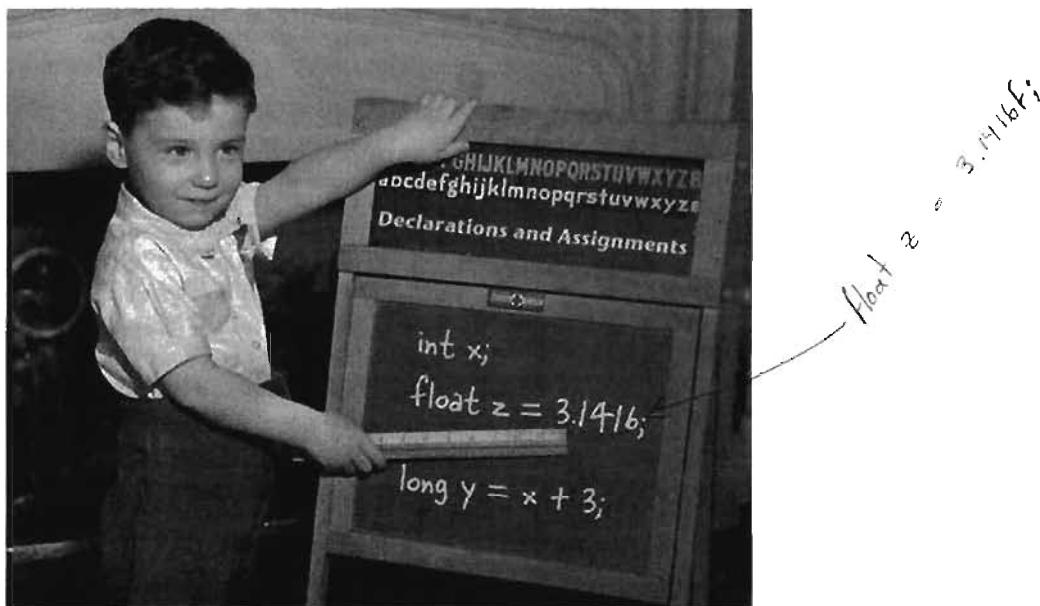
```

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

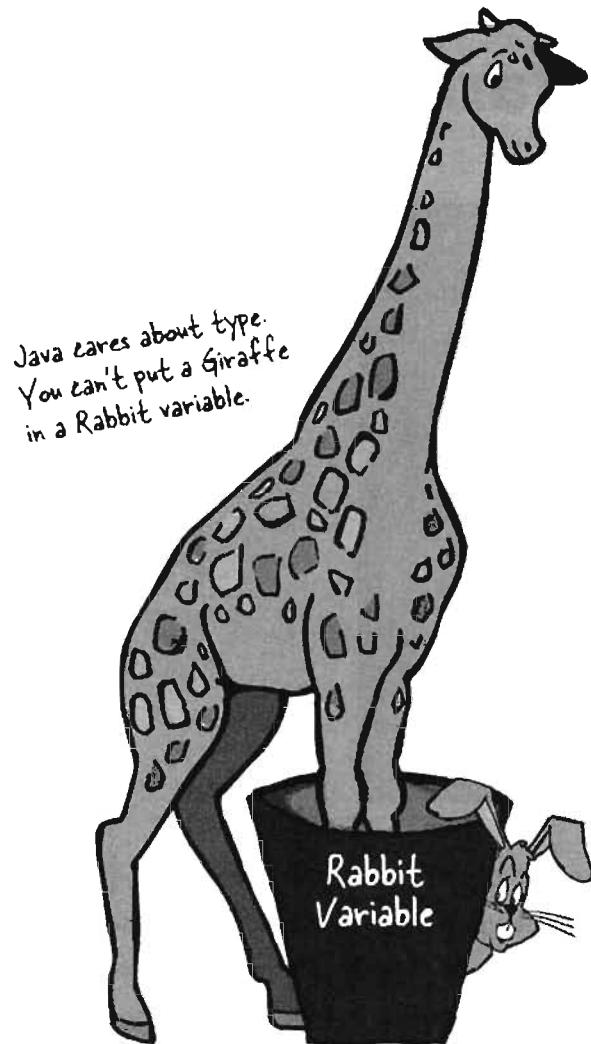
```

3 primitives and references

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."

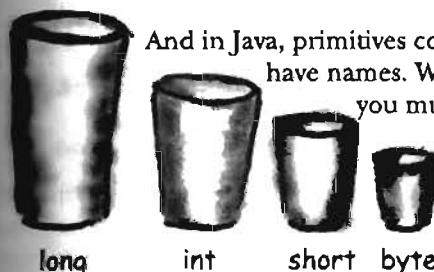
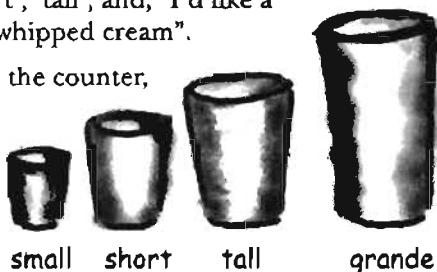
When you think of Java variables, think of cups. Coffee cups, tea cups, giant cups that hold lots and lots of beer, those big cups the popcorn comes in at the movies, cups with curvy, sexy handles, and cups with metallic trim that you learned can never, ever go in the microwave.

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.

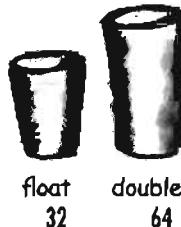
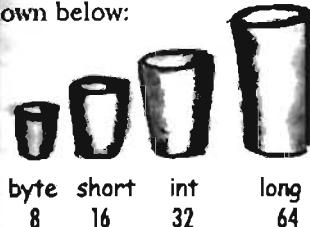
Primitives are like the cups they have at the coffeehouse. If you've been to a Starbucks, you know what we're talking about here. They come in different sizes, and each has a name like 'short', 'tall', and, "I'd like a 'grande' mocha half-caff with extra whipped cream".

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 (VM-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:

<code>int size = 32;</code>	declare an int named <i>size</i> , assign it the value 32
<code>char initial = 'j';</code>	declare a char named <i>initial</i> , assign it the value 'j'
<code>double d = 456.709;</code>	declare a double named <i>d</i> , assign it the value 456.709
<code>boolean isCrazy;</code>	declare a boolean named <i>isCrazy</i> (no assignment)
<code>isCrazy = true;</code>	assign the value <i>true</i> to the previously-declared <i>isCrazy</i>
<code>int y = x + 456;</code>	declare an int named <i>y</i> , assign it the value that is the sum of whatever <i>x</i> 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.**

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:

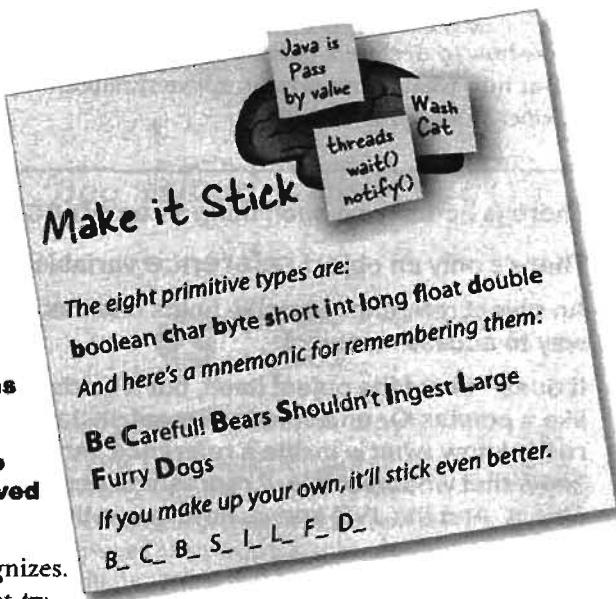
boolean char byte short int long float double

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:

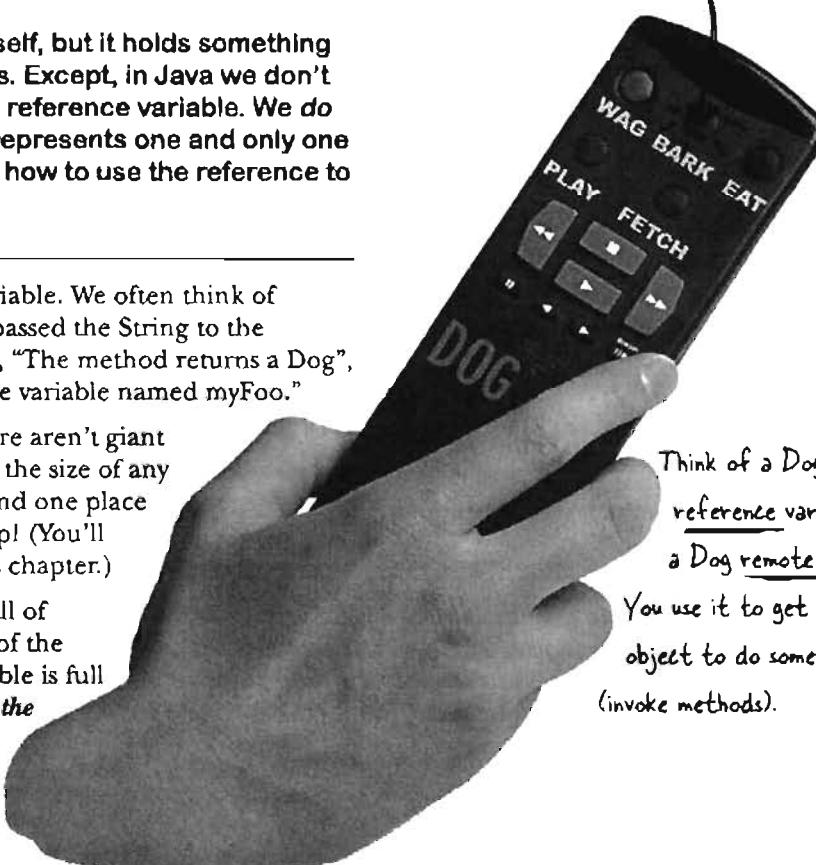
```
myDog.bark();
```

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.

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

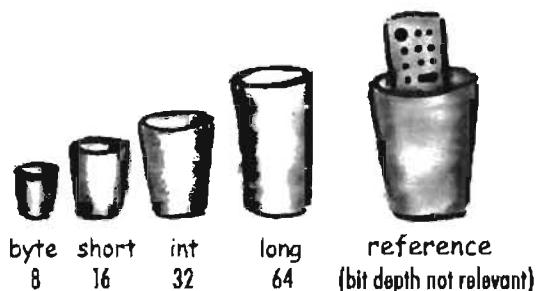
think of this

like this



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

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



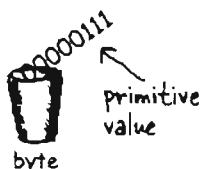
An object reference is just another variable value.

Something that goes in a cup.
Only this time, the value is a remote control.

Primitive Variable

`byte x = 7;`

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

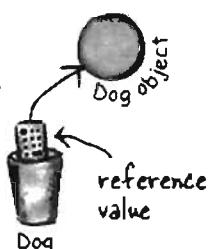


Reference Variable

`Dog myDog = new Dog();`

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

The Dog object itself does not go into the variable!



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

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

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

We don't care how many 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

`Dog myDog = new Dog();`

- 1 Declare a reference variable

`Dog myDog = new Dog();`

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



- 2 Create an object

`Dog myDog = new Dog();`

Tells the JVM to allocate space for a new `Dog` object on the heap (we'll learn a lot more about that process, especially in chapter 9.)

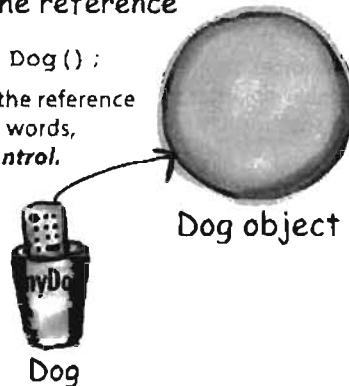


Dog object

- 3 Link the object and the reference

`Dog myDog = new Dog();`

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



object references

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."



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();

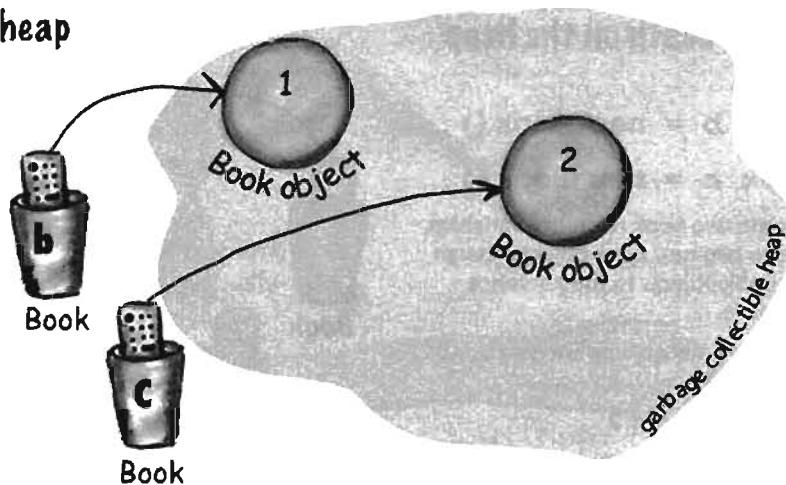
Book c = new Book();

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;

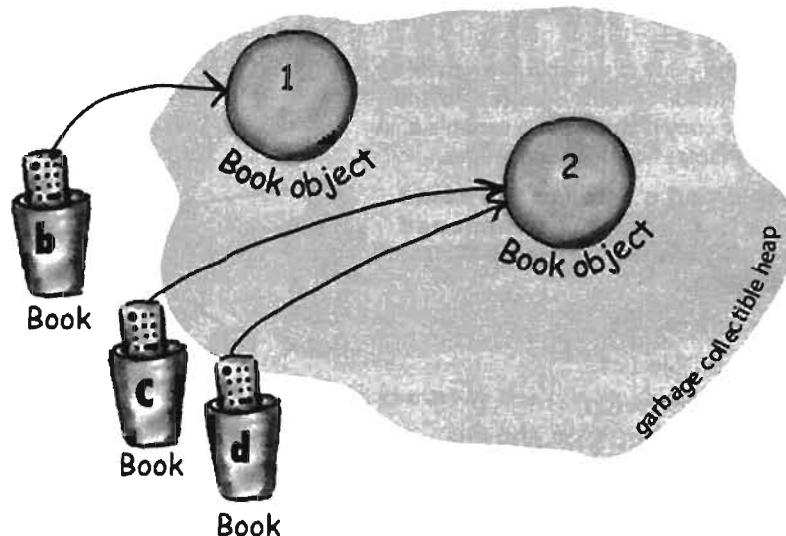
Declare a new Book reference variable. Rather than creating a new, third Book object, assign the value of variable *c* to variable *d*. But what does this mean? It's like saying, "Take the bits in *c*, make a copy of them, and stick that copy into *d*."

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



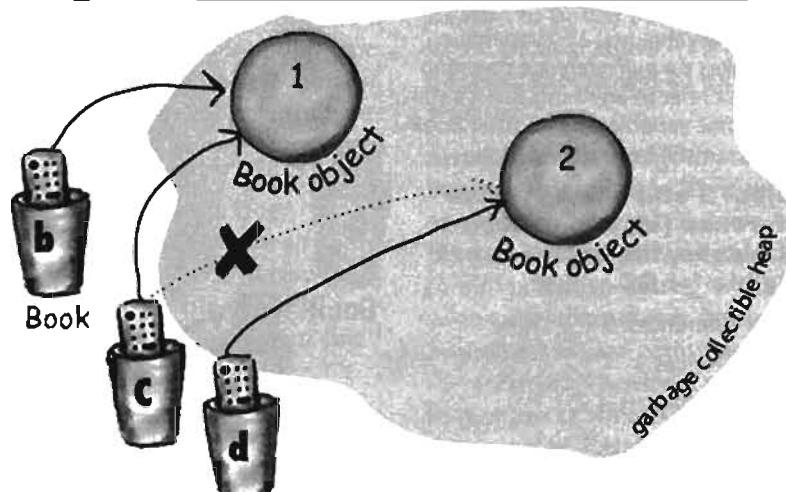
c = b;

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



objects on the heap

Life and death on the heap

`Book b = new Book();`

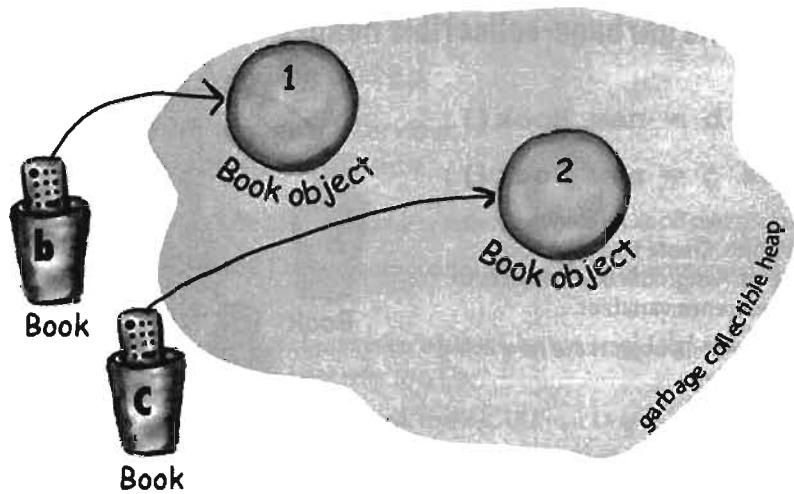
`Book c = new Book();`

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

The two book objects are now living
on the heap.

Active References: 2

Reachable Objects: 2



`b = c;`

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

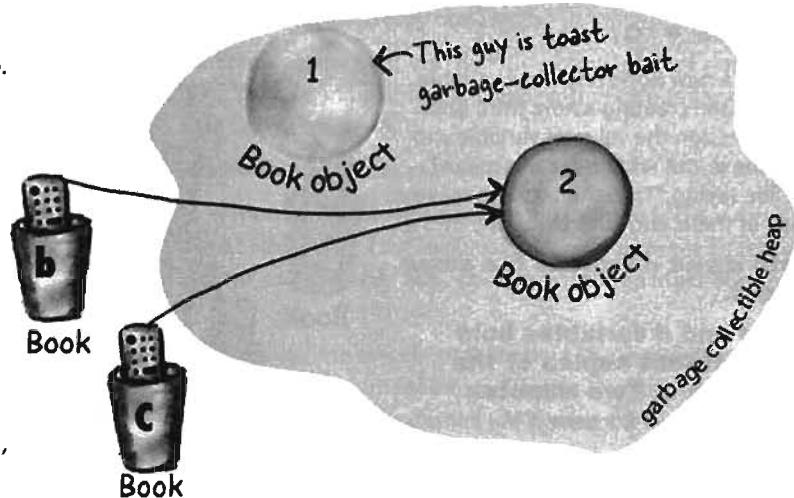
**Both `b` and `c` refer to the same
object. Object 1 is abandoned
and eligible for Garbage Collection (GC).**

Active References: 2

Reachable Objects: 1

Abandoned Objects: 1

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



`c = null;`

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

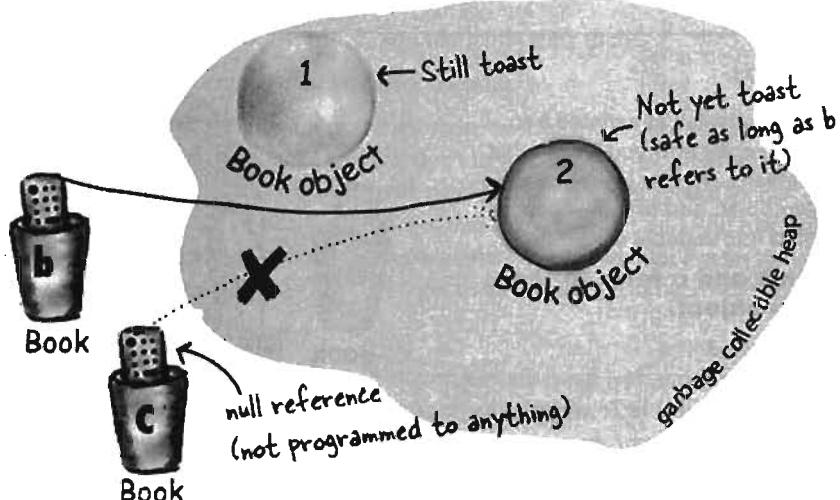
**Object 2 still has an active
reference (b), and as long
as it does, the object is not
eligible for GC.**

Active References: 1

`null` References: 1

Reachable Objects: 1

Abandoned Objects: 1



An array is like a tray of cups

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

```
int[] nums;
```

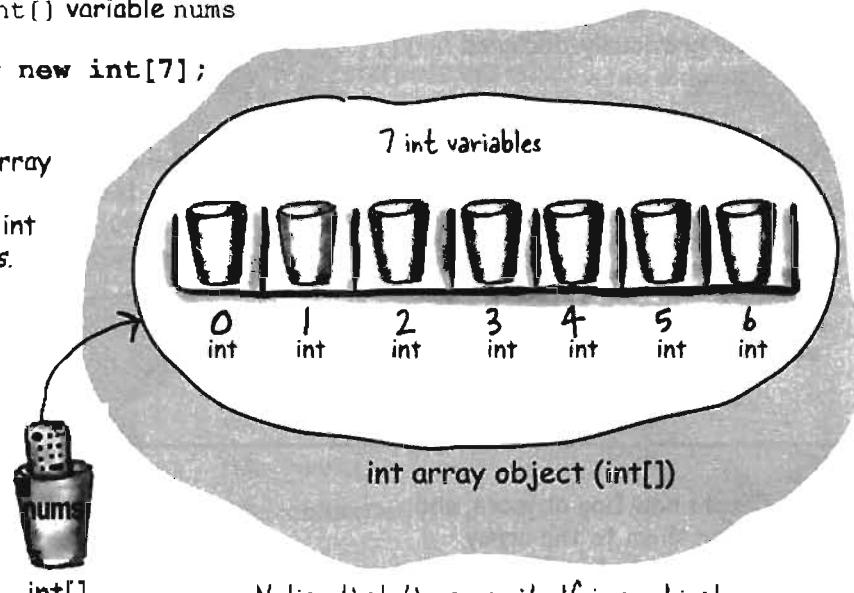
- 2** Create a new int array with a length of 7, and assign it to the previously-declared int[] variable nums

```
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;
```



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!

an array of objects

Make an array of Dogs

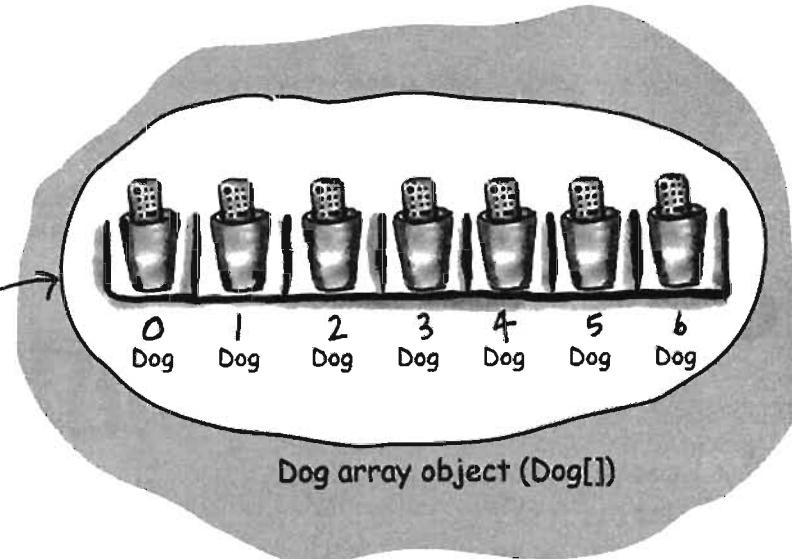
- 1 Declare a Dog array variable
`Dog[] pets;`

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

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

What's missing?

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



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

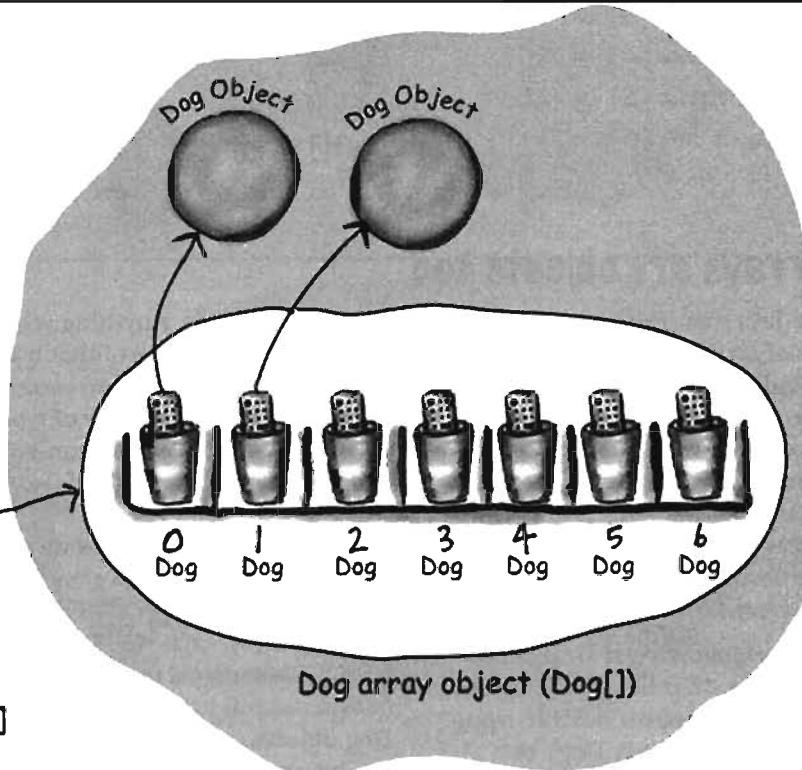
Remember, elements in a Dog array are just Dog reference variables. We still need Dogs!

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

Sharpen your pencil —

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

would make one of the dogs objects? _____





Dog
name
bark()
eat()
chaseCat()

Java cares about type.

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

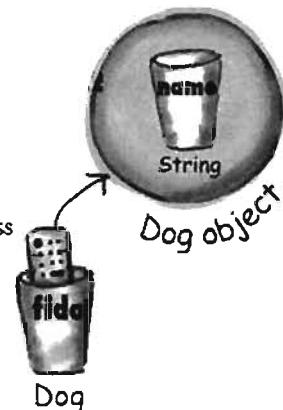
Control your Dog (with a reference variable)

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

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

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

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



What happens if the Dog is in a Dog array?

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

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

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

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

using references

```

class Dog {
    String name;
    public static void main (String[] args) {
        // make a Dog object and access it
        Dog dog1 = new Dog();
        dog1.bark();
        dog1.name = "Bart";

        // now make a Dog array
        Dog[] myDogs = new Dog[3];
        // and put some dogs in it
        myDogs[0] = new Dog();
        myDogs[1] = new Dog();
        myDogs[2] = dog1;

        // now access the Dogs using the array
        // references
        myDogs[0].name = "Fred";
        myDogs[1].name = "Marge";

        // Hmmmm... what is myDogs[2] name?
        System.out.print("last dog's name is ");
        System.out.println(myDogs[2].name);

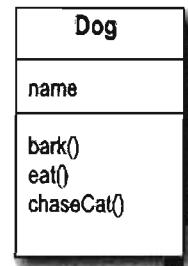
        // now loop through the array
        // and tell all dogs to bark
        int x = 0;
        while(x < myDogs.length) {
            myDogs[x].bark();
            x = x + 1;
        }
    }

    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 How?
%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.



Exercise

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 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");
        }
    }
}
```

exercise: Code Magnets



Code Magnets

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

int y = 0;

ref = index[y];

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

int ref;

while (y < 4) {

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

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

String [] islands = new String[4];

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

int [] index = new int[4];

y = y + 1;

File Edit Window Help Blank

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

class TestArrays {

public static void main(String [] args) {



Pool Puzzle



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

Output

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

Bonus Question!

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

class Triangle {
 double area;
 int height;
 int length;
 public static void main(String [] args) {

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

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

Triangle [] ta = new Triangle(4);
Triangle ta = new [] Triangle(4);
Triangle [] ta = new Triangle(4);

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 us, you probably need to draw diagrams like the ones on page 55 and 56 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).

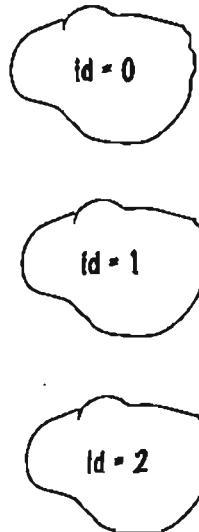
```
class HeapQuiz {
    int id = 0;
    public static void main(String [] args) {
        int x = 0;
        HeapQuiz [ ] hq = new HeapQuiz[5];
        while ( x < 3 ) {
            hq[x] = new HeapQuiz();
            hq[x].id = x;
            x = x + 1;
        }
        hq[3] = hq[1];
        hq[4]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



are here ↗

66



The case of the pilfered references

Five-Minute Mystery



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."

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



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 Blinks
$ java TestArrays
island = Fiji
island = Cozumel
island = Bermuda
island = Azores
```

```
class Books {
    String title;
    String author;
}
class BooksTestDrive {
    public static void main(String [] args) {
        Books [] myBooks = new Books[3];
        int x = 0;
        myBooks[0] = new Books();
        myBooks[1] = new Books();
        myBooks[2] = new Books();
        myBooks[0].title = "The Grapes of Java";
        myBooks[1].title = "The Java Gatsby";
        myBooks[2].title = "The Java Cookbook";
        myBooks[0].author = "bob";
        myBooks[1].author = "sue";
        myBooks[2].author = "ian";
        while (x < 3) {
            System.out.print(myBooks[x].title);
            System.out.print(" by ");
            System.out.println(myBooks[x].author);
            x = x + 1;
        }
    }
}
```

A Remember: We have to actually make the Books objects!

```
class Hobbits {
    String name;
    public static void main(String [] args) {
        Hobbits [] h = new Hobbits[3];
        int z = -1;
        while (z < 2) {
            z = z + 1;
            h[z] = new Hobbits();
            h[z].name = "bilbo";
        }
    }
}
```

B Remember: arrays start with element 0!

```
if (z == 1) {
    h[z].name = "frodo";
}
if (z == 2) {
    h[z].name = "sam";
}
System.out.print(h[z].name + " is a ");
System.out.println("good Hobbit name");
}
```



Puzzle Solutions

```

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;
    }
}

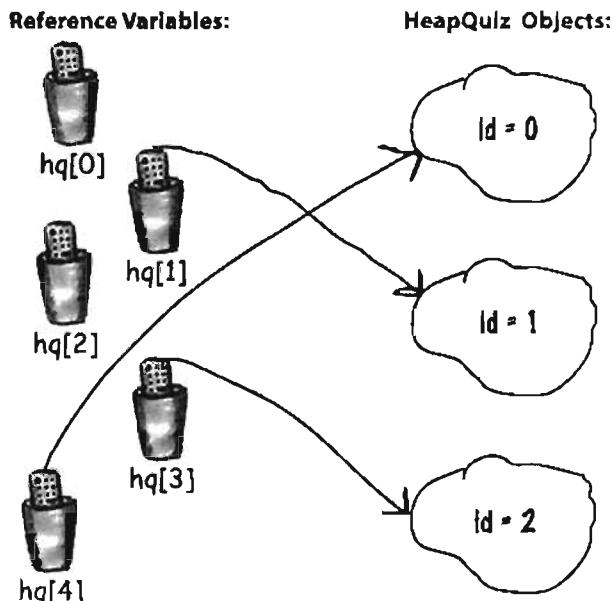
```

Kde 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

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.)



4 methods use instance variables

How Objects Behave



State affects behavior, behavior affects state. We know that objects have **state** and **behavior**, represented by **instance variables** and **methods**. But until now, we haven't looked at how state and behavior are *related*. We already know that each instance of a class (each object of a particular type) can have its own unique values for its instance variables. Dog A can have a *name* "Fido" and a *weight* of 70 pounds. Dog B is "Killer" and weighs 9 pounds. And if the Dog class has a method `makeNoise()`, well, don't you think a 70-pound dog barks a bit deeper than the little 9-pounder? (Assuming that annoying yippy sound can be considered a *bark*.) Fortunately, that's the whole point of an object—it has *behavior* that acts on its *state*. In other words, **methods use instance variable values**. Like, "if dog is less than 14 pounds, make yippy sound, else..." or "increase weight by 5". *Let's go change some state.*

objects have state and behavior

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

A class is the blueprint for an object. When you write a class, you're describing how the JVM should make an object of that type. You already know that every object of that type can have different *instance variable* values. But what about the methods?

Can every object of that type have different method behavior?

Well... *sort of.**

Every instance of a particular class has the same methods, but the methods can *behave* differently based on the value of the instance variables.

The Song class has two instance variables, *title* and *artist*. 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");
```

Song	
title	
artist	

knows

methods	(behavior)
setTitle()	
setArtist()	
play()	

does

five instances
of class Song



Calling play() on this instance
will cause "Sing" to play.
t2.play();

Calling play() on this instance
will cause "My Way" to play.
(but not the Sinatra one)

*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.

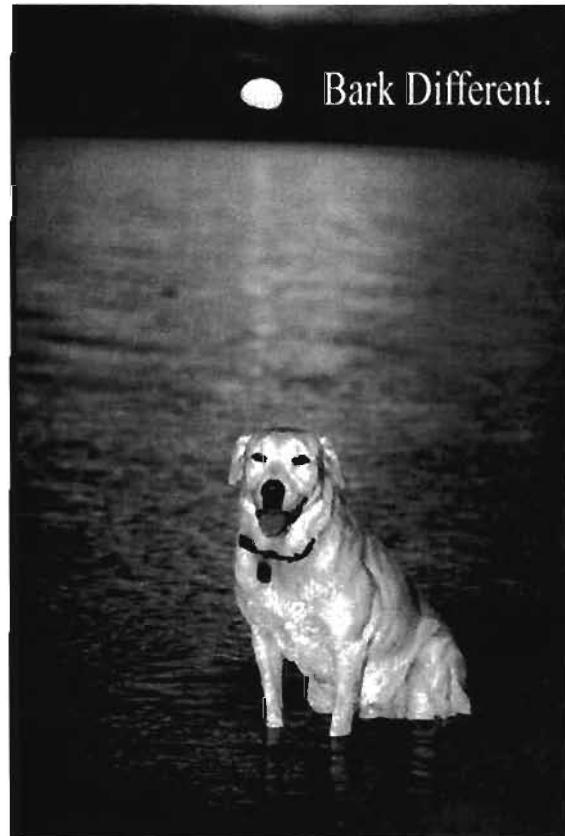
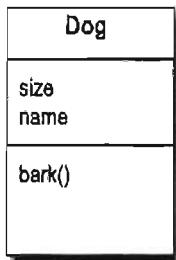
```
class Dog {
    int size;
    String name;

    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!
  
```

method parameters

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:

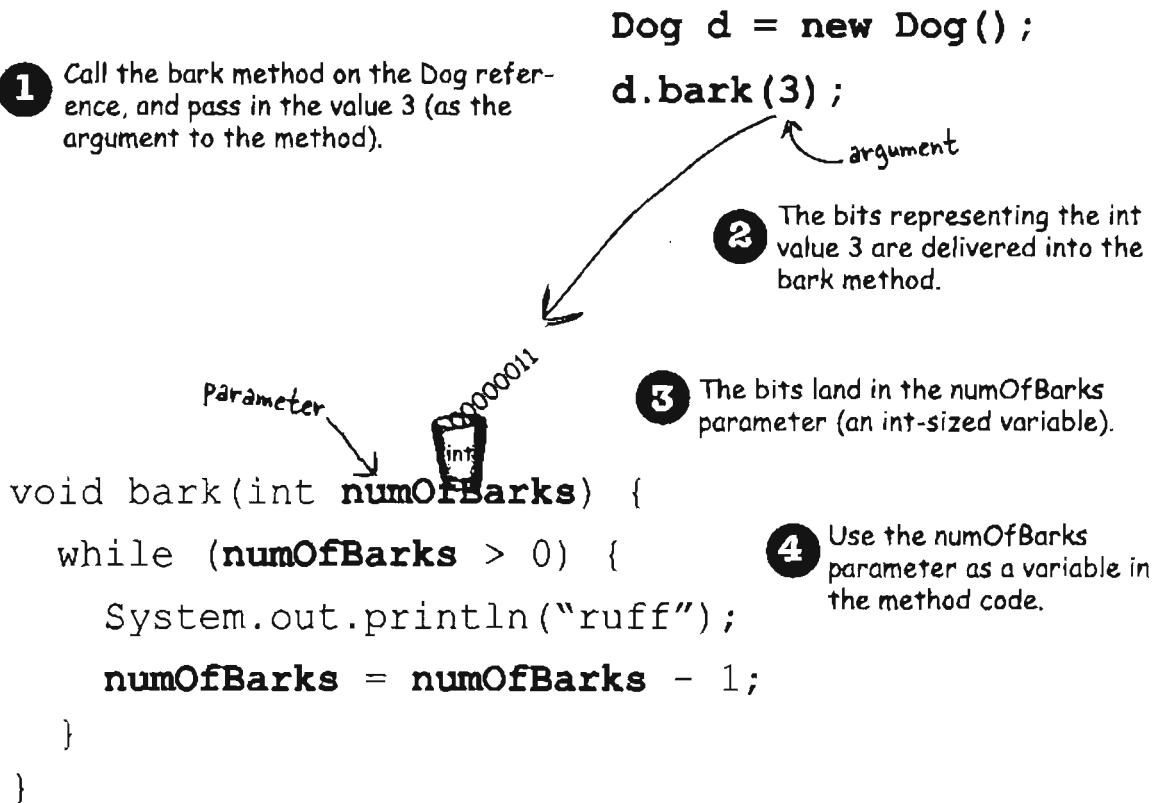
```
d.bark(3);
```

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:

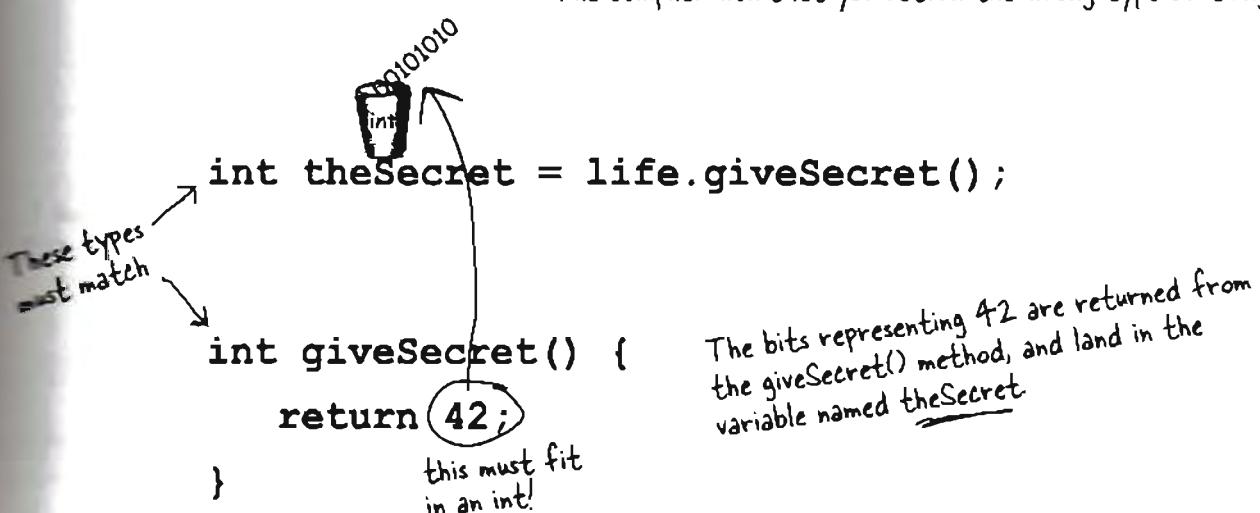
```
int giveSecret() {  
    return 42;  
}
```

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.



multiple arguments

You can send more than one thing to a method

Methods can have multiple parameters. Separate them with commas when you declare them, and separate the arguments with commas when you pass them. Most importantly, if a method has parameters, you *must* pass arguments of the right type and order.

Calling a two-parameter method, and sending it two arguments.

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

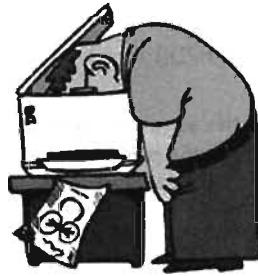
```
void go() {  
    int foo = 7;  
    int bar = 3;  
    t.takeTwo(foo, bar);  
}  
  
void takeTwo(int x, int y) {  
    int z = x + y;  
    System.out.println("Total is " + z);  
}
```

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;`



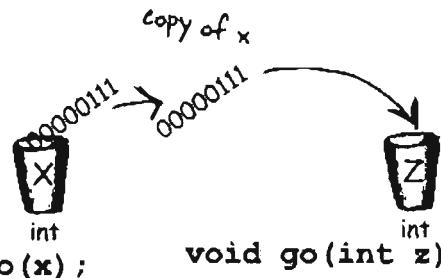
- 1 Declare an int variable and assign it the value '7'. The bit pattern for 7 goes into the variable named x.

`void go(int z) { }`



- 2 Declare a method with an int parameter named z.

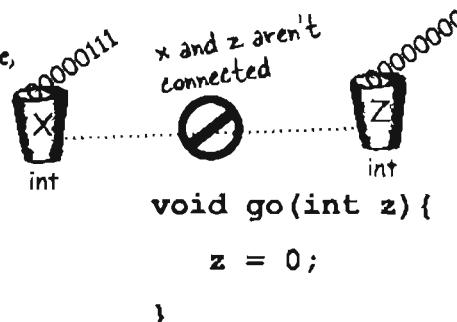
`foo.go(x);`



`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.

x doesn't change, even if z does.



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

x and z aren't connected

- 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.

arguments and return values

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?

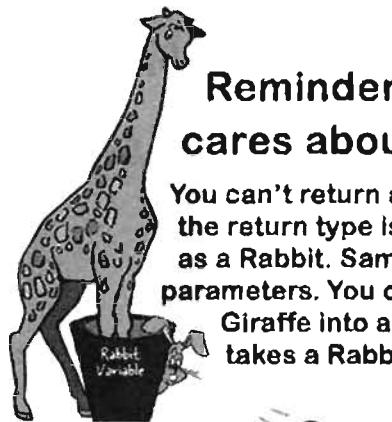
A: Sort of. A method can declare only one return value. BUT... If you want to return, say, three int values, then the declared return type can be an int array. Stuff those ints into the array, and pass it on back. It's a little more involved to return multiple values with different types; we'll be talking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

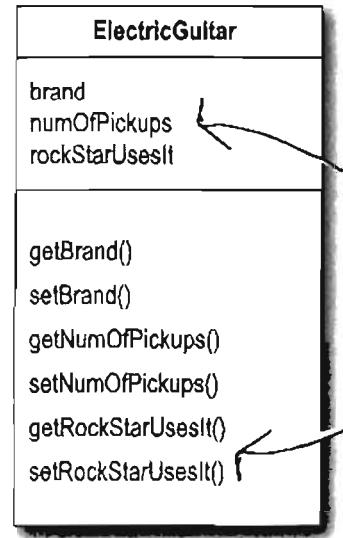
    void setBrand(String aBrand) {
        brand = aBrand;
    }

    int getNumOfPickups() {
        return numOfPickups;
    }

    void setNumOfPickups(int num) {
        numOfPickups = num;
    }

    boolean getRockStarUsesIt() {
        return rockStarUsesIt;
    }

    void setRockStarUsesIt(boolean yesOrNo) {
        rockStarUsesIt = yesOrNo;
    }
}
```



Note: Using these naming conventions means you'll be following an important Java standard!



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) { ↗ We put in checks  
        height = ht;  
    }  
}
```

Hide the data

Yes it is that simple to go from an implementation that's just begging for bad data to one that protects your data *and* protects your right to modify your implementation later.

OK, so how exactly do you *hide the data*? With the **public** and **private** access modifiers. You're familiar with **public**—we use it with every main method.

Here's an encapsulation *starter rule of thumb* (all standard disclaimers about rules of thumb are in effect): mark your instance variables **private** and provide **public** getters and setters for access control. When you have more design and coding savvy in Java, you will probably do things a little differently, but for now, this approach will keep you safe.

Mark instance variables **private.**

Mark getters and setters **public.**

"Sadly, Bill forgot to encapsulate his Cat class and ended up with a flat cat."

(overheard at the water cooler).



This week's interview: An Object gets candid about encapsulation.

HeadFirst: What's the big deal about encapsulation?

Object: OK, you know that dream where you're giving a talk to 500 people when you suddenly realize—you're *naked*!

HeadFirst: Yeah, we've had that one. It's right up there with the one about the Pilates machine and... no, we won't go there. OK, so you feel naked. But other than being a little exposed, is there any danger?

Object: Is there any danger? Is there any *danger*? [starts laughing] Hey, did all you other instances hear that, "*Is there any danger?*" he asks? [falls on the floor laughing]

HeadFirst: What's funny about that? Seems like a reasonable question.

Object: OK, I'll explain it. It's [bursts out laughing again, uncontrollably]

HeadFirst: Can I get you anything? Water?

Object: Whew! Oh boy. No I'm fine, really. I'll be serious. Deep breath. OK, go on.

HeadFirst: So what does encapsulation protect you from?

Object: Encapsulation puts a force-field around my instance variables, so nobody can set them to, let's say, something *inappropriate*.

HeadFirst: Can you give me an example?

Object: 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?

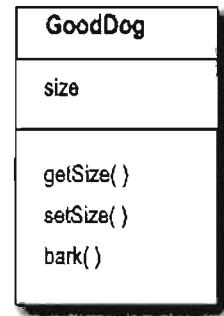
Object: The point to setters (and getters, too) is that *you can change your mind later, without breaking anybody else's code!* Imagine if half the people in your company used your class with public instance variables, and one day you suddenly realized, "Oops—there's something I didn't plan for with 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 minuscule and would rarely—if ever—be worth it.

how objects behave

Encapsulating the GoodDog class

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.

```
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();  
    }  
}
```



instead of:

int x = 3 + 24;

you can say:

int x = 3 + one.getSize();

How do objects in an array behave?

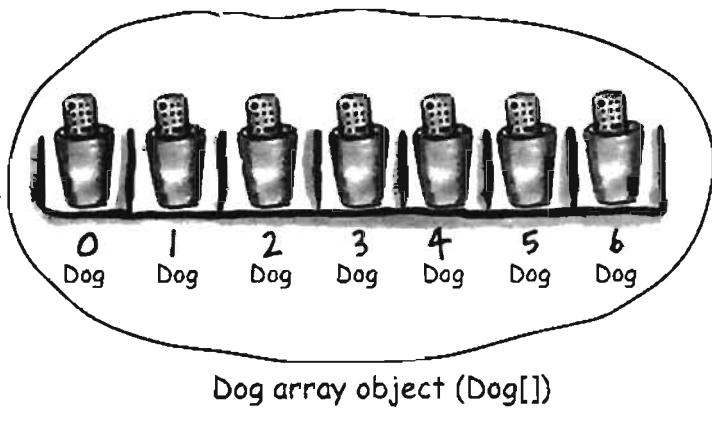
Just like any other object. The only difference is how you *get* to them. In other words, how you get the remote control. Let's try calling methods on Dog objects in an array.

- 1 Declare and create a Dog array, to hold 7 Dog references.

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

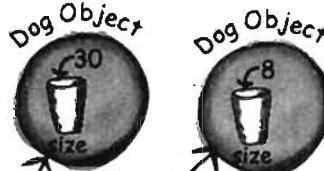


Dog[]



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

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

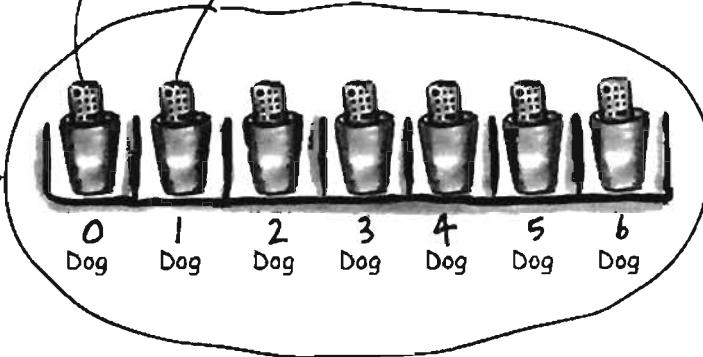


- 3 Call methods on the two Dog objects.

```
pets[0].setSize(30);
int x = pets[0].getSize();
pets[1].setSize(8);
```



Dog[]



Initializing instance variables

Declaring and initializing instance variables

You already know that a variable declaration needs at least a name and a type:

```
int size;  
String name;
```

And you know that you can initialize (assign a value) to the variable at the same time:

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

But when you don't initialize an instance variable, what happens when you call a getter method? In other words, what is the *value* of an instance variable *before* you initialize it?

```
class PoorDog {  
  
    private int size; ← declare two instance variables,  
    private String name; but don't assign a value  
  
    public int getSize() { ← What will these return??  
        return size;  
    }  
    public String getName() {  
        return name;  
    }  
  
}
```

```
public class PoorDogTestDrive {  
    public static void main (String[] args) {  
        PoorDog one = new PoorDog();  
        System.out.println("Dog size is " + one.getSize());  
        System.out.println("Dog name is " + one.getName());  
    }  
}
```

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

Instance variables always get a default value. If you don't explicitly assign a value to an instance variable, or you don't call a setter method, the instance variable still has a value!

integers	0
floating points	0.0
booleans	false
references	null

What do you think? Will this even compile?
You don't have to initialize instance variables, because they always have a default value. Number primitives (including char) get 0, booleans get false, and object reference variables get null.
(Remember, null just means a remote control that isn't controlling / programmed to anything. A reference, but no actual object)

The difference between instance and local variables

- 1** **Instance** variables are declared inside a class but not within a method.

```
class Horse {
    private double height = 15.2;
    private String breed;
    // more code...
}
```

- 2** Local variables are declared within a method.

```
class AddThing {
    int a;
    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.

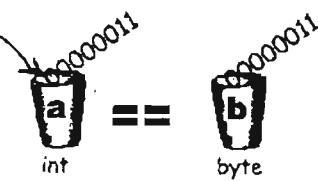
To compare two primitives, use the `==` operator

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)



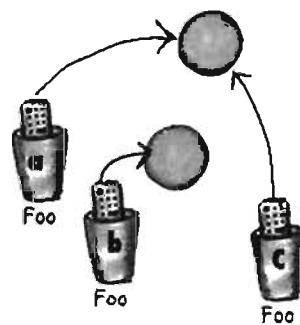
To see if two references are the same (which means they refer to the same object on the heap) use the `==` operator

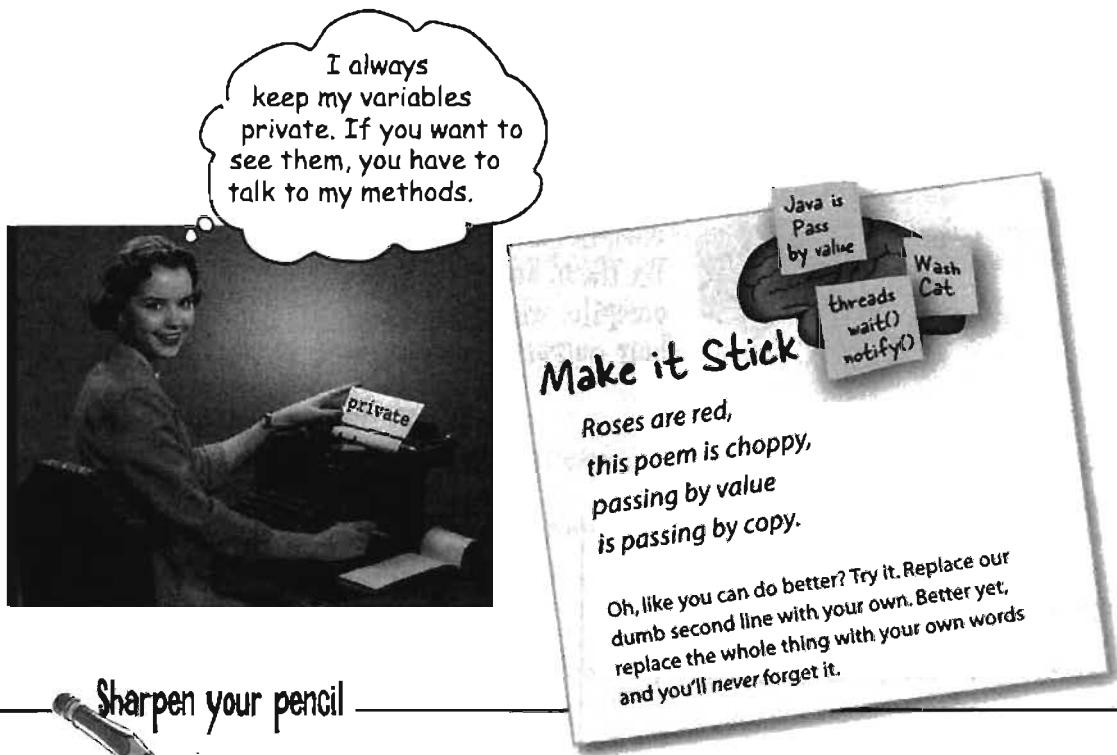
Remember, the `==` operator cares only about the pattern of bits in the variable. The rules are the same whether the variable is a reference or primitive. So the `==` operator returns true if two reference variables refer to the same object! In that case, we don't know what the bit pattern is (because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

the bit patterns are the
same for a and c, so they
are equal using `==`





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);
```

exercise: Be the Compiler



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

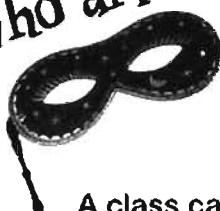
```
class XCopy {  
  
    public static void main(String [] args) {  
  
        int orig = 42;  
  
        XCopy x = new XCopy();  
  
        int y = x.go(orig);  
  
        System.out.println(orig + " " + y);  
    }  
  
    int go(int arg) {  
  
        arg = arg * 2;  
  
        return arg;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

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);  
    }  
}
```



Who am I?



A bunch of Java components, in full costume, are playing a party game, "Who am I?" They give you a clue, and you try to guess who they are, based on what they say. Assume they always tell the truth about themselves. If they happen to say something that could be true for more than one 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



Mixed Messages

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

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

Candidates:

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

```
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;  
}
```

methods use instance variables



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
```

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

ivar = x;	doStuff(x);	ivar + factor;	Puzzle4
obs.ivar = x;	obs.doStuff(x);	ivar * (2 + factor);	Puzzle4b int
obs[x].ivar = x;	obs[x].doStuff(factor);	ivar * (5 - factor);	Puzzle4b() short
obs[x].ivar = y;	obs[x].doStuff(x);	ivar * factor;	
Puzzle4 [] obs = new Puzzle4[6];		x = x + 1;	obs [x] = new Puzzle4b(x);
Puzzle4b [] obs = new Puzzle4b[6];		x = x - 1;	obs [] = new Puzzle4b();
Puzzle4b [] obs = new Puzzle4b[6];			obs [x] = new Puzzle4b();
			obs = new Puzzle4b();

```
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 _____
            }
        }
    }
}
```

puzzle: Five Minute Mystery



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 skuny 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

```

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.

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 method.

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 answers

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
```

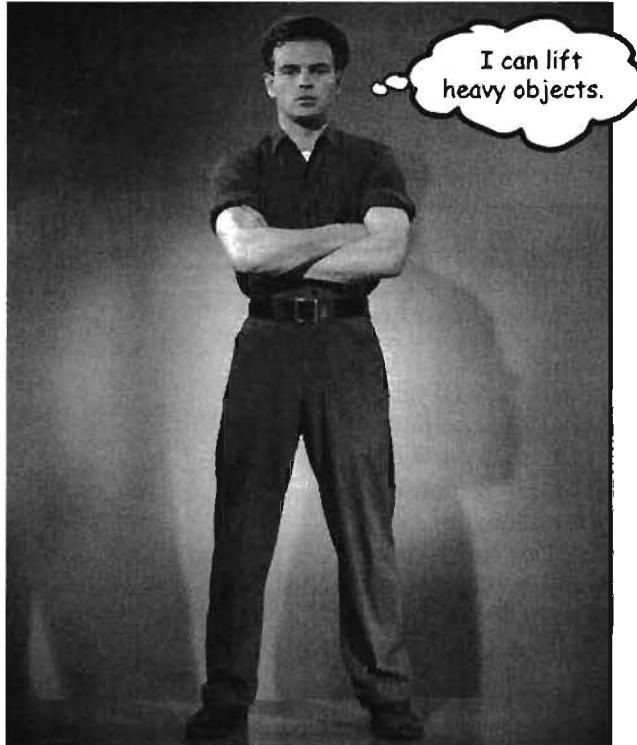
Answer to the 5-minute mystery...

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.

Candidates:	Possible output:
x < 9	14 7
index < 5	9 5
x < 20	19 1
index < 5	14 1
x < 7	25 1
index < 7	7 7
x < 19	20 1
index < 1	20 5

5 writing a program

Extra-Strength Methods



Let's put some muscle in our methods. We dabbled with variables, played with a few objects, and wrote a little code. But we were weak. We need more tools. Like **operators**. We need more operators so we can do something a little more interesting than, say, *bark*. And **loops**. We need loops, but what's with the wimpy *while* loops? We need **for** loops if we're really serious. Might be useful to **generate random numbers**. 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.

building a real game

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

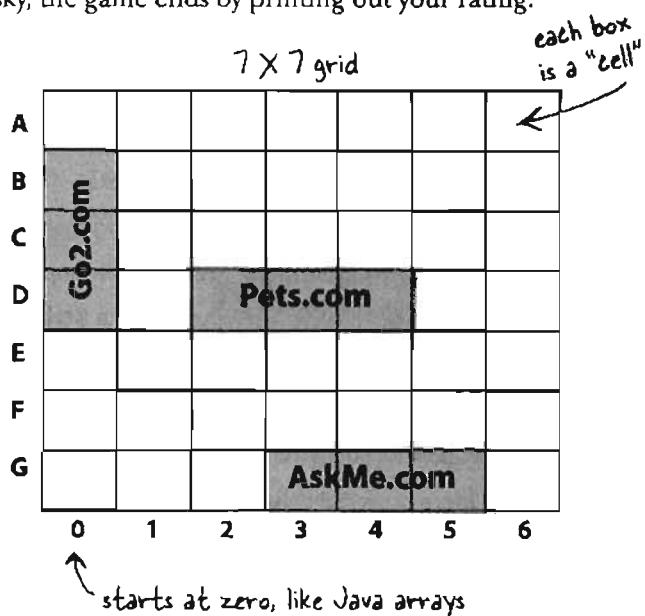
It's you against the computer, but unlike the real Battleship game, in this one you don't place any ships of your own. Instead, your job is to sink the computer's ships in the fewest number of guesses.

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

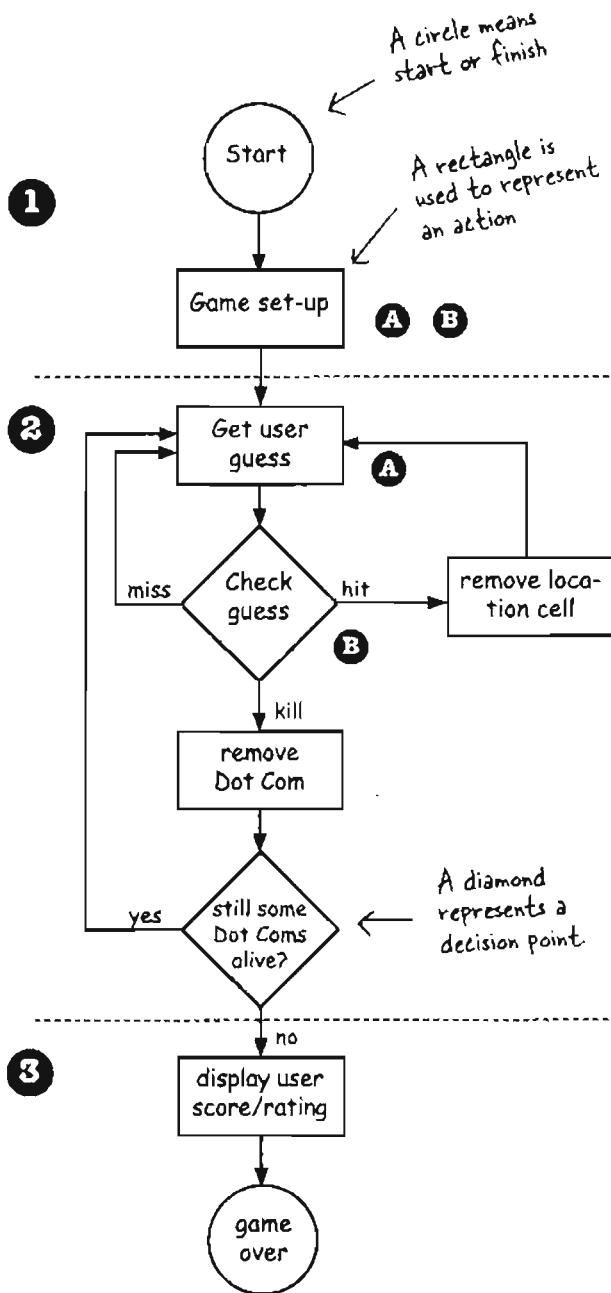
Repeat the following until there are no more Dot Coms:

- 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.

a simpler version of the game

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.

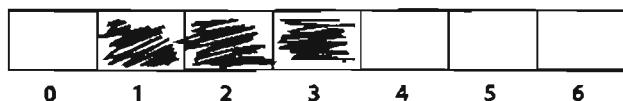
The goal is the same, though, so the game still needs to make a DotCom instance, assign it a location somewhere in the row, get user input, and when all of the DotCom's cells have been hit, the game is over. This simplified version of the game gives us a big head start on building the full game. If we 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.

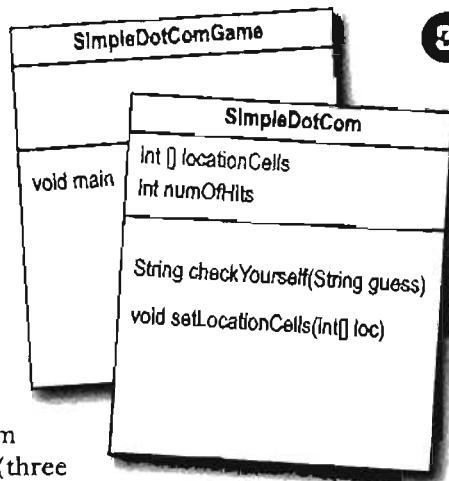
- 1 Game starts, and creates ONE DotCom and gives it a location on three cells in the single row of seven cells.

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 numHits variable.

- 3 Game finishes when all three cells have been hit (the numHits 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 **prep code** for the methods. (You'll see this in just a moment.)
- Write **test code** for the methods.
- Implement the class.
- Test the methods.
- Debug and reimplement as needed.
- Express gratitude that we don't have to test our so-called *learning experience* app on actual live users.



Flex those dendrites.

How would you decide which class or classes to build first, when you're writing a program? Assuming that all but the tiniest programs need more than one class (if you're following good OO principles and not having one class do many different jobs), where do you start?

The three things we'll write for each class:

prep code **test code** **real code**

This bar is displayed on the next set of pages to tell you which part you're working on. For example, if you see this picture at the top of a page, it means you're working on prepcode for the SimpleDotCom class.

SimpleDotCom class

prep code **test code** **real code**

prep code

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

test code

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

real code

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

To Do:

SimpleDotCom class

- write prep code
- write test code
- write final Java code

SimpleDotComGame class

- write prep code
- write test code [no]
- write final Java code

SimpleDotCom class

prep code test code real code

```
SimpleDotCom
int[] locationCells
int numHits

String checkYourself(String guess)
void setLocationCells(int[] loc)
```

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 *numHits* 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.



Oh my! For a minute there I thought you weren't gonna write your test code first. Who! Don't scare me like that.

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.

SimpleDotCom class

prep code test code real 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 precode 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 precode:

```
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);
    }
}
```

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 precode to javacode; you'll see a few adjustments. The precode 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  are for things (syntax and language features) you haven't seen yet. They're explained on the opposite page.

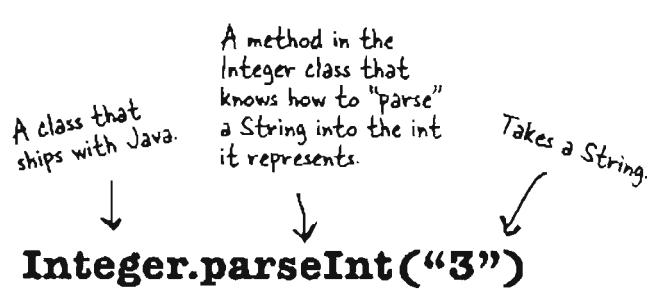
GET the user guess
CONVERT the user guess to an int
REPEAT with each cell in the int array
IF the user guess matches
INCREMENT the number of hits
// FIND OUT if it was the last cell
IF number of hits is 3,
RETURN "kill" as the result
ELSE it was not a kill, so
RETURN "hit"
ELSE
RETURN "miss"

```
public String checkYourself(String stringGuess) {
    int guess = Integer.parseInt(stringGuess);  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) {  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";
            numOfHits++;  we got a hit!
            break;  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

Read this for loop declaration as "repeat for each element in the 'locationCells' array: take the next element in the array and assign it to the int variable 'cell'."

② The for loop

for (int cell : locationCells) { }

Declare a variable that will hold one element from the array. Each time through the loop, this variable (in this case an int variable named "cell"), will hold a different element from the array, until there are no more elements (or the code does a "break"... see #4 below).

The colon (:) means "in", so the value IN locationCells...

The array to iterate over in the loop. Each time through the loop, the next element in the array will be assigned to the variable "cell". (More on this at the end of this chapter.)

③ The post-increment operator

numOfHits++

The ++ means add 1 to whatever's there (in other words, increment by 1).

numOfHits++ is the same (in this case) as saying numOfHits = numOfHits + 1, except slightly more efficient.

④ break statement

break;

Gets you out of a loop. Immediately. Right here. No iteration, no boolean test, just get out now!

SimpleDotCom class

prep code test code real code

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
```

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 `checkYourself()` method. If the code is working correctly, we should see the result print out:

```
java SimpleDotComTestDrive  
hit
```

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.

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

```

SimpleDotCom class

prep code test code real code

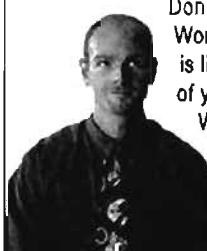
Precode for the SimpleDotComGame class

Everything happens in main()

There are some things you'll have to take on faith. For example, we have one line of precode 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 precode, you should assume that *somewhere* 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:
 - precode*
 - testcode*
 - real (Java) code*
- Precode should describe *what* to do, not *how* to do it. Implementation comes later.
- Use the precode 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 ● 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; ← make a variable to track how  
    ← many guesses the user makes  
DECLARE a variable to hold user  
guess count, set it to 0  
MAKE a SimpleDot-  
Com object  
COMPUTE a random number  
between 0 and 4  
MAKE an int array  
with the 3 cell locations, and  
INVOKE setLoca-  
tionCells on the dot  
com object  
DECLARE a boolean  
isAlive  
WHILE the dot  
com is still alive  
GET user input  
// CHECK it  
INVOKE checkYo-  
urself() on dot com  
INCREMENT  
numOfGuesses  
IF result is "kill"  
SET gameAlive to  
false  
PRINT the number  
of user guesses  
  
    GameHelper helper = new GameHelper(); ← this is a special class we wrote that has  
    ← the method for getting user input. For  
now, pretend it's part of Java  
  
    SimpleDotCom theDotCom = new SimpleDotCom(); ← make the dot com object  
  
    int randomNum = (int) (Math.random() * 5); ← make a random number for the first  
    ← cell, and use it to make the cell  
    ← locations array  
  
    int[] locations = {randomNum, randomNum+1, randomNum+2};  
  
    theDotCom.setLocationCells(locations); ← give the dot com its locations (the array)  
  
    boolean isAlive = true; ← make a boolean variable to track whether the game  
    ← is still alive, to use in the while loop test. repeat  
    ← while game is still alive.  
  
    while(isAlive == true) {  
  
        String guess = helper.getUserInput("enter a number"); ← get user input String  
  
        String result = theDotCom.checkYourself(guess); ← ask the dot com to check  
        ← the guess; save the returned  
        ← result in a String  
  
        numOfGuesses++; ← increment guess count  
  
        if (result.equals("kill")) ← was it a "kill"? if so, set isAlive to false (so we won't  
            isAlive = false; ← re-enter the loop) and print user guess count  
  
        System.out.println("You took " + numOfGuesses + " guesses"); ←  
  
    } // close if  
  
} // close while  
  
} // close main
```

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

```
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.

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)

- Getting user input using the GameHelper class

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

We declare a String variable to hold the user input String we get back ("3", "5", etc.).

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

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.

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  logo, you're seeing code that you have to type as-is and take on faith. Trust it. You'll learn how that code works *later*.



```
import java.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
```

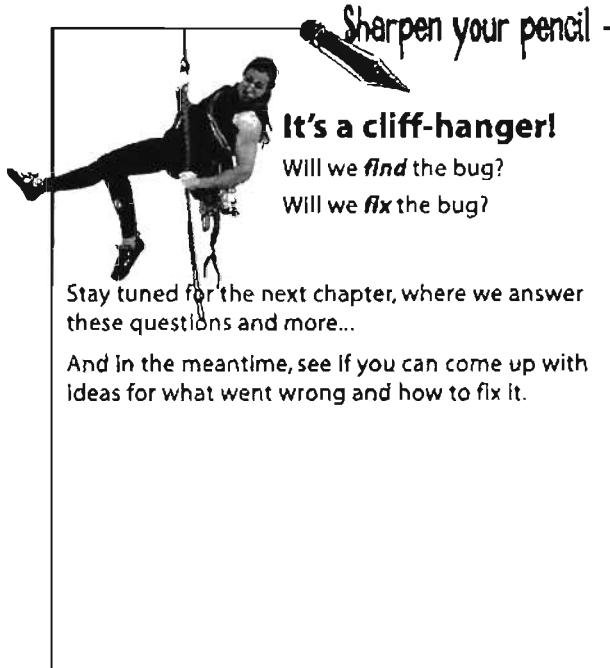
What's this? A bug?

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
```

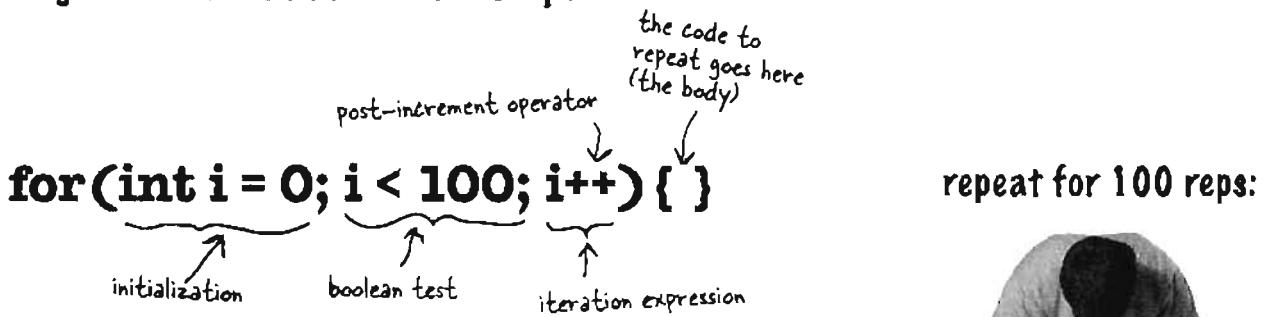


for loops

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.

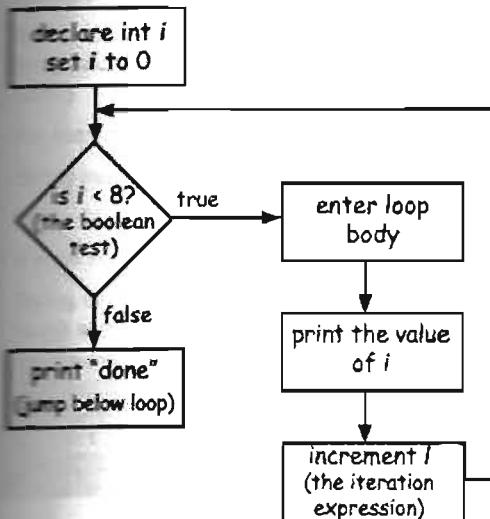
Part Three: Iteration expression

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.

Trips through a loop

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

System.out.println("done");
```



Difference between for and while

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

```
int i = 0; ← we have to declare and
              initialize the counter
while (i < 8) {
    System.out.println(i);
    i++; ← we have to increment
          the counter
}
System.out.println("done");
```

output:

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

++ **--**

Pre and Post Increment/Decrement Operator

The shortcut for adding or subtracting 1 from a variable.

x++;

Is the same as:

x = x + 1;

They both mean the same thing in this context:

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

And:

x--;

Is the same as:

x = x - 1;

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

enhanced for

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.

for (String name : nameArray) { }

Declare an iteration variable
that will hold a single element
in the array.

The colon (:) means "IN".

The code to repeat goes here
(the body).

The elements in the array **MUST** be compatible with the declared variable type.

With each iteration, a different element in the array will be assigned to the variable "name".

The collection of elements that you want to iterate over. Imagine that somewhere earlier, the code said:
String[] nameArray = {"Fred", "Mary", "Bob"};
With the first iteration, the name variable has the value of "Fred", and with the second iteration, a value of "Mary", etc.

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.

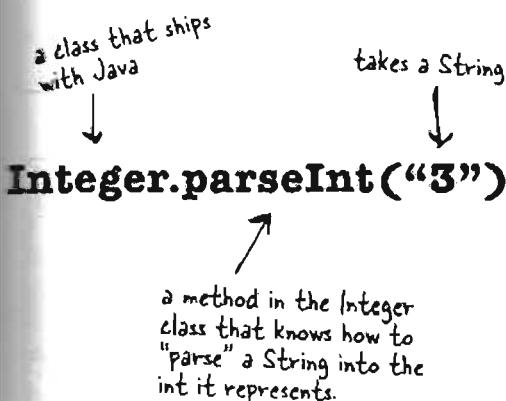
For example, *this won't work:*

```
String num = "2";
int x = 2;
if (x == num) // horrible explosion!
```

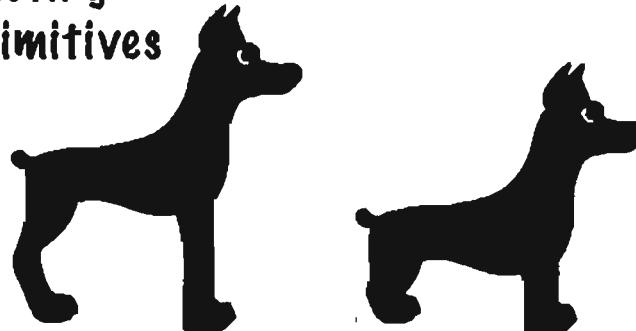
Trying to compile that makes the compiler laugh and mock you:

```
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



`long` → `short`
can be cast to

`01011101` but you might lose something
but you might lose something
`1101` bits on the left side were cut off

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.

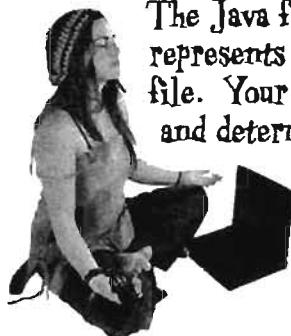
*It involves sign bits, binary, 'two's complement' and other geekery, all of which are discussed at the beginning of appendix B.

exercise: Be the JVM



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 OK  
% 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++) {`

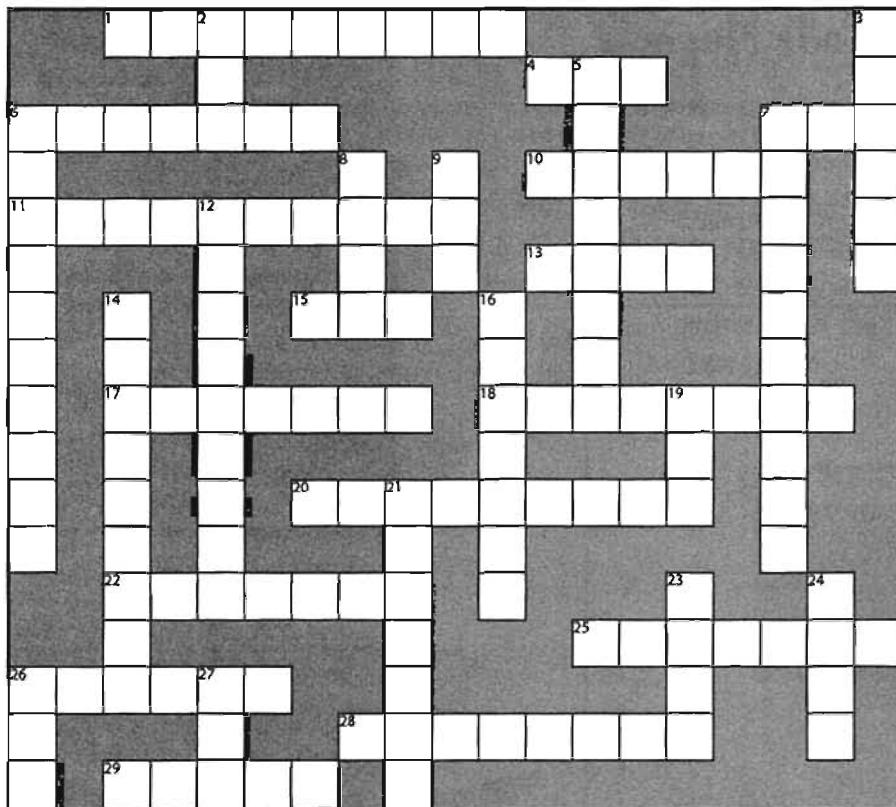
`public static void main(String [] args) {`

```

File Edit Window Help Run
1 java MultiFor
2 4
3 3
4 4
5 3
6 4
7 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. Precode-esque
- 13. Change
- 15. The big toolkit
- 17. An array unit
- 18. Instance or local

Down

- 20. Automatic toolkit
- 22. Looks like a primitive, but...
- 25. Un-castable
- 26. Math method
- 28. Converter method
- 29. Leave early
- 2. Increment type
- 3. Class's workhorse
- 5. Pre is a type of _____
- 6. For's iteration _____
- 7. Establish first value
- 8. While or For
- 9. Update an instance variable
- 12. 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**

match each candidate with one of the possible outputs

exercise solutions



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
```

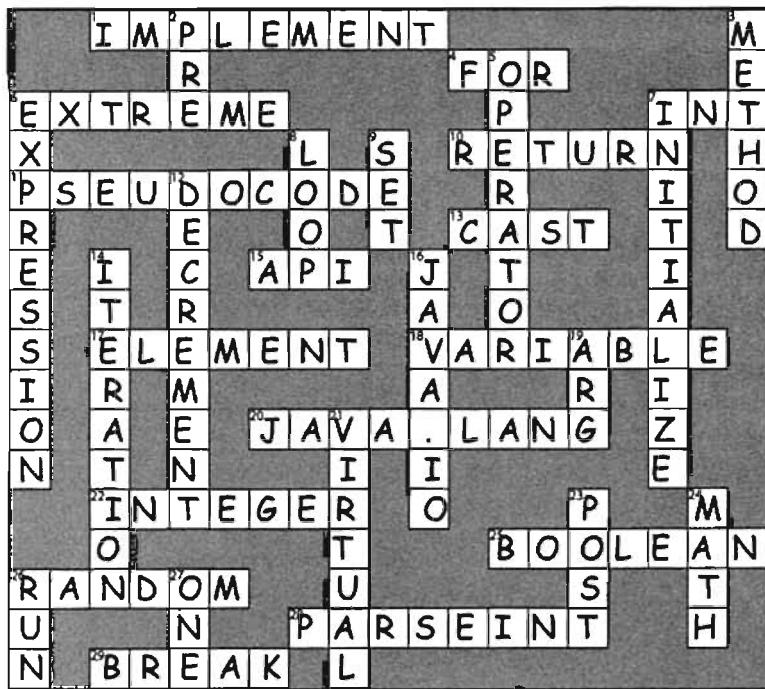
Code Magnets:

```
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 Windows  
% 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

6 get to know the Java API

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.

we still have a bug

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

How it's supposed to look

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

A complete game interaction
(your mileage may vary)

```
File Edit Window Help Smile
%java 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); ← 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) { ← Repeat with each
                                        thing in the array.
        [REDACTED] ← Compare the user
                      guess to this element
                      (cell), in the array.

        break; ← Get out of the loop, no need
                  to test the other cells.
    } // end if
} // end for

if (numOfHits == locationCells.length) { ← We're out of the loop, but
    result = "kill"; let's see if we're now 'dead'
} // end if                                         (hit 3 times) and change the
                                                result String to "kill".

System.out.println(result); ← Display the result for the user
                            ("miss", unless it was changed to "hit" or "kill").

return result; ← Return the result back to
} // end method                                     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.

fixing the bug

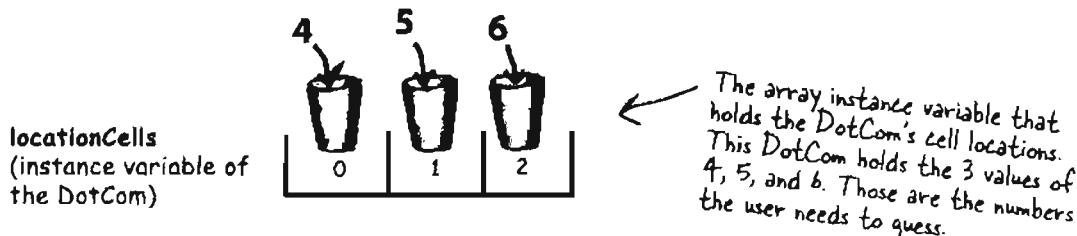
How do we fix it?

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.

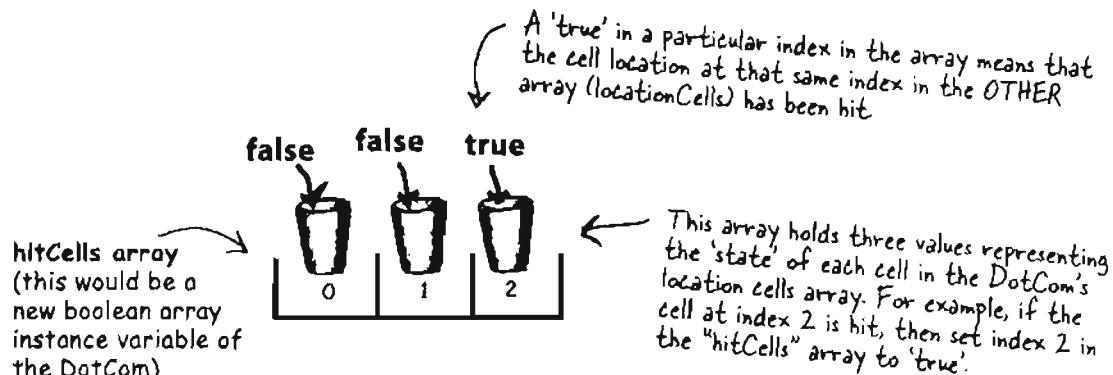


The DotCom has an instance variable—an int array—that holds that DotCom object's cell locations.



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.



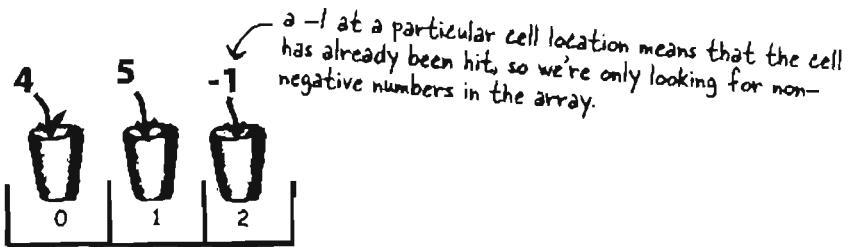
Option one is too clunky

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...

Option two

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

`locationCells`
(instance variable of
the DotCom)



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...

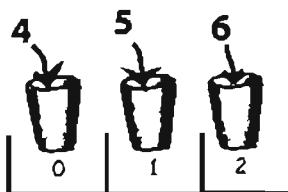
prep code

prep code test code real code

③ Option three

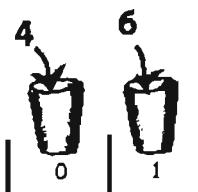
We delete each cell location as it gets hit, and then modify the array to be smaller. Except arrays can't change their size, so we have to make a new array and copy the remaining cells from the old array into the new smaller array.

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 precode for part of the checkYourself() method:

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

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

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



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

when arrays aren't enough

Wake up and smell the library

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

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

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

The Java 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.

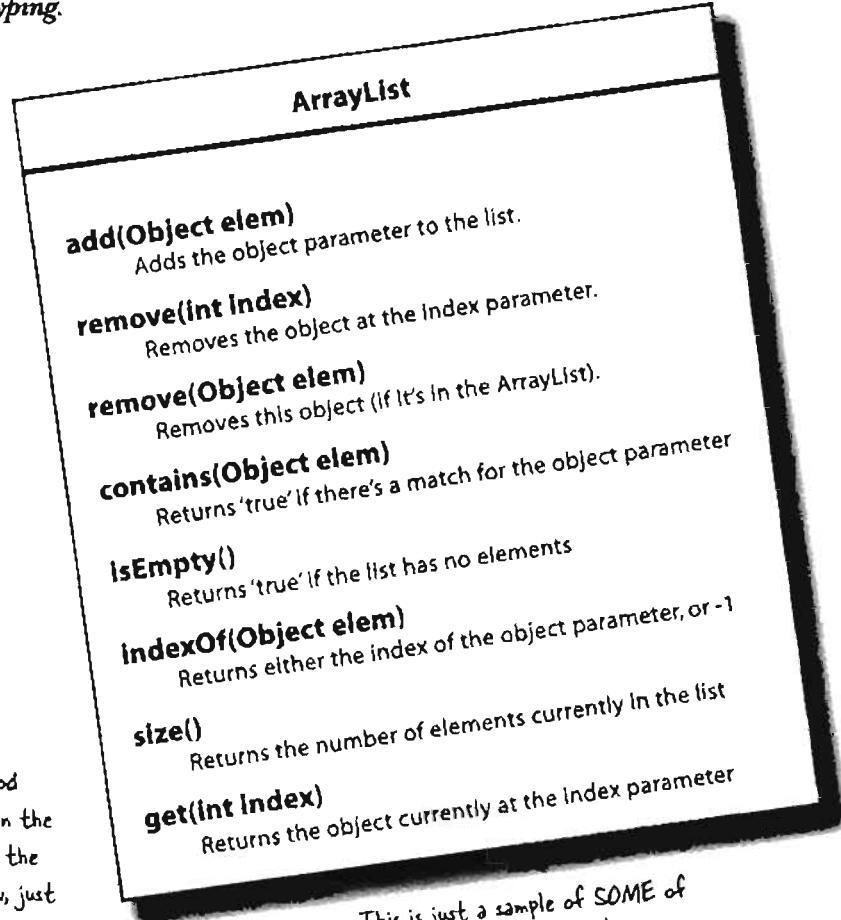
That means no typing.

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.

(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

- ➊ Make one

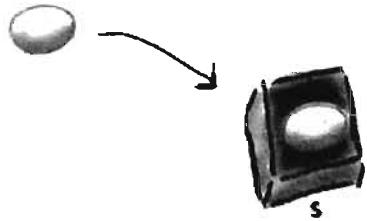
`ArrayList<Egg> myList = new ArrayList<Egg>();`

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

A new ArrayList object is created on the heap. It's little because it's empty.
- ➋ Put something in it

`Egg s = new Egg();`

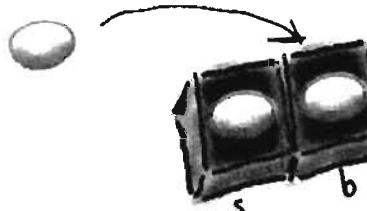
`myList.add(s);`



Now the ArrayList grows a "box" to hold the Egg object
- ➌ Put another thing in it

`Egg b = new Egg();`

`myList.add(b);`



The ArrayList grows again to hold the second Egg object.
- ➍ Find out how many things are in it

`int theSize = myList.size();`

The ArrayList is holding 2 objects so the size() method returns 2
- ➎ 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
- ➏ 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
- ➐ Find out if it's empty

`boolean empty = myList.isEmpty();`

it's definitely NOT empty, so isEmpty() returns false
- ➑ Remove something from it

`myList.remove(s);`



Hey look - it shrank!

when arrays aren't enough



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

ArrayList

regular array

ArrayList<String> myList = new ArrayList<String>();	String [] myList = new String[2];
String a = new String("whooahoo");	String a = new String("whooahoo");
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.



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

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

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

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

ArrayList: I can't *believe* you buy into that urban legend. No, I am *not* just a less-efficient array. I will admit that there are a few *extremely* rare situations where an array might be just a tad, I repeat, *tad* bit faster for certain things. But is it worth the *minuscule* performance gain to give up all this *power*. Still, look at all this *flexibility*. And as for the primitives, of course you can put a primitive in an `ArrayList`, as long as it's wrapped in a primitive wrapper class (you'll see a lot more on that in chapter 10). And 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.

difference between ArrayList and array

Comparing ArrayList to a regular array

ArrayList	regular array
<pre>ArrayList<String> myList = new ArrayList<String>();</pre>	<pre>String [] myList = new String[2];</pre>
<pre>String a = new String("whooohoo"); myList.add(a);</pre>	<pre>String a = new String("whooohoo"); myList[0] = a;</pre>
<pre>String b = new String("Frog"); myList.add(b);</pre>	<pre>String b = new String("Frog"); myList[1] = b;</pre>
<pre>int theSize = myList.size();</pre>	<pre>int theSize = myList.length;</pre>
<pre>Object o = myList.get(1);</pre>	<pre>String o = myList[1];</pre>
<pre>myList.remove(1);</pre>	<pre>myList[1] = null;</pre>
<pre>boolean isIn = myList.contains(b);</pre>	<pre>boolean isIn = false; for (String item : myList) { if (b.equals(item)) { isIn = true; break; } }</pre>

Here's where it
starts to look
really different...

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

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

Comparing ArrayList to a regular array

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).

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.

ArrayLists 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.

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.

the buggy DotCom code

prep code test code real code

Let's fix the DotCom code.

Remember, this is how the buggy version looks:

```
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
```

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.

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;           ← Ignore this line for
public class DotCom {               now; we talk about
                                    it at the end of the
                                    chapter.

    private ArrayList<String> locationCells;
    // private int numHits;
    // don't need that now           ← Change the String array to an ArrayList that holds Strings.

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

    public String checkYourself(String userInput) {
        String result = "miss";
        int index = locationCells.indexOf(userInput);

        if (index >= 0) {             ← If index is greater than or equal to
            locationCells.remove(index); ← zero, the user guess is definitely in the
                                         list, so remove it.

            if (locationCells.isEmpty()) { ← If the list is empty, this
                result = "kill";       was the killing blow!
            } else {
                result = "hit";
            } // close if
        } // close outer if

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

```

Now with
ArrayList
power!

making the DotComBust

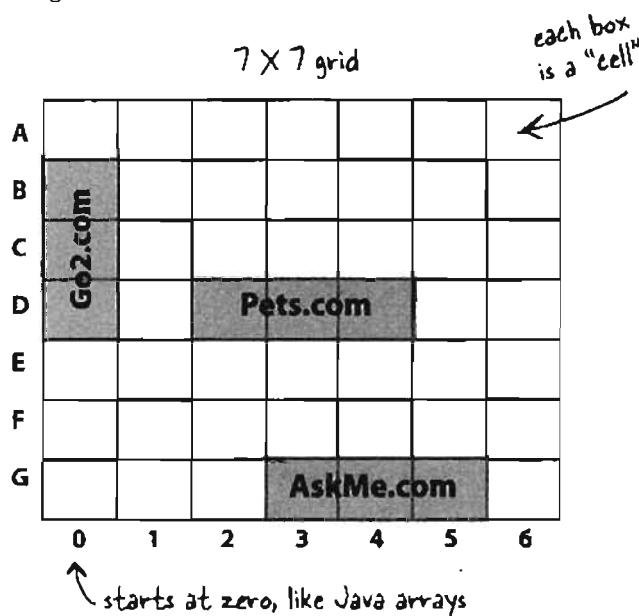
Let's build the REAL game: "Sink a Dot Com"

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 7×7 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 7×7 grid and three Dot Coms. Each Dot Com takes up three cells.

part of a game interaction

```
File Edit Window Help Sett
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).

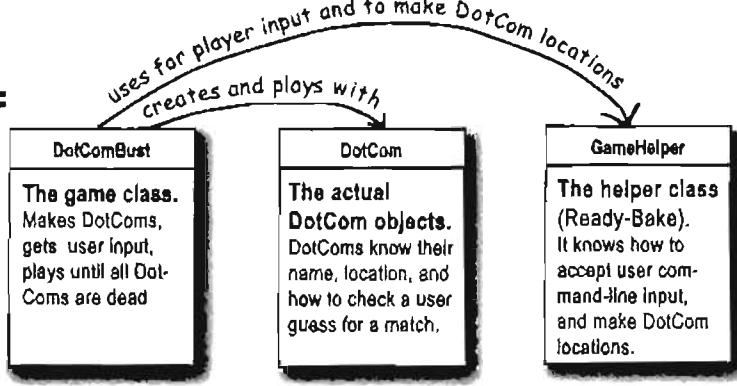
B DotComBust class (the game)

- ◎ 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.

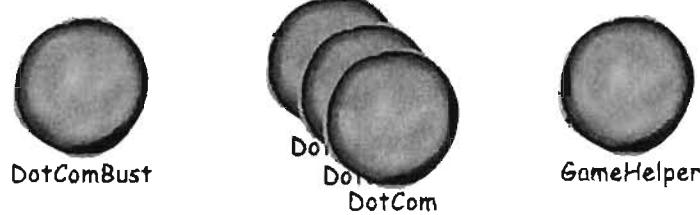
DotComBust class continued...

- ◎ 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.

3 Classes:



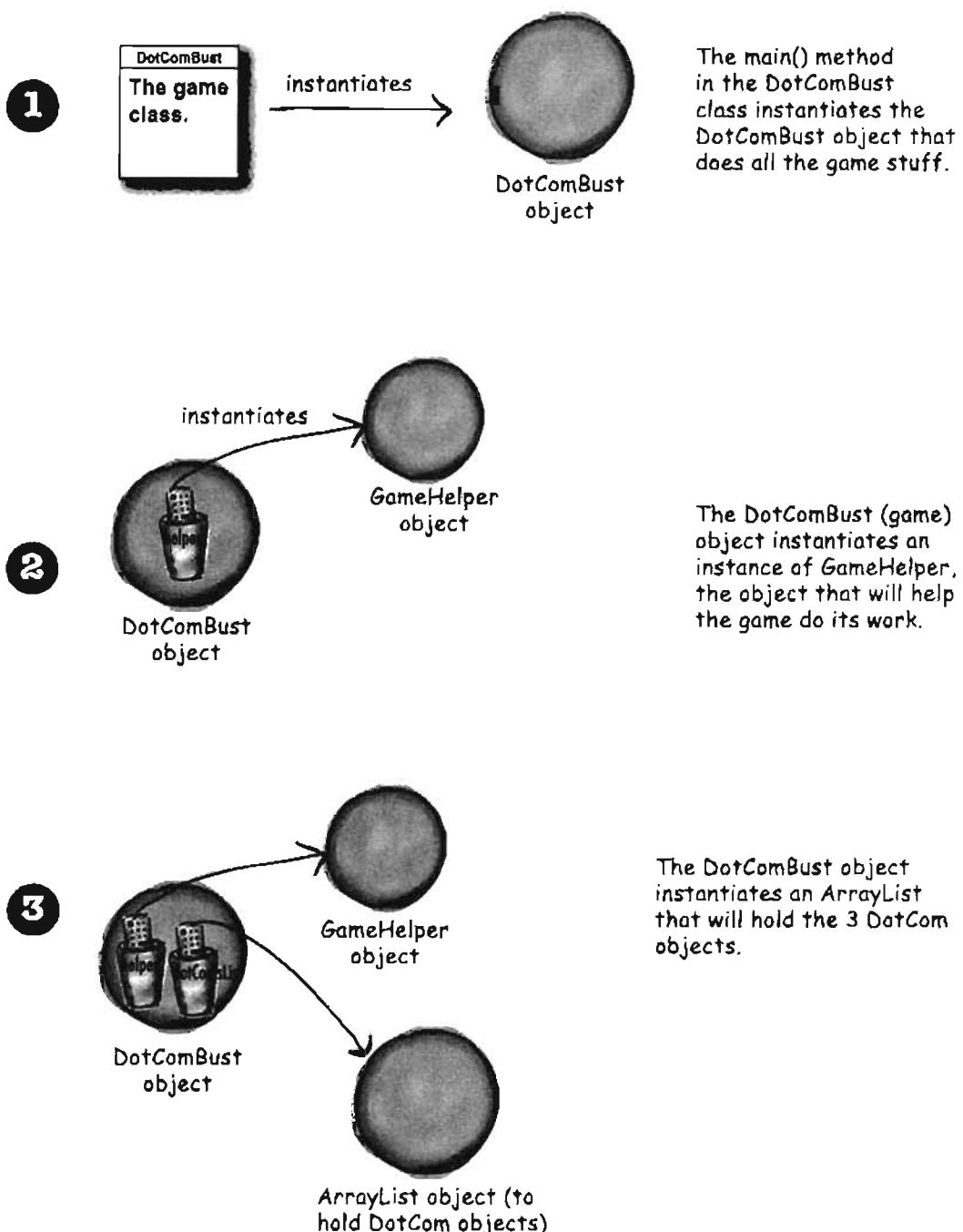
5 Objects:

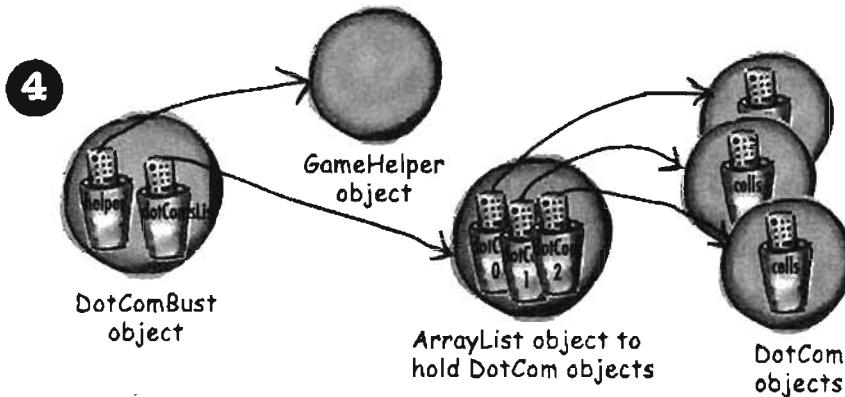


Plus 4
ArrayLists: 1 for the DotComBust and 1 for each of the 3 DotCom objects.

detailed structure of the game

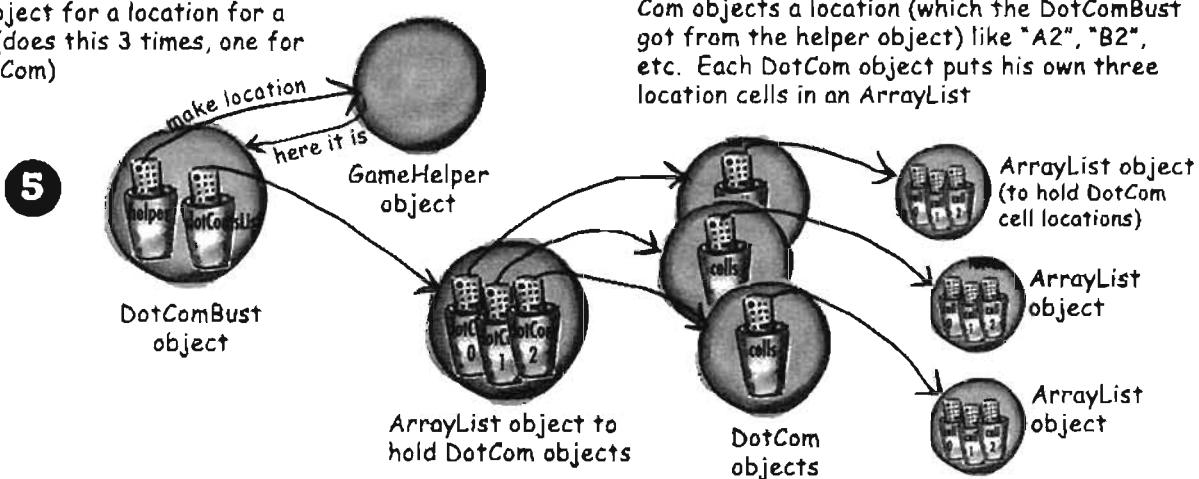
Who does what in the DotComBust game (and when)





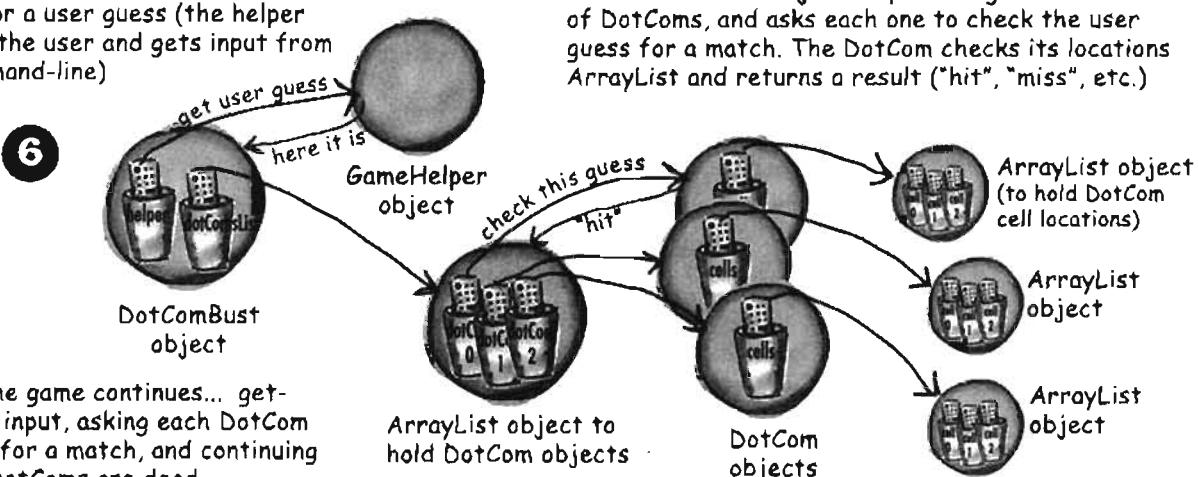
The `DotComBust` object creates three `DotCom` objects (and puts them in the `ArrayList`)

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)



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

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.)

the DotComBust class (the game)

prep code test code real code

DotComBust
GameHelper helper ArrayList dotComsList int numOfGuesses
setUpGame() startPlaying() checkUserGuess() finishGame()

Variable Declarations

Method Declarations

Method Implementations

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.

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.

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: 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

142

here

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.

ask the helper for a DotCom location
repeat with each DotCom in the list
call our own checkUserGuess method
as long as the DotCom list is NOT empty

declare and initialize the variables we'll need
get user input
print brief instructions for user
call our own finishGame method
make three DotCom objects, give 'em names, and stick 'em in the ArrayList
call the setter method on this DotCom to give it the location you just got from the helper

prep code link code real code

```

private void checkUserGuess(String userGuess) {
    numOfGuesses++;
    String result = "miss";
    for (DotCom dotComToTest : dotComsList) {
        result = dotComToTest.checkYourself(userGuess);
        if (result.equals("hit")) {
            break;
        }
        if (result.equals("kill")) {
            dotComsList.remove(dotComToTest);
            break;
        }
    } // close for
    System.out.println(result);
} // 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.");
        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();
    game.setUpGame();
    game.startPlaying();
} // close method
}

```

**Whatever you do,
DON'T turn the
page!**

**Not until you've
finished this
exercise.**

**Our version is on
the next page.**



- 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
- increment the number of guesses the user has made
- get out of the loop early, no point in testing the others
- print a message telling the user how he did in the game
- assume it's a 'miss', unless told otherwise
- tell the game object to start the main game play loop (keeps asking for user input and checking the guess)
- ask the DotCom to check the user guess, looking for a hit (or kill)
- create the game object

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) { ← Repeat with each DotCom in the list.
            ArrayList<String> newLocation = helper.placeDotCom(3); ← Ask the helper for a
            dotComToSet.setLocationCells(newLocation); ← Call the setter method on this
        } // close for loop
    } // close setUpGame method

    private void startPlaying() {
        while(!dotComsList.isEmpty()) { ← As long as the DotCom list is NOT empty (the ! means NOT, it's
            String userGuess = helper.getUserInput("Enter a guess"); ← Get user input
            checkUserGuess(userGuess); ← Call our own checkUserGuess method.
        } // close while
        finishGame(); ← Call our own finishGame method.
    } // close startPlaying method
}
```

Declare and initialize the variables we'll need.

Make an ArrayList of DotCom objects (in other words, a list that will hold ONLY DotCom objects, just as DotCom[] would mean an array of DotCom objects).

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

Print brief instructions for user.

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

As long as the DotCom list is NOT empty (the ! means NOT, it's the same as (dotComsList.isEmpty()) == false).

Get user input

Call our own checkUserGuess method.

Call our own finishGame method.

prep code

real code

```

private void checkUserGuess(String userGuess) {
    numOfGuesses++;
    ← increment the number of guesses the user has made

    String result = "miss";
    ← assume it's a 'miss', unless told otherwise

    for (DotCom dotComToTest : dotComsList) {
        ← repeat with all DotComs in the list

        result = dotComToTest.checkYourself(userGuess);
        ← ask the DotCom to check the user
        guess, looking for a hit (or kill)

        if (result.equals("hit")) {
            break;
            ← get out of the loop early, no point
            in testing the others
        }
        if (result.equals("kill")) {
            dotComsList.remove(dotComToTest);
            ← this guy's dead, so take him out of the
            DotComs list then get out of the loop
            break;
        }
    } // close for

    System.out.println(result);
    ← print the result for the user
} // close method
                                print a message telling the
                                user how he did in the game
                                }

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.");
        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();
    ← create the game object
    game.setUpGame();
    ← tell the game object to set up the game
    game.startPlaying();
    ← tell the game object to start the main
    game play loop (keeps asking for user
    input and checking the guess)
} // close method

```

the DotCom code

prep code

real code

The final version of the DotCom class

```
import java.util.*;
```

```
public class DotCom {  
    private ArrayList<String> locationCells;  
    private String name;
```

DotCom's instance variables:

- an ArrayList of cell locations
- the DotCom's name

```
    public void setLocationCells(ArrayList<String> loc) { ← A setter method that updates  
        locationCells = loc;  
    } ← the DotCom's location.  
        (Random location provided by  
        the GameHelper placeDotCom()  
        method.)
```

```
    public void setName(String n) ← Your basic setter method  
    name = n;  
}
```

```
    public String checkYourself(String userInput) { ← The ArrayList indexOf() method in  
        String result = "miss"; ← action! If the user guess is one of the  
        int index = locationCells.indexOf(userInput); ← entries in the ArrayList, indexOf()  
        if (index >= 0) { ← will return its ArrayList location. If  
            locationCells.remove(index); ← not, indexOf() will return -1.  
                ← Using ArrayList's remove() method to delete an entry.
```

```
                if (locationCells.isEmpty()) { ← Using the isEmpty() method to see if all  
                    result = "kill"; ← of the locations have been guessed  
                    System.out.println("Ouch! You sunk " + name + " : ( ");  
                } else { ← Tell the user when a DotCom has been sunk.  
                    result = "hit";  
                } // close if  
            } // close if  
            return result; ← Return: 'miss' or 'hit' or 'kill'.  
        } // close method  
    } // close class
```

Super Powerful Boolean Expressions

So far, when we've used boolean expressions for our loops or `if` tests, they've been pretty simple. We will be using more powerful boolean expressions in some of the Ready-Bake code you're about to see, and even though we know you wouldn't peek, we thought this would be a good time to discuss how to energize your expressions.

'And' and 'Or' Operators (`&&`, `||`)

Let's say you're writing a `chooseCamera()` method, with lots of rules about which camera to select. Maybe you can choose cameras ranging from \$50 to \$1000, but in some cases you want to limit the price range more precisely. You want to say something like:

'If the price *range* is between \$300 *and* \$400 then choose X.'

```
if (price >= 300 && price < 400) {
    camera = "X";
}
```

Let's say that of the ten camera brands available, you have some logic that applies to only a few of the list:

```
if (brand.equals("A") || brand.equals("B")) {
    // do stuff for only brand A or brand B
}
```

Boolean expressions can get really big and complicated:

```
if ((zoomType.equals("optical") &&
    (zoomDegree >= 3 && zoomDegree <= 8)) ||
    (zoomType.equals("digital") &&
    (zoomDegree >= 5 && zoomDegree <= 12))) {
    // do appropriate zoom stuff
}
```

If you want to get *really* technical, you might wonder about the *precedence* of these operators. Instead of becoming an expert in the arcane world of precedence, we recommend that you *use parentheses* to make your code clear.

Not equals (`!=` and `!`)

Let's say that you have a logic like, "of the ten available camera models, a certain thing is *true for all but one*."

```
if (model != 2000) {
    // do non-model 2000 stuff
}
or for comparing objects like strings...
if (!brand.equals("X")) {
    // do non-brand X stuff
}
```

Short Circuit Operators (`&&` , `||`)

The operators we've looked at so far, `&&` and `||`, are known as *short circuit* operators. In the case of `&&`, the expression will be true only if *both* sides of the `&&` are true. So if the JVM sees that the left side of a `&&` expression is false, it stops right there! Doesn't even bother to look at the right side.

Similarly, with `||`, the expression will be true if *either* side is true, so if the JVM sees that the left side is true, it declares the entire statement to be true and doesn't bother to check the right side.

Why is this great? Let's say that you have a reference variable and you're not sure whether it's been assigned to an object. If you try to call a method using this null reference variable (i.e. no object has been assigned), you'll get a `NullPointerException`. So, try this:

```
if (refVar != null &&
    refVar.isValidType()) {
    // do 'got a valid type' stuff
}
```

Non Short Circuit Operators (`&` , `|`)

When used in boolean expressions, the `&` and `|` operators act like their `&&` and `||` counterparts, except that they force the JVM to *always* check *both* sides of the expression. Typically, `&` and `|` are used in another context, for manipulating bits.

Ready-bake: GameHelper



```
import java.io.*;
import java.util.*;

public class GameHelper {

    private static final String alphabet = "abcdefg";
    private int gridLength = 7;
    private int gridSize = 49;
    private int [] grid = new int[gridSize];
    private int comCount = 0;

    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.toLowerCase();
    }

    public ArrayList<String> placeDotCom(int comSize) {
        ArrayList<String> alphaCells = new ArrayList<String>();
        String [] alphacoords = new String [comSize];           // holds 'f6' type coords
        String temp = null;                                     // temporary String for concat
        int [] coords = new int[comSize];                      // current candidate coords
        int attempts = 0;                                       // current attempts counter
        boolean success = false;                                // flag = found a good location ?
        int location = 0;                                       // current starting location

        comCount++;
        int incr = 1;                                         // nth dot com to place
        if ((comCount % 2) == 1) {                            // set horizontal increment
            incr = gridLength;                               // if odd dot com (place vertically)
        }                                                       // set vertical increment

        while ( !success & attempts++ < 200 ) {             // main search loop (32)
            location = (int) (Math.random() * gridSize);      // get random starting point
            //System.out.print(" try " + location);
            int x = 0;                                         // nth position in dotcom to place
            success = true;                                    // assume success
            while (success && x < comSize) {                  // look for adjacent unused spots
                if (grid[location] == 0) {                      // if not already used

```

This is the helper class for the game. Besides the user input method (that prompts the user and reads input from the command-line), the helper's Big Service is to create the cell locations for the DotComs. If we were you, we'd just back away slowly from this code, except to type it in and compile it. We tried to keep it fairly small to you wouldn't have to type so much, but that means it isn't the most readable code. And remember, you won't be able to compile the DotComBust game class until you have this class.

Note: For extra credit, you might try 'un-commenting' the `System.out.println()`'s in the `placeDotCom()` method, just to watch it work! These print statements will let you "cheat" by giving you the location of the DotComs, but it will help you test it.



Ready-bake Code

GameHelper class code continued...

```

        coords[x++] = location;           // save location
        location += incr;
        if (location >= gridSize){       // try 'next' adjacent
            success = false;           // out of bounds - 'bottom'
        }
        if (x>0 && (location % gridLength == 0)) { // out of bounds - right edge
            success = false;           // failure
        }
    } else {
        // System.out.print(" used " + location);
        success = false;               // found already used location
    }
}
} // end while

int x = 0;                                // turn location into alpha coords
int row = 0;
int column = 0;
// System.out.println("\n");
while (x < comSize) {
    grid[coords[x]] = 1;                   // mark master grid pts. as 'used'
    row = (int) (coords[x] / gridLength);   // get row value
    column = coords[x] % gridLength;        // get numeric column value
    temp = String.valueOf(alphabet.charAt(column)); // convert to alpha

    alphaCells.add(temp.concat(Integer.toString(row)));
    x++;
    // System.out.print(" coord "+x+" = " + alphaCells.get(x-1));
}
// System.out.println("\n");

return alphaCells;
}
}

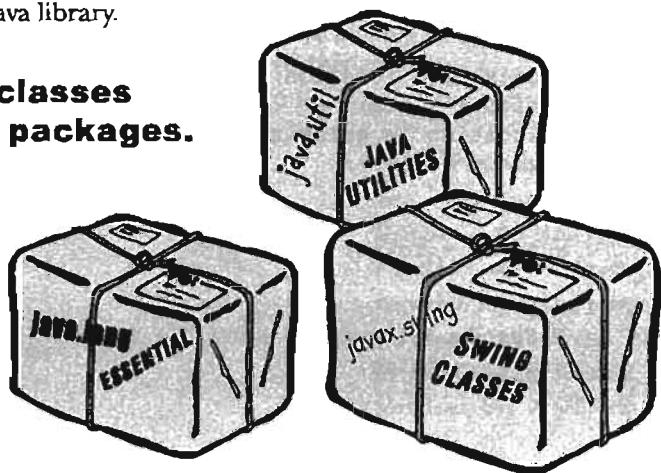
```

This is the statement that tells you exactly where the DotCom is located.

Using the Library (the Java API)

You made it all the way through the DotComBust game, thanks to the help of `ArrayList`. And now, as promised, it's time to learn how to fool around in the Java library.

In the Java API, classes are grouped into packages.



To use a class in the API, you have to know which package the class is in.

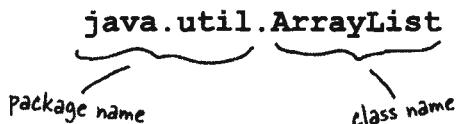
Every class in the Java library belongs to a package. The package has a name, like `javax.swing` (a package that holds some of the Swing GUI classes you'll learn about soon). `ArrayList` is in the package called `java.util`, which surprise surprise, holds a pile of *utility* classes. You'll learn a lot more about packages in chapter 16, including how to put your *own* classes into your *own* packages. For now though, we're just looking to *use* some of the classes that come with Java.

Using a class from the API, in your own code, is simple. You just treat the class as though you wrote it yourself... as though you compiled it, and there it sits, waiting for you to use it. With one big difference: somewhere in your code you have to indicate the *full* name of the library class you want to use, and that means package name + class name.

Even if you didn't know it, *you've already been using classes from a package*. `System` (`System.out.println`), `String`, and `Math` (`Math.random()`), all belong to the `java.lang` package.

You have to know the full name* of the class you want to use in your code.

ArrayList is not the *full* name of ArrayList, just as 'Kathy' isn't a full name (unless it's like Madonna or Cher, but we won't go there). The full name of ArrayList is actually:



You have to tell Java which ArrayList you want to use. You have two options:

A IMPORT

Put an import statement at the top of your source code file:

```
import java.util.ArrayList;
public class MyClass { ... }
```

OR

B TYPE

Type the full name everywhere in your code. Each time you use it. *Anywhere* you use it.

When you declare and/or instantiate it:

```
java.util.ArrayList<Dog> list = new java.util.ArrayList<Dog>();
```

When you use it as an argument type:

```
public void go(java.util.ArrayList<Dog> list) { }
```

When you use it as a return type:

```
public java.util.ArrayList<Dog> foo() { ... }
```

there are no
Dumb Questions

Q: Why does there have to be a full name? Is that the only purpose of a package?

A: Packages are important for three main reasons. First, they help the overall organization of a project or library. Rather than just having one horrendously large pile of classes, they're all grouped into packages for specific kinds of functionality (like GUI, or data structures, or database stuff, etc.)

Second, packages give you a name-scoping, to help prevent collisions if you and 12 other programmers in your company all decide to make a class with the same name. If you have a class named Set and someone else (including the Java API) has a class named Set, you need some way to tell the JVM which Set class you're trying to use.

Third, packages provide a level of security, because you can restrict the code you write so that only other classes in the same package can access it. You'll learn all about that in chapter 16.

Q: OK, back to the name collision thing. How does a full name really help? What's to prevent two people from giving a class the same package name?

A: Java has a naming convention that usually prevents this from happening, as long as developers adhere to it. We'll get into that in more detail in chapter 16.

*Unless the class is in the java.lang package.

Where'd that 'x' come from? (or, what does it mean when a package starts with **javax**?)

In the first and second versions of Java (1.02 and 1.1), all classes that shipped with Java (in other words, the standard library) were in packages that began with **java**. There was always **java.lang**, of course — the one you don't have to import. And there was **java.net**, **java.io**, **java.util** (although there was no such thing as **ArrayList** way back then), and a few others, including the **java.awt** package that held GUI-related classes.

Looming on the horizon, though, were other packages not included in the standard library. These classes were known as **extensions**, and came in two main flavors: **standard**, and not **standard**. Standard extensions were those that Sun considered official, as opposed to experimental, early access, or beta packages that might or might not ever see the light of day.

Standard extensions, by convention, all began with an 'x' appended to the regular **java** package starter. The mother of all standard extensions was the Swing library. It included several packages, all of which began with **javax.swing**.

But standard extensions can get promoted to first-class, ships-with-Java, standard-out-of-the-box library packages. And that's what happened to Swing, beginning with version 1.2 (which eventually became the first version dubbed 'Java 2').

"Cool," everyone thought (including us). "Now everyone who has Java will have the Swing classes, and we won't have to figure out how to get those classes installed with our end-users."

Trouble was lurking beneath the surface, however, because when packages get promoted, well of COURSE they have to start with **java**, not **javax**. Everyone KNOWS that packages in the standard library don't have that "x", and that only extensions have the "x". So, just (and we mean just) before version 1.2 went final, Sun changed the package names and deleted the "x" (among other changes). Books were printed and in stores featuring Swing code with the new names. Naming conventions were intact. All was right with the Java world.

Except the 20,000 or so screaming developers who realized that with that simple name change came disaster! All of their Swing-using code had to be changed! The horror! Think of all those import statements that started with **javax**...

And in the final hour, desperate, as their hopes grew thin, the developers convinced Sun to "screw the convention, save our code". The rest is history. So when you see a package in the library that begins with **javax**, you know it started life as an extension, and then got a promotion.



BULLET POINTS

- **ArrayList** is a class in the Java API.
- To put something into an **ArrayList**, use **add()**.
- To remove something from an **ArrayList** use **remove()**.
- To find out where something is (and if it is) in an **ArrayList**, use **indexOf()**.
- To find out if an **ArrayList** is empty, use **isEmpty()**.
- To get the size (number of elements) in an **ArrayList**, use the **size() method**.
- To get the length (number of elements) in a regular old array, remember, you use the **length variable**.
- An **ArrayList** **resizes dynamically** to whatever size is needed. It grows when objects are added, and it **shrinks** when objects are removed.
- You declare the type of the array using a **type parameter**, which is a type name in angle brackets. Example: **ArrayList<Button>** means the **ArrayList** will be able to hold only objects of type **Button** (or subclasses of **Button** as you'll learn in the next couple of chapters).
- Although an **ArrayList** holds objects and not primitives, the compiler will automatically "wrap" (and "unwrap" when you take it out) a primitive into an **Object**, and place that object in the **ArrayList** instead of the primitive. (More on this feature later in the book.)
- Classes are grouped into packages.
- A class has a full name, which is a combination of the package name and the class name. Class **ArrayList** is really **java.util.ArrayList**.
- To use a class in a package other than **java.lang**, you must tell Java the full name of the class.
- You use either an **import** statement at the top of your source code, or you can type the full name every place you use the class in your code.

there are no
Dumb Questions

Q: Does import make my class bigger? Does it actually compile the imported class or package into my code?

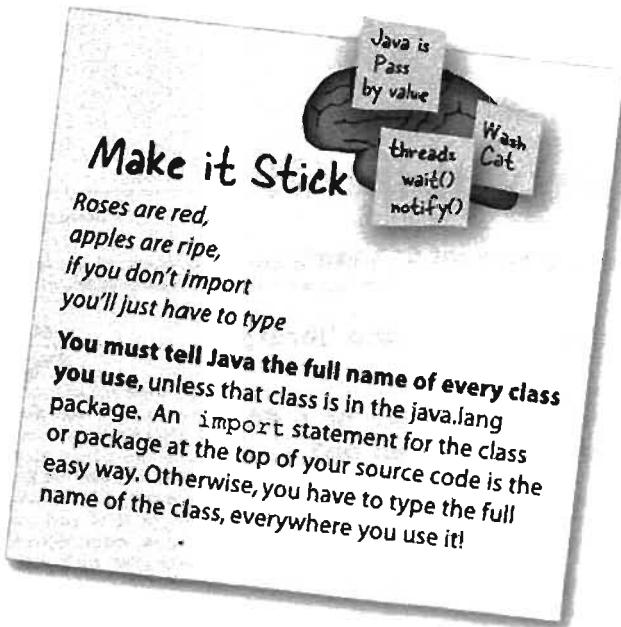
A: Perhaps you're a C programmer? An import is not the same as an include. So the answer is no and no. Repeat after me: "an import statement saves you from typing." That's really it. You don't have to worry about your code becoming bloated, or slower, from too many imports. An import is simply the way you give Java the *full name of a class*.

Q: OK, how come I never had to import the String class? Or System?

A: Remember, you get the java.lang package sort of "pre-imported" for free. Because the classes in java.lang are so fundamental, you don't have to use the full name. There is only one java.lang.String class, and one java.lang.System class, and Java darn well knows where to find them.

Q: Do I have to put my own classes into packages? How do I do that? Can I do that?

A: In the real world (which you should try to avoid), yes, you *will* want to put your classes into packages. We'll get into that in detail in chapter 16. For now, we won't put our code examples in a package.



One more time, in the unlikely event that you don't already have this down:



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

- Julia, 31, hand model

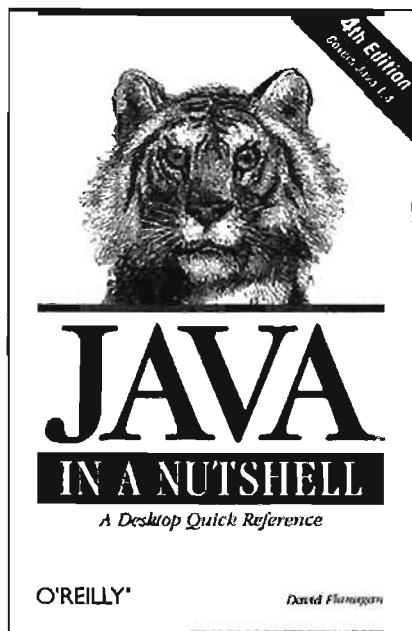
How to play with the API

Two things you want to know:

- 1 What classes are in the library?
- 2 Once you find a class, how do you know what it can do?



1 Browse a Book



2 Use the HTML API docs

Java 2 Platform Packages	
<code>java.applet</code>	Provides the classes necessary to create an applet and the classes an applet uses to communicate with its applet context.
<code>java.awt</code>	Contains all of the classes for creating user interfaces and for performing graphics and images.
<code>java.awt.color</code>	Provides classes for color spaces.
<code>java.awt.datatransfer</code>	Provides interfaces and classes for transforming data between one application and another.

1 Browse a Book



Flipping through a reference book is the best way to find out what's in the Java library. You can easily stumble on a class that looks useful, just by browsing pages.

class name
package name
class description

java.util.Currency

Returned By: `java.text.DecimalFormat.getCurrency()`, `java.text.DecimalFormatSymbols.getCurrency()`,
`java.text.NumberFormat.getCurrency()`, `Currency.getInstance()`

Date Java 1.0
cloneable serializable comparable

This class represents dates and times and lets you work with them in a system-independent way. You can create a Date by specifying the number of milliseconds from the epoch (midnight GMT, January 1st, 1970) or the year, month, date, and, optionally, the hour, minute, and second. Years are specified as the number of years since 1900. If you call the Date constructor with no arguments, the Date is initialized to the current time and date. The instance methods of the class allow you to get and set the various date and time fields, to compare dates and times, and to convert dates to and from string representations. As of Java 1.1, many of the date methods have been deprecated in favor of the methods of the Calendar class.

Object	Date
Cloneable	Comparable
Serializable	

```

public class Date implements Cloneable, Comparable, Serializable {
    // Public Constructors
    public Date();
    public Date(long date);
    public Date(String s);
    public Date(int year, int month, int date);
    public Date(int year, int month, int date, int hrs, int min);
    public Date(int year, int month, int date, int hrs, int min, int sec);
    // Property Accessor Methods (by property name)
    public long getTime();
    public void setTime(long time);
    // Public Instance Methods
    public boolean after(java.util.Date when);
    public boolean before(java.util.Date when);
    // public int compareTo(java.util.Date anotherDate);
    // Methods Implementing Comparable
    // public int compareTo(Object o);
    // Public Methods Overriding Object
    // public Object clone();
    public boolean equals(Object obj);
    public int hashCode();
    public String toString();
    // Deprecated Public Methods
    // public int getDate();
    // public int getDay();
    // public int getHours();
    // public int getMinutes();
    // public int getMonth();
    // public int getSeconds();
    // public int getTimezoneOffset();
    // public int getYear();
    // public static long parse(String s);
    public void setDate(int date);
    public void setHours(int hours);
    public void setMinutes(int minutes);
    public void setMonth(int month);
}

```

methods (and other things we'll talk about later)

using the Java API documentation

2

Use the HTML API docs

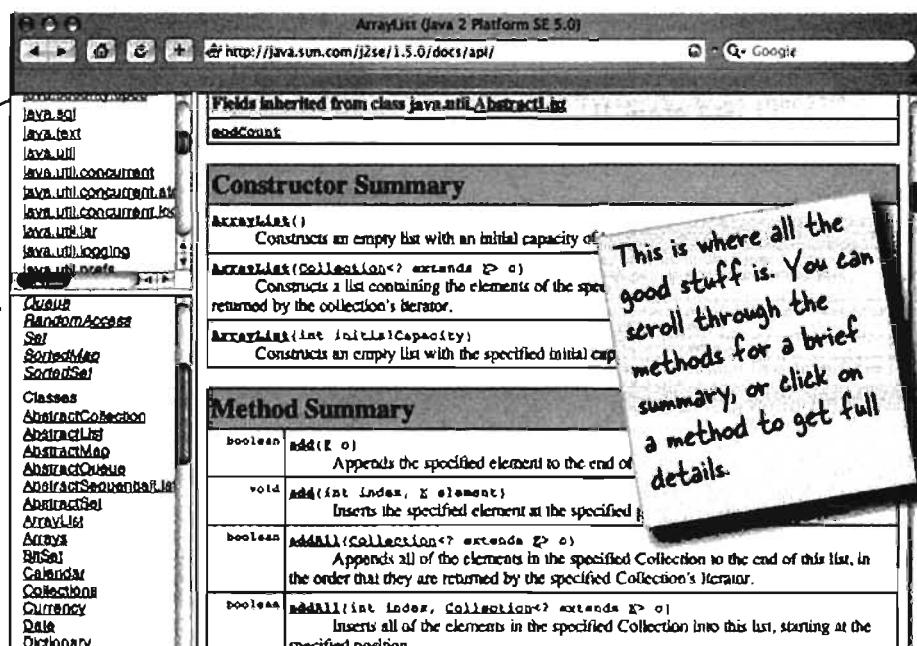
Java comes with a fabulous set of online docs called, strangely, the Java API. They're part of a larger set called the Java 5 Standard Edition Documentation (which, depending on what day of the week you look, Sun may be referring to as "Java 2 Standard Edition 5.0"), and you have to download the docs separately; they don't come shrink-wrapped with the Java 5 download. If you have a high-speed internet connection, or tons of patience, you can also browse them at java.sun.com. Trust us, you probably want these on your hard drive.

The API docs are the best reference for getting more details about a class and its methods. Let's say you were browsing through the reference book and found a class called `Calendar`, in `java.util`. The book tells you a little about it, enough to know that this is indeed what you want to use, but you still need to know more about the methods.

The reference book, for example, tells you what the methods take, as arguments, and what they return. Look at `ArrayList`, for example. In the reference book, you'll find the method `indexOf()`, that we used in the `DotCom` class. But if all you knew is that there is a method called `indexOf()` that takes an object and returns the index (an int) of that object, you still need to know one crucial thing: what happens if the object is not in the `ArrayList`? Looking at the method signature alone won't tell you how that works. But the API docs will (most of the time, anyway). The API docs tell you that the `indexOf()` method returns a -1 if the object parameter is not in the `ArrayList`. That's how we knew we could use it both as a way to check if an object is even in the `ArrayList`, and to get its index at the same time, if the object was there. But without the API docs, we might have thought that the `indexOf()` method would blow up if the object wasn't in the `ArrayList`.

Scroll through the packages and select one (click it) to restrict the list in the lower frame to only classes from that package.

Scroll through the classes and select one (click it) to choose the class that will fill the main browser frame.





Code Magnets

Can you reconstruct the code snippets to make a working Java program that produces the output listed below? **NOTE:** To do this exercise, you need one NEW piece of info—if you look in the API for `ArrayList`, you'll find a second `add` method that takes two arguments:

`add(int index, Object o)`

It lets you specify to the `ArrayList` where to put the object you're adding.

```
a.remove(2);
printAL(a);
printAL(a);
a.add(0, "zero");
a.add(1, "one");
```

```
public static void printAL(ArrayList<String> al) {
```

```
    if (a.contains("two")) {
        a.add("2.2");
    }
```

```
    a.add(2, "two");
```

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

```
    System.out.print(element + " ");
}
System.out.println(" ");
```

```
    if (a.contains("three")) {
        a.add("four");
    }
```

```
public class ArrayListMagnet {
```

```
    if (a.indexOf("four") != 4) {
        a.add(4, "4.2");
    }
}
```

```
import java.util.*;
```

```
printAL(a);
```

```
ArrayList<String> a = new ArrayList<String>();
```

```
for (String element : al) {
```

```
    a.add(3, "three");
    printAL(a);
}
```

```
File Edit Window Help Dance
i java ArrayListMagnet
zero one two three
zero one three four
zero one three four 4.2
zero one three four 4.2
```

puzzle: crossword



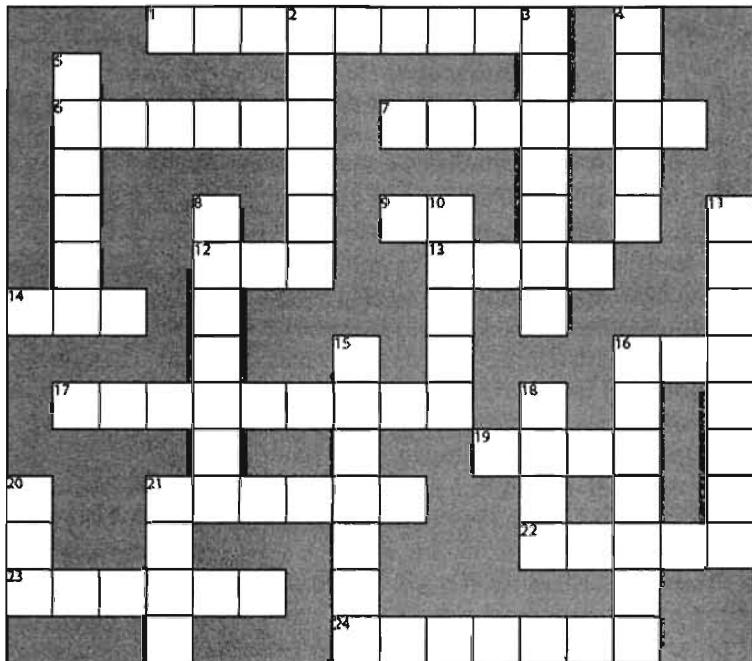
JavaCross 7.0

How does this crossword puzzle help you learn Java? Well, all of the words are Java related (except one red herring).

Hint: When in doubt, remember `ArrayList`.

Across

1. I can't behave
6. Or, in the courtroom
7. Where it's at baby
9. A fork's origin
12. Grow an `ArrayList`
13. Wholly massive
14. Value copy
16. Not an object
17. An array on steroids
19. Extent
21. 19's counterpart
22. Spanish geek snacks (Note: This has nothing to do with Java.)
23. For lazy fingers
24. Where packages roam



Down

2. Where the Java action is.
3. Addressable unit
4. 2nd smallest
5. Fractional default
8. Library's grandest
10. Must be low density
11. He's in there somewhere
15. As if
16. dearth method
18. What shopping and arrays have in common
20. Library acronym
21. What goes around

More Hints:

- | | | | | | | |
|---|---------------------------------|---|---|----------------------|---------------------|---|
| 1. Variables | 2. What's overridable? | 3. Think <code>ArrayList</code> | 4. & 10. Primitive | 16. Common primitive | 21. Array's parent | 22. Not about Java - Spanish appetizers |
| 5. Think <code>ArrayList</code> | 6. Think <code>ArrayList</code> | 7. Think <code>ArrayList</code> | 8. 10. Primitive | 17. Common primitive | 20. Library acronym | 18. He's making a |
| 9. 10. Primitive | 11. He's making a | 12. Not about Java - Spanish appetizers | 13. Wholly massive | 14. Value copy | 19. Extent | 23. For lazy fingers |
| 15. As if | 16. dearth method | 17. Common primitive | 18. What shopping and arrays have in common | 19. Extent | 20. Library acronym | 21. What goes around |
| 22. Not about Java - Spanish appetizers | 23. For lazy fingers | 24. Where packages roam | | | | |



Exercise Solutions

```
File Edit Window Help Dance
t java ArrayListMagnet
zero one two three
zero one three four
zero one three four 4.2
zero one three four 4.2
```

```
import java.util.*;

public class ArrayListMagnet {

    public static void main (String[] args) {

        ArrayList<String> a = new ArrayList<String>();

        a.add(0,"zero");
        a.add(1,"one");

        a.add(2,"two");

        a.add(3,"three");
        printAL(a);

        if (a.contains("three")) {
            a.add("four");
        }

        a.remove(2);
        printAL(a);

        if (a.indexOf("four") != 4) {
            a.add(4, "4.2");
        }

        printAL(a);

        if (a.contains("two")) {
            a.add("2.2");
        }

        printAL(a);
    }

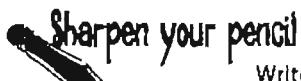
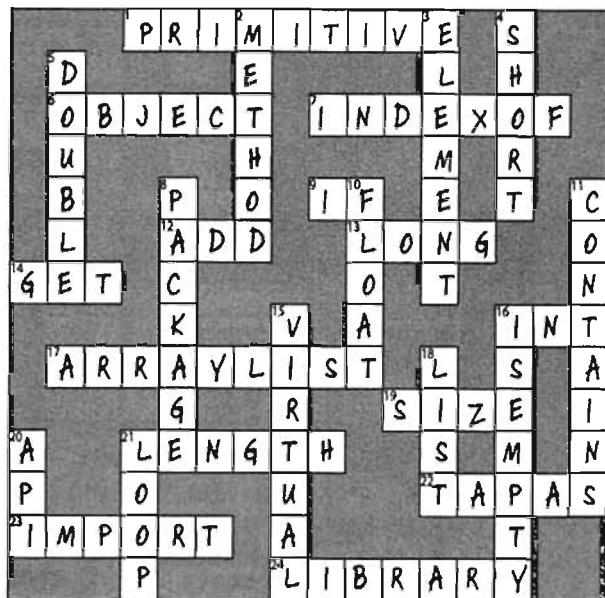
    public static void printAL(ArrayList<String> al) {

        for (String element : al) {

            System.out.print(element + " ");
        }
        System.out.println(" ");
    }
}
```



JavaCross answers



Write your OWN set of clues! Look at each word, and try to write your own clues. Try making them easier, or harder, or more technical than the ones we have.

Across

1. _____
6. _____
7. _____
9. _____
12. _____
13. _____
14. _____
16. _____
17. _____
19. _____
21. _____
22. _____
23. _____
24. _____

Down

2. _____
3. _____
4. _____
5. _____
8. _____
10. _____
11. _____
15. _____
16. _____
18. _____
20. _____
21. _____

Better Living in Objectville



We were underpaid,
overworked coders 'till we
tried the Polymorphism Plan. But
thanks to the Plan, our future is
bright. Yours can be too!

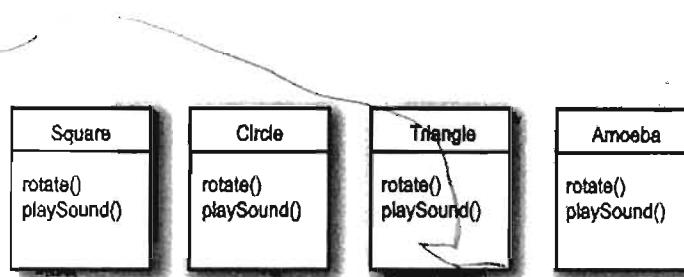
Plan your programs with the future in mind. If there were a way to write Java code such that you could take more vacations, how much would it be worth to you? What if you could write code that someone else could extend, easily? And if you could write code that was flexible, for those pesky last-minute spec changes, would that be something you're interested in? Then this is your lucky day. For just three easy payments of 60 minutes time, you can have all this. When you get on the Polymorphism Plan, you'll learn the 5 steps to better class design, the 3 tricks to polymorphism, the 8 ways to make flexible code, and if you act now—a bonus lesson on the 4 tips for exploiting inheritance. Don't delay, an offer this good will give you the design freedom and programming flexibility you deserve. It's quick, it's easy, and it's available now. Start today, and we'll throw in an extra level of abstraction!

Chair Wars Revisited...

Remember way back in chapter 2, when Larry (procedural guy) and Brad (OO guy) were vying for the Aeron chair? Let's look at a few pieces of that story to review the basics of inheritance.

LARRY: You've got duplicated code! The rotate procedure is in all four Shape things. It's a stupid design. You have to maintain four different rotate "methods". How can that ever be good?

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

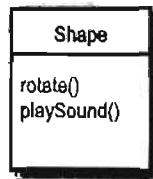


1

I looked at what all four classes have in common.

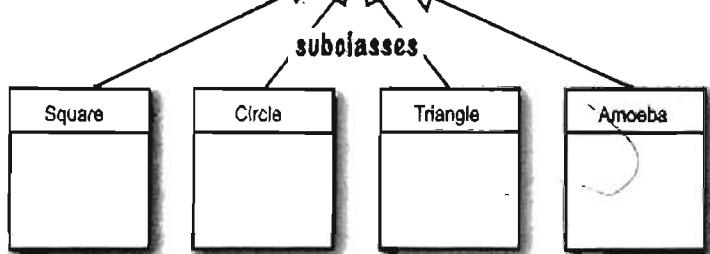
2

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.

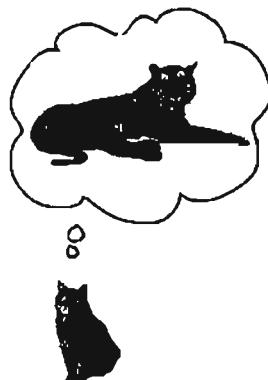
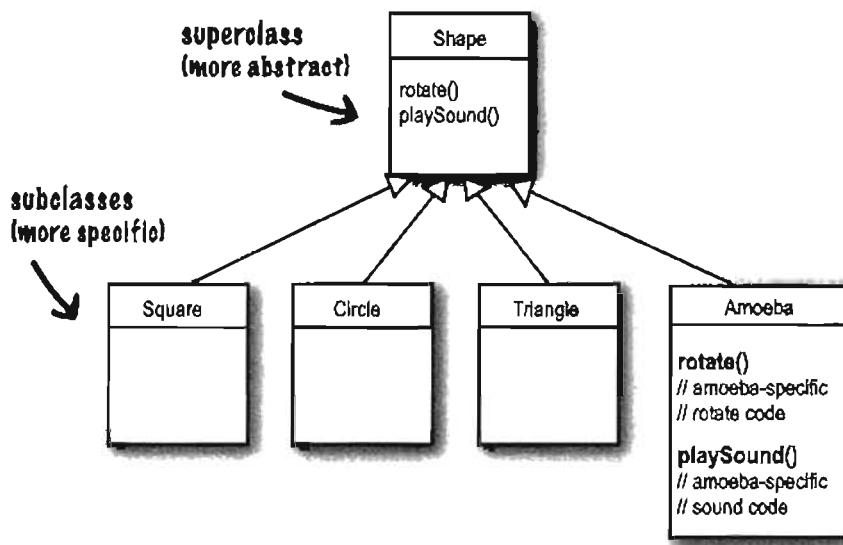
The Shape class is called the **superclass** of the other four classes. The other four are the **subclasses** of Shape. The subclasses inherit the methods of the superclass. In other words, if the Shape class has the functionality, then the subclasses automatically get that same functionality.

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?

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.



How would you represent a house cat and a tiger, in an inheritance structure. Is a domestic cat a specialized version of a tiger? Which would be the subclass and which would be the superclass? Or are they both subclasses to some other class?

How would you design an inheritance structure? What methods would be overridden?

Think about it. Before you turn the page.

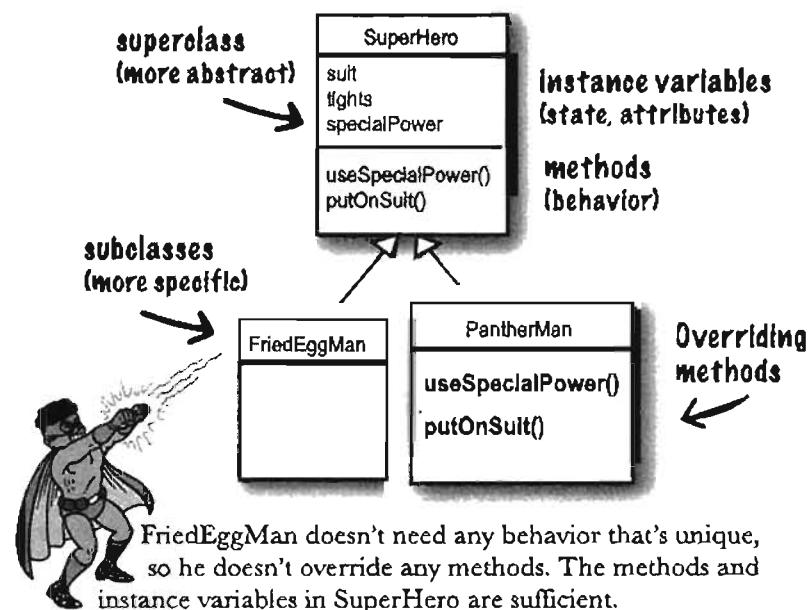
3 way inheritance works

Understanding Inheritance

When you design with inheritance, you put common code in a class and then tell other more specific classes that the common (more abstract) class is their superclass. When one class inherits from another, the **subclass inherits from the superclass**.

In Java, we say that the **subclass extends the superclass**. An inheritance relationship means that the subclass inherits the **members** of the superclass. When we say “members of a class” we mean the instance variables and methods.

For example, if PantherMan is a subclass of SuperHero, the PantherMan class automatically inherits the instance variables and methods common to all superheroes including suit, tights, specialPower, useSpecialPower () and so on. But the PantherMan **subclass can add new methods and instance variables** of its own, and it **can override the methods it inherits from the superclass** SuperHero.



FriedEggMan doesn't need any behavior that's unique, so he doesn't override any methods. The methods and instance variables in SuperHero are sufficient.

PantherMan, though, has specific requirements for his suit and special powers, so useSpecialPower () and putOnSuit () are both overridden in the PantherMan class.

Instance variables are not overridden because they don't need to be. They don't define any special behavior, so a subclass can give an inherited instance variable any value it chooses. PantherMan can set his inherited tights to purple, while FriedEggMan sets his to white.

inheritance and polymorphism

An inheritance example:

```

public class Doctor {
    boolean worksAtHospital;

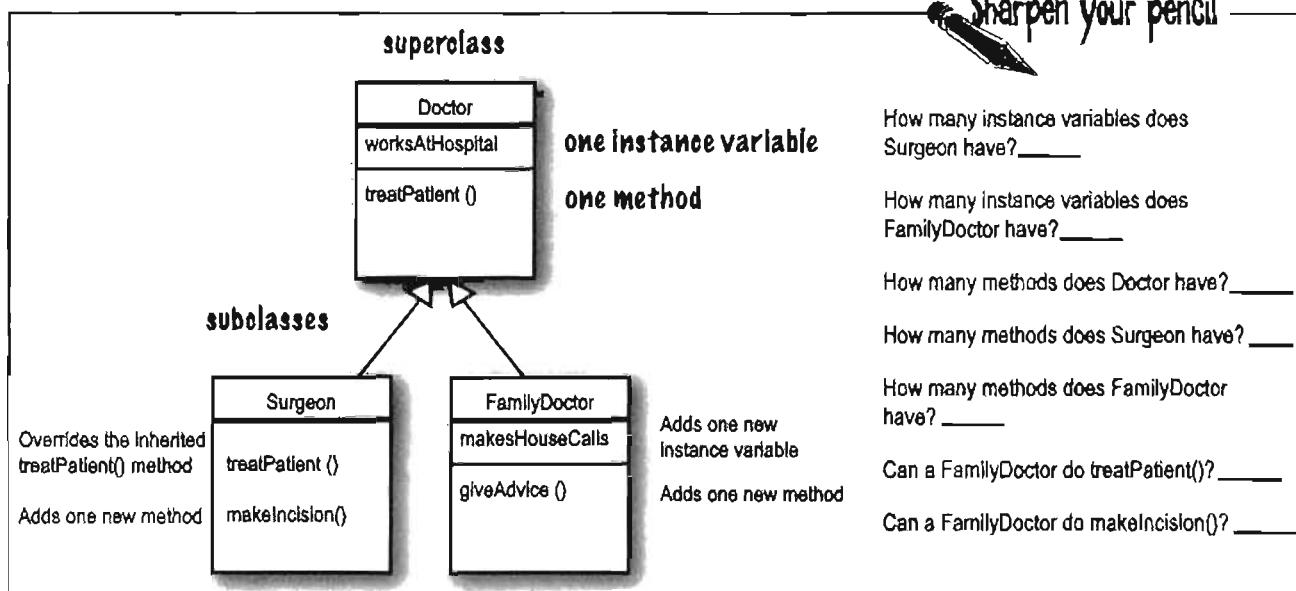
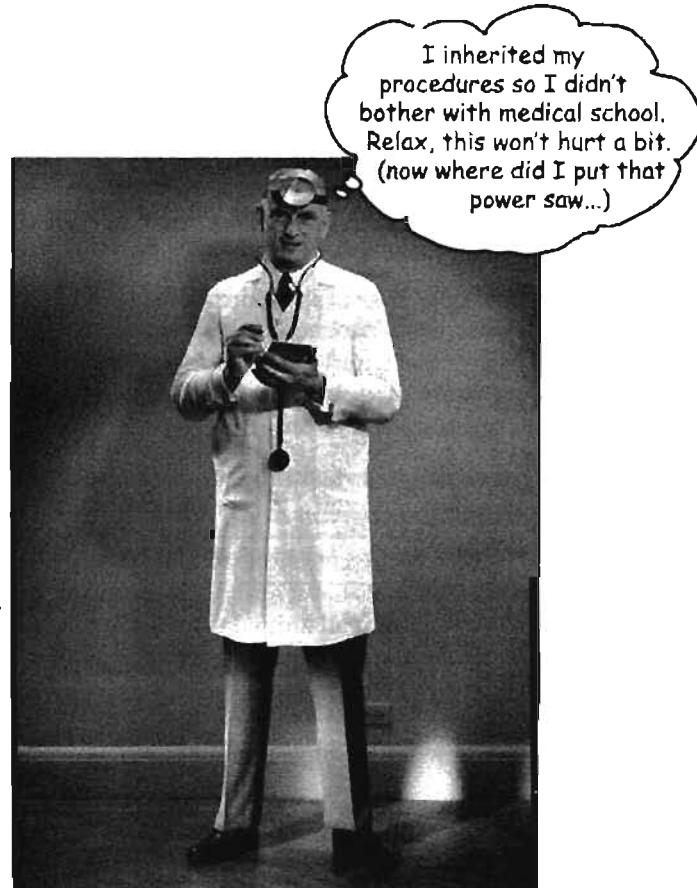
    void treatPatient() {
        // perform a checkup
    }
}

public class FamilyDoctor extends Doctor {
    boolean makesHouseCalls;
    void giveAdvice() {
        // give homespun advice
    }
}

public class Surgeon extends Doctor{
    void treatPatient() {
        // perform surgery
    }

    void makeIncision() {
        // make incision (yikes!)
    }
}

```



Let's design the inheritance tree for an Animal simulation program

Imagine you're asked to design a simulation program that lets the user throw a bunch of different animals into an environment to see what happens. We don't have to code the thing now, we're mostly interested in the design.

We've been given a list of *some* of the animals that will be in the program, but not all. We know that each animal will be represented by an object, and that the objects will move around in the environment, doing whatever it is that each particular type is programmed to do.

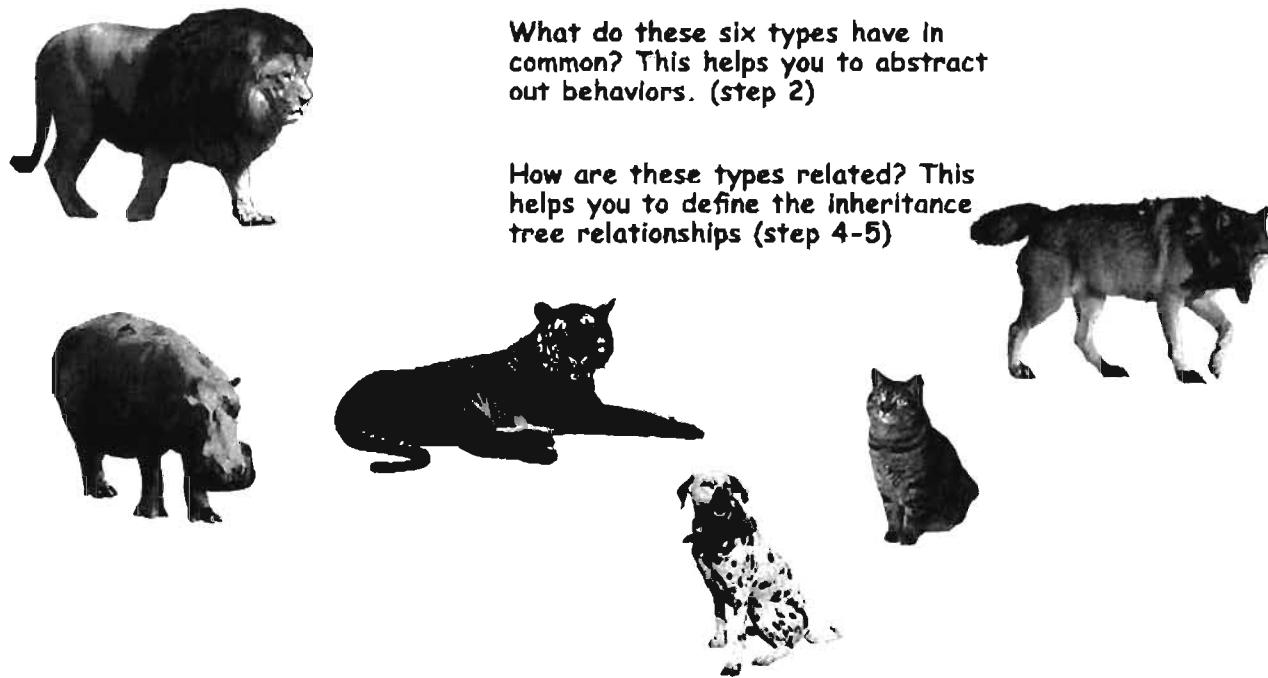
And we want other programmers to be able to add new kinds of animals to the program at any time.

First we have to figure out the common, abstract characteristics that all animals have, and build those characteristics into a class that all animal classes can extend.

- 1 Look for objects that have common attributes and behaviors.

What do these six types have in common? This helps you to abstract out behaviors. (step 2)

How are these types related? This helps you to define the inheritance tree relationships (step 4-5)



Using inheritance to avoid duplicating code in subclasses

We have five *instance variables*:

picture – the file name representing the JPEG of this animal
food – the type of food this animal eats. Right now, there can be only two values: *meat* or *grass*.

hunger – an int representing the hunger level of the animal. It changes depending on when (and how much) the animal eats.

boundaries – values representing the height and width of the ‘space’ (for example, 640 x 480) that the animals will roam around in.

location – the X and Y coordinates for where the animal is in the space.

We have four *methods*:

makeNoise() – behavior for when the animal is supposed to make noise.

eat() – behavior for when the animal encounters its preferred food source, *meat* or *grass*.

sleep() – behavior for when the animal is considered asleep.

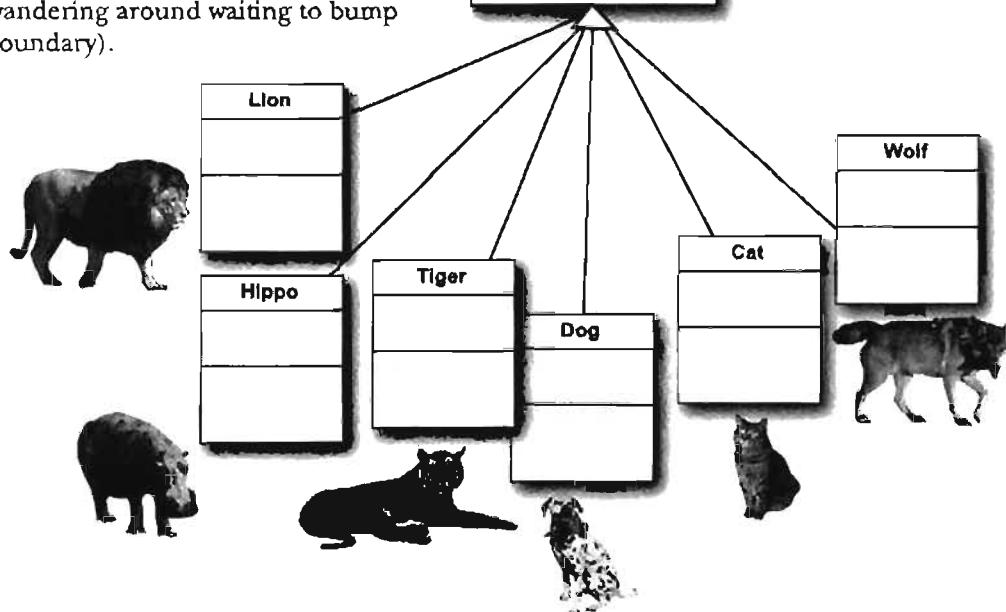
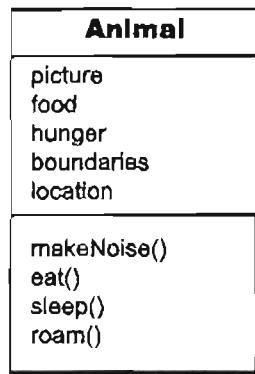
roam() – behavior for when the animal is not eating or sleeping (probably just wandering around waiting to bump into a food source or a boundary).

2

Design a class that represents the common state and behavior.

These objects are all animals, so we’ll make a common superclass called *Animal*.

We’ll put in methods and instance variables that all animals might need.



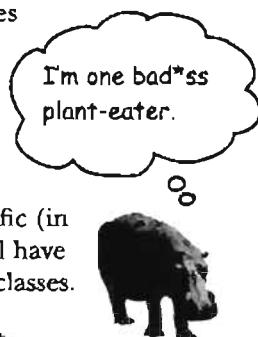
Do all animals eat the same way?

Assume that we all agree on one thing: the instance variables will work for *all* Animal types. A lion will have his own value for picture, food (we're thinking meat), hunger, boundaries, and location. A hippo will have different values for his instance variables, but he'll still have the same variables that the other Animal types have. Same with dog, tiger, and so on. But what about *behavior*?

Which methods should we override?

Does a lion make the same noise as a dog? Does a cat eat like a hippo? Maybe in *your* version, but in ours, eating and making noise are Animal-type-specific. We can't figure out how to code those methods in such a way that they'd work for any animal. OK, that's not true. We could write the makeNoise() method, for example, so that all it does is play a sound file defined in an instance variable for that type, but that's not very specialized. Some animals might make different noises for different situations (like one for eating, and another when bumping into an enemy, etc.)

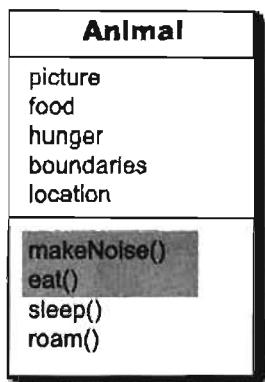
So just as with the Amoeba overriding the Shape class rotate() method, to get more amoeba-specific (in other words, *unique*) behavior, we'll have to do the same for our Animal subclasses.



3

Decide if a subclass needs behaviors (method implementations) that are specific to that particular subclass type.

Looking at the Animal class, we decide that eat() and makeNoise() should be overridden by the individual subclasses.



We better override these two methods, eat() and makeNoise(), so that each animal type can define its own specific behavior for eating and making noise. For now, it looks like sleep() and roam() can stay generic.

Looking for more inheritance opportunities

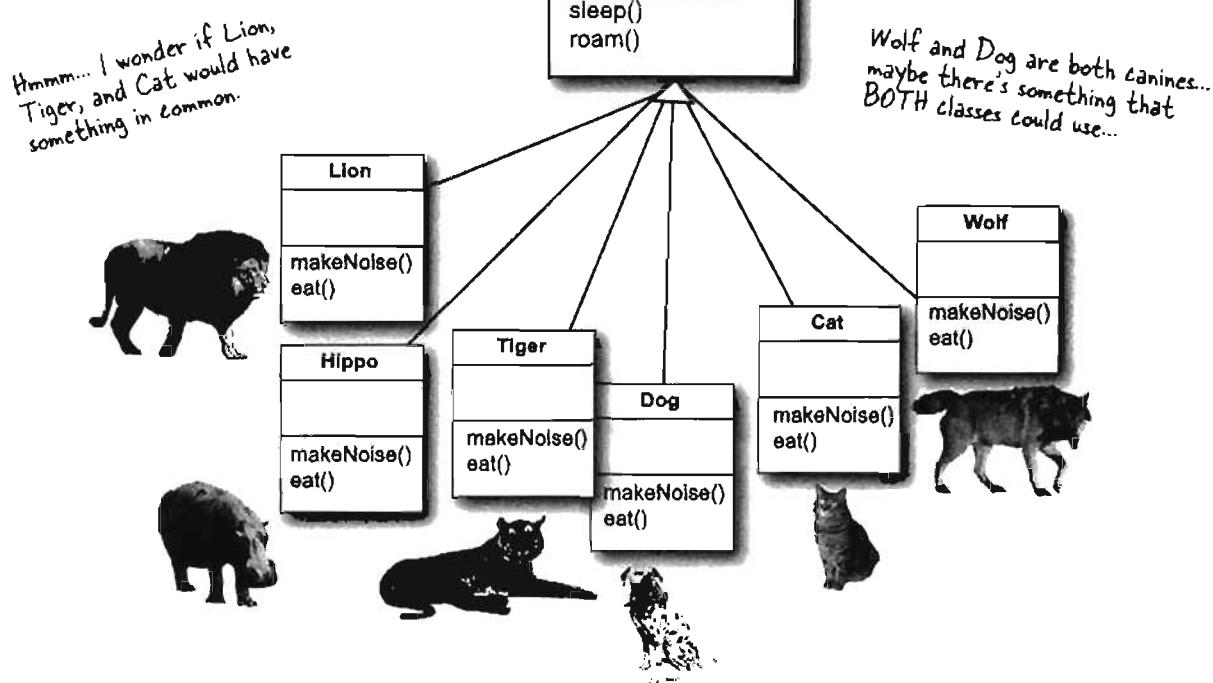
The class hierarchy is starting to shape up. We have each subclass override the `makeNoise()` and `eat()` methods, so that there's no mistaking a Dog bark from a Cat meow (quite insulting to both parties). And a Hippo won't eat like a Lion.

But perhaps there's more we can do. We have to look at the subclasses of `Animal`, and see if two or more can be grouped together in some way, and given code that's common to only *that* new group. Wolf and Dog have similarities. So do Lion, Tiger, and Cat.

4

Look for more opportunities to use abstraction, by finding two or more subclasses that might need common behavior.

We look at our classes and see that Wolf and Dog might have some behavior in common, and the same goes for Lion, Tiger, and Cat.



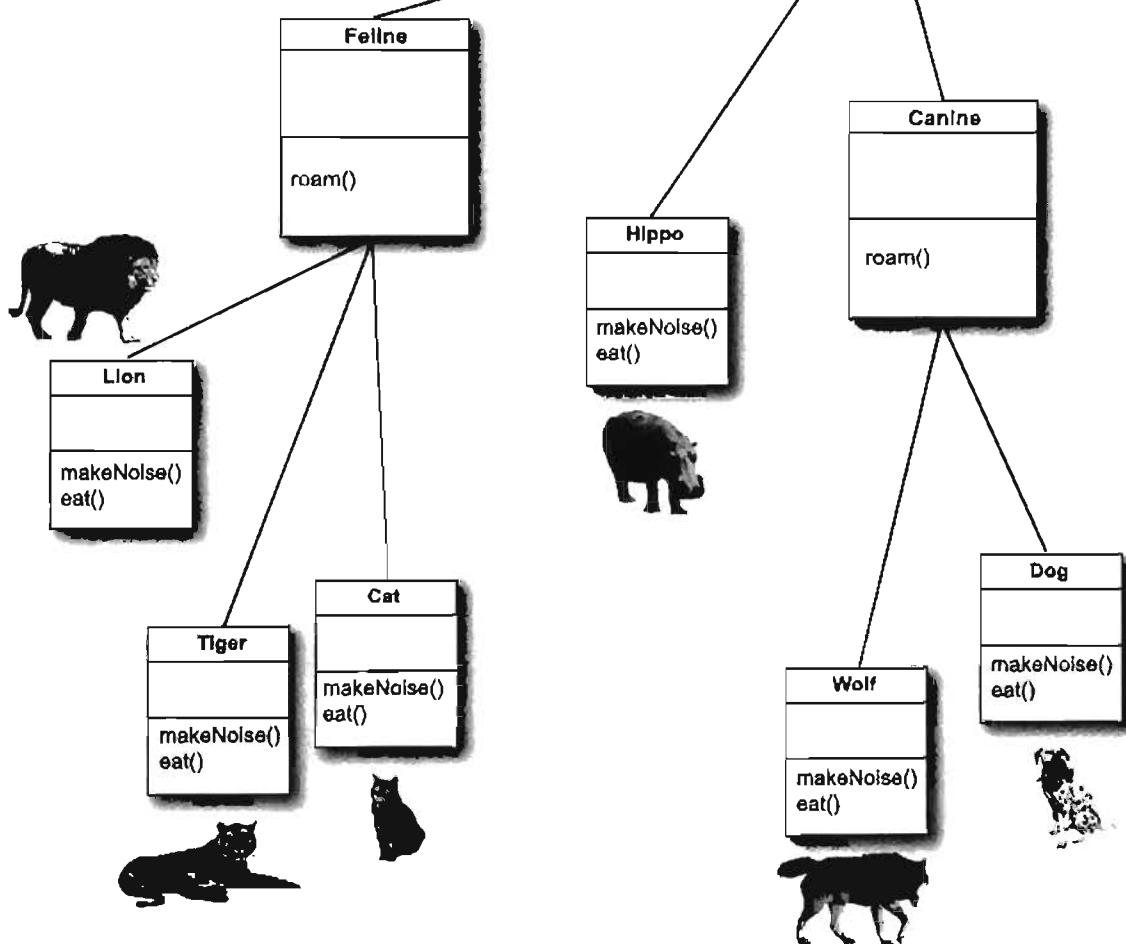
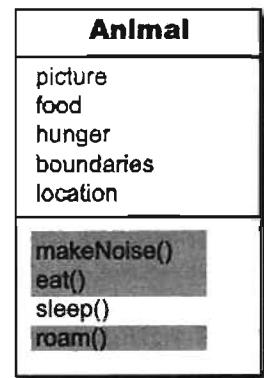
designing for inheritance

5 Finish the class hierarchy

Since animals already have an organizational hierarchy (the whole kingdom, genus, phylum thing), we can use the level that makes the most sense for class design. We'll use the biological "families" to organize the animals by making a Feline class and a Canine class.

We decide that Canines could use a common `roam()` method, because they tend to move in packs. We also see that Felines could use a common `roam()` method, because they tend to avoid others of their own kind. We'll let Hippo continue to use its inherited `roam()` method—the generic one it gets from Animal.

So we're done with the design for now; we'll come back to it later in the chapter.



Which method is called?

The Wolf class has four methods. One inherited from Animal, one inherited from Canine (which is actually an overridden version of a method in class Animal), and two overridden in the Wolf class. When you create a Wolf object and assign it to a variable, you can use the dot operator on that reference variable to invoke all four methods. But which *version* of those methods gets called?

make a new Wolf object

```
Wolf w = new Wolf();
```

calls the version in Wolf

```
w.makeNoise();
```

calls the version in Canine

```
w.roam();
```

calls the version in Wolf

```
w.eat();
```

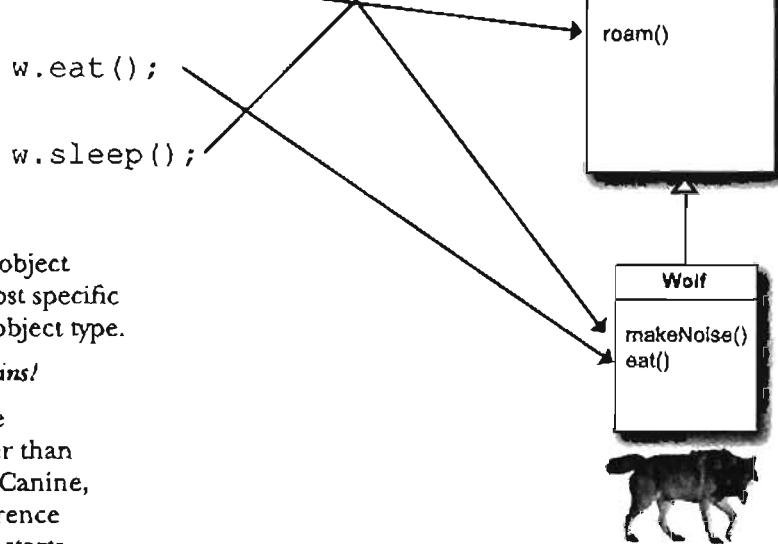
calls the version in Animal

```
w.sleep();
```

When you call a method on an object reference, you're calling the most specific version of the method for that object type.

In other words, *the lowest one wins!*

"Lowest" meaning lowest on the inheritance tree. Canine is lower than Animal, and Wolf is lower than Canine, so invoking a method on a reference to a Wolf object means the JVM starts looking first in the Wolf class. If the JVM doesn't find a version of the method in the Wolf class, it starts walking back up the inheritance hierarchy until it finds a match.



practice designing an inheritance tree

Designing an Inheritance Tree

Class	Superclasses	Subclasses
Clothing	—	Boxers, Shirt
Boxers	Clothing	
Shirt	Clothing	

Inheritance Table

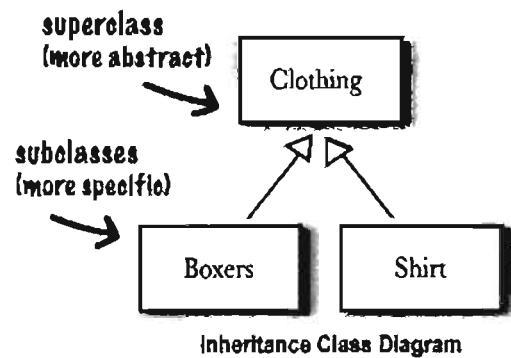


Find the relationships that make sense. Fill in the last two columns

Class	Superclasses	Subclasses
Musician		
Rock Star		
Fan		
Bass Player		
Concert Pianist		

Hint: not everything can be connected to something else.

Hint: you're allowed to add to or change the classes listed.



Draw an inheritance diagram here.

there are no
Dumb Questions

Q: You said that the JVM starts walking up the inheritance tree, starting at the class type you invoked the method on (like the Wolf example on the previous page). But what happens if the JVM doesn't ever find a match?

A: Good question! But you don't have to worry about that. The compiler guarantees that a particular method is callable for a specific reference type, but it doesn't say (or care) from which class that method actually comes from at runtime. With the Wolf example, the compiler checks for a sleep() method, but doesn't care that sleep() is actually defined in (and inherited from) class Animal. Remember that if a class inherits a method, it has the method.

Where the inherited method is defined (in other words, in which superclass it is defined) makes no difference to the compiler. But at runtime, the JVM will always pick the right one. And the right one means, the most specific version for that particular object.

Using IS-A and HAS-A

Remember that when one class inherits from another, we say that the subclass *extends* the superclass. When you want to know if one thing should extend another, apply the IS-A test.

Triangle IS-A Shape, yeah, that works.

Cat IS-A Feline, that works too.

Surgeon IS-A Doctor, still good.

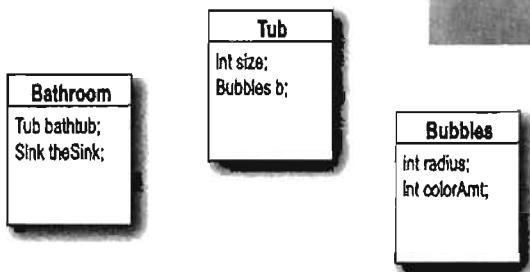
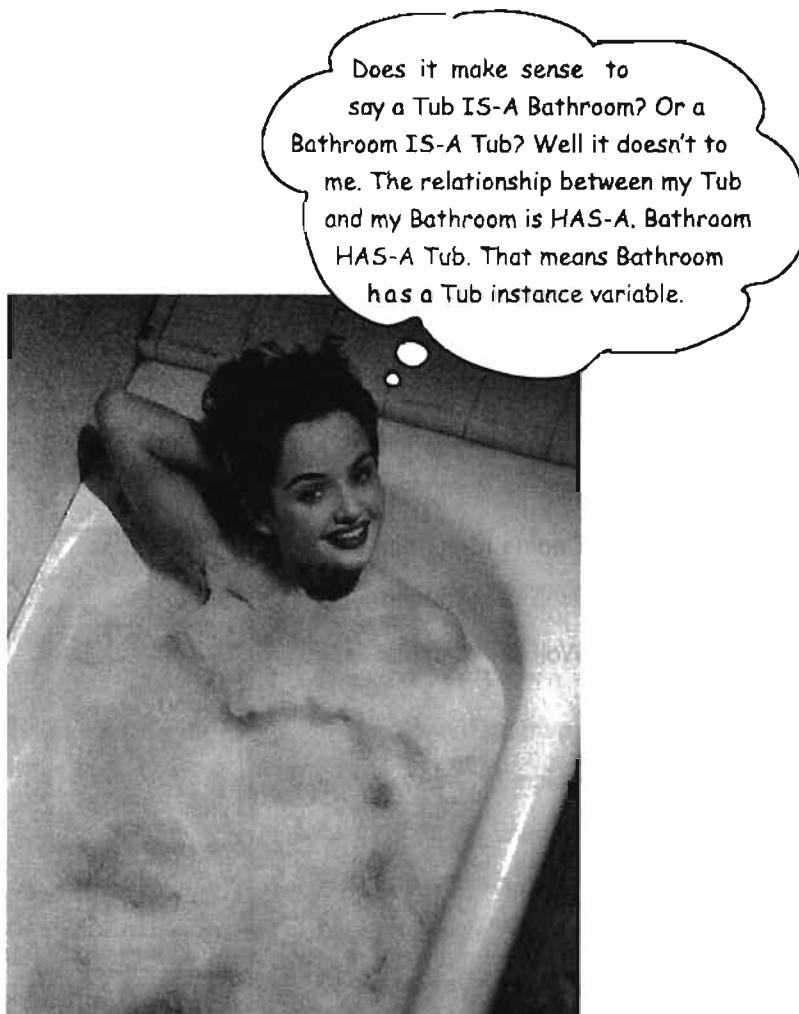
Tub extends Bathroom, sounds reasonable.

Until you apply the IS-A test.

To know if you've designed your types correctly, ask, "Does it make sense to say type X IS-A type Y?" If it doesn't, you know there's something wrong with the design, so if we apply the IS-A test, Tub IS-A Bathroom is definitely false.

What if we reverse it to Bathroom extends Tub? That still doesn't work, Bathroom IS-A Tub doesn't work.

Tub and Bathroom *are* related, but not through inheritance. Tub and Bathroom are joined by a HAS-A relationship. Does it make sense to say "Bathroom HAS-A Tub"? If yes, then it means that Bathroom has a Tub instance variable. In other words, Bathroom has a *reference* to a Tub, but Bathroom does not *extend* Tub and vice-versa.



Bathroom HAS-A Tub and Tub HAS-A Bubbles.
But nobody inherits from (extends) anybody else.

But wait! There's more!

The IS-A test works *anywhere* in the inheritance tree. If your inheritance tree is well-designed, the IS-A test should make sense when you ask *any* subclass if it IS-A *any* of its supertypes.

If class B extends class A, class B IS-A class A.

This is true anywhere in the inheritance tree. If class C extends class B, class C passes the IS-A test for both B and A.

Canine extends Animal

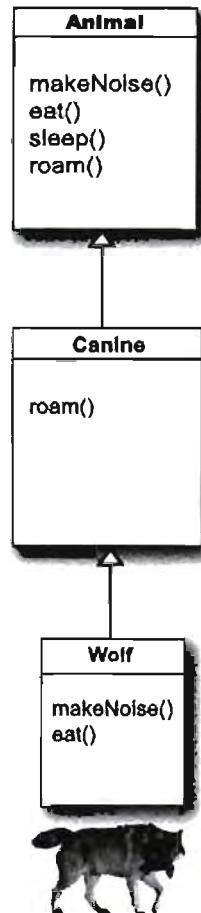
Wolf extends Canine

Wolf extends Animal

Canine IS-A Animal

Wolf IS-A Canine

Wolf IS-A Animal



With an inheritance tree like the one shown here, you're *always* allowed to say "Wolf extends Animal" or "Wolf IS-A Animal". It makes no difference if Animal is the superclass of the superclass of Wolf. In fact, as long as Animal is *somewhere* in the inheritance hierarchy above Wolf, Wolf IS-A Animal will always be true.

The structure of the Animal inheritance tree says to the world: "Wolf IS-A Canine, so Wolf can do anything a Canine can do. And Wolf IS-A Animal, so Wolf can do anything an Animal can do."

It makes no difference if Wolf overrides some of the methods in Animal or Canine. As far as the world (of other code) is concerned, a Wolf can do those four methods. *How* he does them, or *in which class they're overridden* makes no difference. A Wolf can `makeNoise()`, `eat()`, `sleep()`, and `roam()` because a Wolf extends from class Animal.

How do you know if you've got your inheritance right?

There's obviously more to it than what we've covered so far, but we'll look at a lot more OO issues in the next chapter (where we eventually refine and improve on some of the design work we did in *this* chapter).

For now, though, a good guideline is to use the IS-A test. If "X IS-A Y" makes sense, both classes (X and Y) should probably live in the same inheritance hierarchy. Chances are, they have the same or overlapping behaviors.

Keep in mind that the inheritance IS-A relationship works in only one direction!

Triangle IS-A Shape makes sense, so you can have Triangle extend Shape.

But the reverse—Shape IS-A Triangle—does *not* make sense, so Shape should not extend Triangle. Remember that the IS-A relationship implies that if X IS-A Y, then X can do anything a Y can do (and possibly more).



Sharpen your pencil

Put a check next to the relationships that make sense.

- Oven extends Kitchen
- Guitar extends Instrument
- Person extends Employee
- Ferrari extends Engine
- FriedEgg extends Food
- Beagle extends Pet
- Container extends Jar
- Metal extends Titanium
- GratefulDead extends Band
- Blonde extends Smart
- Beverage extends Martini

Hint: apply the IS-A test

who inherits what

~~there are no~~ Dumb Questions

Q: So we see how a subclass gets to inherit a superclass method, but what if the superclass wants to use the subclass version of the method?

A: A superclass won't necessarily know about any of its subclasses. You might write a class and much later someone else comes along and extends it. But even if the superclass creator does know about (and wants to use) a subclass version of a method, there's no sort of reverse or backwards inheritance. Think about it, children inherit from parents, not the other way around.

Q: In a subclass, what if I want to use BOTH the superclass version and my overriding subclass version of a method? In other words, I don't want to completely replace the superclass version, I just want to add more stuff to it.

A: You can do this! And it's an important design feature. Think of the word "extends" as meaning, "I want to extend the functionality of the superclass".

```
public void roam() {  
    super.roam();  
    // my own roam stuff  
}
```

You can design your superclass methods in such a way that they contain method implementations that will work for any subclass, even though the subclasses may still need to 'append' more code. In your subclass overriding method, you can call the superclass version using the keyword `super`. It's like saying, "first go run the superclass version, then come back and finish with my own code..."

this calls the inherited version of
roam(), then comes back to do
your own subclass-specific code

Who gets the Porsche, who gets the porcelain? (how to know what a subclass can inherit from its superclass)



A subclass inherits members of the superclass. Members include instance variables and methods, although later in this book we'll look at other inherited members. A superclass can choose whether or not it wants a subclass to inherit a particular member by the level of access the particular member is given.

There are four access levels that we'll cover in this book. Moving from most restrictive to least, the four access levels are:

private default protected public

Access levels control *who sees what*, and are crucial to having well-designed, robust Java code. For now we'll focus just on public and private. The rules are simple for those two:

public members are inherited
private members are not inherited

When a subclass inherits a member, it is *as if the subclass defined the member itself*. In the Shape example, Square inherited the `rotate()` and `playSound()` methods and to the outside world (other code) the Square class simply *has a* `rotate()` and `playSound()` method.

The members of a class include the variables and methods defined in the class plus anything inherited from a superclass.

Note: get more details about default and protected in chapter 16 (deployment) and appendix B.

When designing with inheritance, are you using or abusing?

Although some of the reasons behind these rules won't be revealed until later in this book, for now, simply *knowing* a few rules will help you build a better inheritance design.

DO use inheritance when one class is a more specific type of a superclass. Example: Willow is a more specific type of Tree, so Willow extends Tree makes sense.

DO consider inheritance when you have behavior (implemented code) that should be shared among multiple classes of the same general type. Example: Square, Circle, and Triangle all need to rotate and play sound, so putting that functionality in a superclass Shape might make sense, and makes for easier maintenance and extensibility. Be aware, however, that while inheritance is one of the key features of object-oriented programming, it's not necessarily the best way to achieve behavior reuse. It'll get you started, and often it's the right design choice, but design patterns will help you see other more subtle and flexible options. If you don't know about design patterns, a good follow-on to this book would be *Head First Design Patterns*.

DO NOT use inheritance just so that you can reuse code from another class, if the relationship between the superclass and subclass violate either of the above two rules. For example, imagine you wrote special printing code in the Alarm class and now you need printing code in the Piano class, so you have Piano extend Alarm so that Piano inherits the printing code. That makes no sense! A Piano is *not* a more specific type of Alarm. (So the printing code should be in a Printer class, that all printable objects can take advantage of via a HAS-A relationship.)

DO NOT use inheritance if the subclass and superclass do not pass the IS-A test. Always ask yourself if the subclass IS-A more specific type of the superclass. Example: Tea IS-A Beverage makes sense. Beverage IS-A Tea does not.

BULLET POINTS



- A subclass *extends* a superclass.
- A subclass *inherits* all *public* instance variables and methods of the superclass, but does not inherit the *private* instance variables and methods of the superclass.
- Inherited methods *can* be overridden; instance variables *cannot* be overridden (although they can be *redefined* in the subclass, but that's not the same thing, and there's almost never a need to do it.)
- Use the IS-A test to verify that your inheritance hierarchy is valid. If X extends Y, then X IS-A Y must make sense.
- The IS-A relationship works in only one direction. A Hippo is an Animal, but not all Animals are Hippos.
- When a method is overridden in a subclass, and that method is invoked on an instance of the subclass, the overridden version of the method is called. (*The lowest one wins.*)
- If class B extends A, and C extends B, class B IS-A class A, and class C IS-A class B, and class C also IS-A class A.

So what does all this inheritance really buy you?

You get a lot of OO mileage by designing with inheritance. You can get rid of duplicate code by abstracting out the behavior common to a group of classes, and sticking that code in a superclass. That way, when you need to modify it, you have only one place to update, and *the change is magically reflected in all the classes that inherit that behavior*. Well, there's no magic involved, but it is pretty simple: make the change and compile the class again. That's it. **You don't have to touch the subclasses!**

Just deliver the newly-changed superclass, and all classes that extend it will automatically use the new version.

A Java program is nothing but a pile of classes, so the subclasses don't have to be recompiled in order to use the new version of the superclass. As long as the superclass doesn't *break* anything for the subclass, everything's fine. (We'll discuss what the word 'break' means in this context, later in the book. For now, think of it as modifying something in the superclass that the subclass is depending on, like a particular method's arguments or return type, or method name, etc.)

➊ You avoid duplicate code.

Put common code in one place, and let the subclasses inherit that code from a superclass. When you want to change that behavior, you have to modify it in only one place, and everybody else (i.e. all the subclasses) see the change.

➋ You define a common protocol for a group of classes.



Inheritance lets you guarantee that all classes grouped under a certain supertype have all the methods that the supertype has.*

In other words, you define a common protocol for a set of classes related through inheritance.

When you define methods in a superclass, that can be inherited by subclasses, you're announcing a kind of protocol to other code that says, "All my subtypes (i.e. subclasses) can do these things, with these methods that look like this..."

In other words, you establish a *contract*.

Class Animal establishes a common protocol for all Animal subtypes:

Animal
makeNoise()
eat()
sleep()
roam()

You're telling the world that *any* Animal can do these four things. That includes the method arguments and return types.

And remember, when we say *any Animal*, we mean Animal *and any class that extends from Animal*. Which again means, *any class that has Animal somewhere above it in the inheritance hierarchy*.

But we're not even at the really cool part yet, because we saved the best—*polymorphism*—for last.

When you define a supertype for a group of classes, *any subclass of that supertype can be substituted where the supertype is expected*.

Say, what?

Don't worry, we're nowhere near done explaining it. Two pages from now, you'll be an expert.

*When we say "all the methods" we mean "all the *inheritable* methods", which for now actually means, "all the *public* methods", although later we'll refine that definition a bit more.

And I care because...

Because you get to take advantage of polymorphism.

Which matters to me because...

Because you get to refer to a subclass object using a reference declared as the supertype.

And that means to me...

You get to write really flexible code. Code that's cleaner (more efficient, simpler). Code that's not just easier to develop, but also much, much easier to extend, in ways you never imagined at the time you originally wrote your code.

That means you can take that tropical vacation while your co-workers update the program, and your co-workers might not even need your source code.

You'll see how it works on the next page.

We don't know about you, but personally, we find the whole tropical vacation thing particularly motivating.



the way polymorphism works

To see how polymorphism works, we have to step back and look at the way we normally declare a reference and create an object...

The 3 steps of object declaration and assignment

Dog myDog = new Dog();

1 Declare a reference variable

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

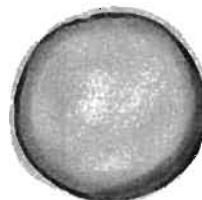
Tells the JVM to allocate space for a reference variable. The reference variable is, forever, of type Dog. In other words, a remote control that has buttons to control a Dog, but not a Cat or a Button or a Socket.



2 Create an object

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

Tells the JVM to allocate space for a new Dog object on the garbage collectible heap.

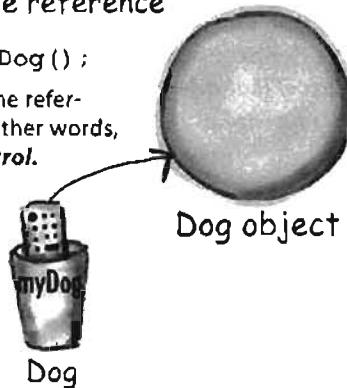


Dog object

3 Link the object and the reference

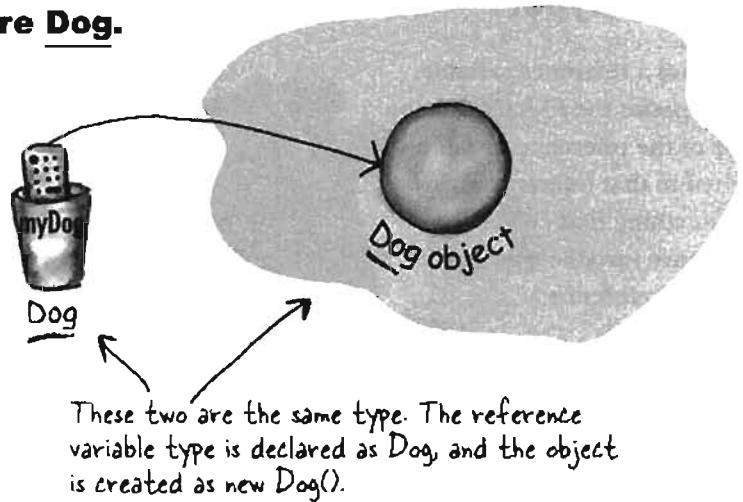
```
Dog myDog = new Dog();
```

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



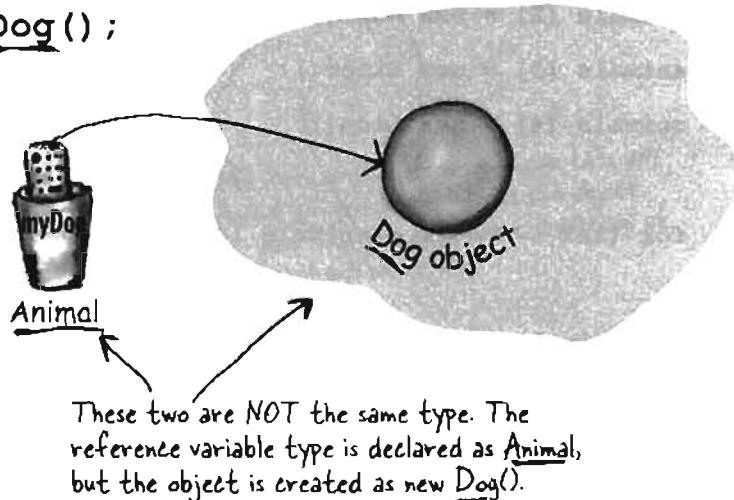
The important point is that the reference type AND the object type are the same.

In this example, both are Dog.



But with polymorphism, the reference and the object can be different.

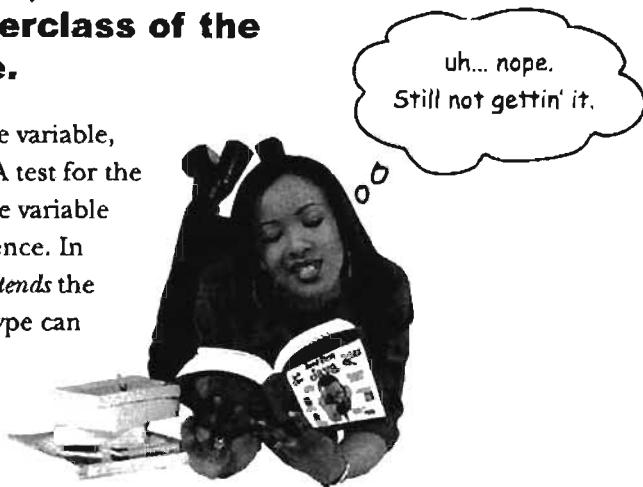
Animal myDog = new Dog();



polymorphism in action

With polymorphism, the reference type can be a superclass of the actual object type.

When you declare a reference variable, any object that passes the IS-A test for the declared type of the reference variable can be assigned to that reference. In other words, anything that *extends* the declared reference variable type can be *assigned* to the reference variable. *This lets you do things like make polymorphic arrays.*



OK, OK maybe an example will help.

```
Animal[] animals = new Animal[5];  
  
animals [0] = new Dog();  
animals [1] = new Cat();  
animals [2] = new Wolf();  
animals [3] = new Hippo();  
animals [4] = new Lion();  
  
for (int i = 0; i < animals.length; i++) {  
  
    animals[i].eat();  
    animals[i].roam();  
}
```

Declare an array of type Animal. In other words, an array that will hold objects of type Animal.

But look what you get to do... you can put ANY subclass of Animal in the Animal array!

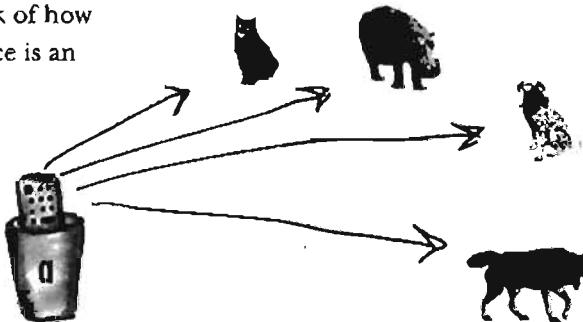
And here's the best polymorphic part (the raison d'être for the whole example), you get to loop through the array and call one of the Animal-class methods, and every object does the right thing!

When 'i' is 0, a Dog is at index 0 in the array, so you get the Dog's eat() method. When 'i' is 1, you get the Cat's eat() method
Same with roam().

But wait! There's more!

You can have polymorphic arguments and return types.

If you can declare a reference variable of a supertype, say, Animal, and assign a subclass object to it, say, Dog, think of how that might work when the reference is an argument to a method...



```
class Vet {
    public void giveShot(Animal a) {
        // do horrible things to the Animal at
        // the other end of the 'a' parameter
        a.makeNoise();
    }
}
```

The Animal parameter can take ANY Animal type as the argument. And when the Vet is done giving the shot, it tells the Animal to makeNoise(), and whatever Animal is really out there on the heap, that's whose makeNoise() method will run.

```
class PetOwner {
    public void start() {
        Vet v = new Vet();
        Dog d = new Dog();
        Hippo h = new Hippo();
        v.giveShot(d);
        v.giveShot(h);
    }
}
```

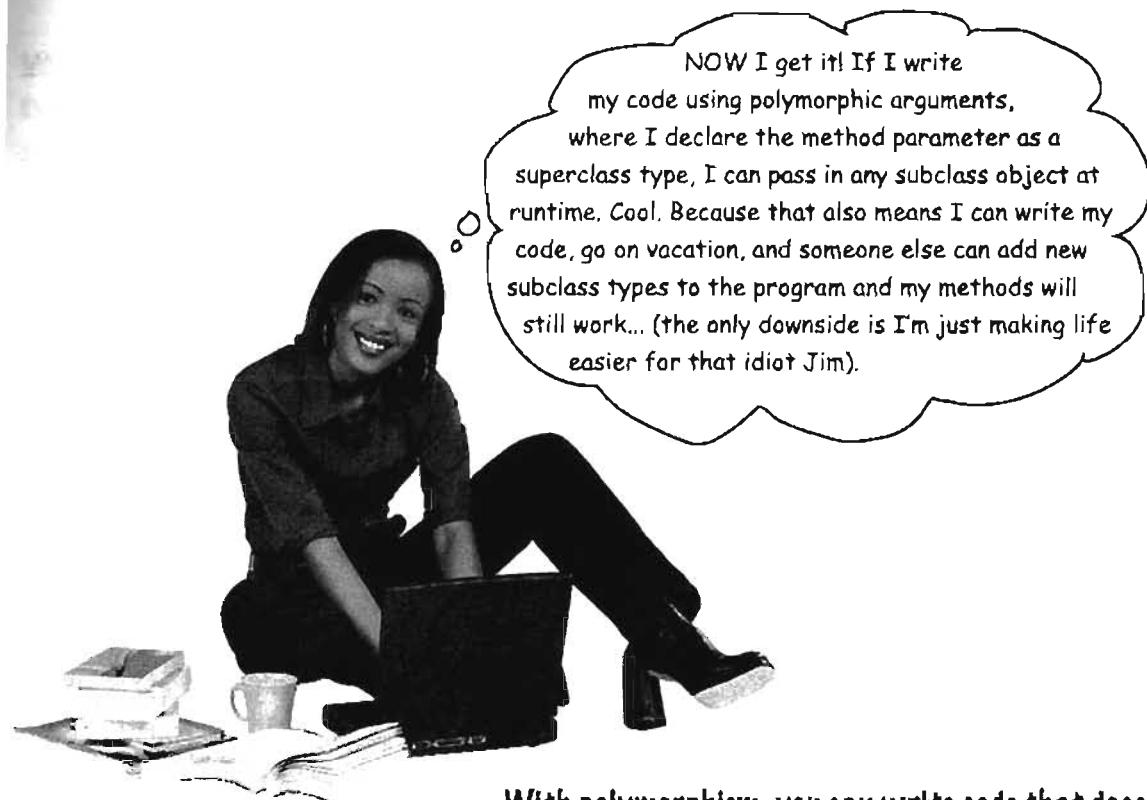
The Vet's giveShot() method can take any Animal you give it. As long as the object you pass in as the argument is a subclass of Animal, it will work.



Dog's makeNoise() runs

Hippo's makeNoise() runs

exploiting the power of polymorphism



With polymorphism, you can write code that doesn't have to change when you introduce new subclass types into the program.

Remember that Vet class? If you write that Vet class using arguments declared as type *Animal*, your code can handle any *Animal subclass*. That means if others want to take advantage of your Vet class, all they have to do is make sure *their* new Animal types extend class *Animal*. The Vet methods will still work, even though the Vet class was written without any knowledge of the new Animal subtypes the Vet will be working on.

BRAIN POWER

Why is polymorphism guaranteed to work this way? Why is it always safe to assume that any *subclass* type will have the methods you think you're calling on the *superclass* type (the *superclass reference type* you're using the dot operator on)?

there are no
Dumb Questions

Q: Are there any practical limits on the levels of subclassing? How deep can you go?

A: If you look in the Java API, you'll see that most inheritance hierarchies are wide but not deep. Most are no more than one or two levels deep, although there are exceptions (especially in the GUI classes). You'll come to realize that it usually makes more sense to keep your inheritance trees shallow, but there isn't a hard limit (well, not one that you'd ever run into).

Q: Hey, I just thought of something... if you don't have access to the source code for a class, but you want to change the way a method of that class works, could you use subclassing to do that? To extend the "bad" class and override the method with your own better code?

A: Yep. That's one cool feature of OO, and sometimes it saves you from having to rewrite the class from scratch, or track down the programmer who hid the source code.

Q: Can you extend *any* class? Or is it like class members where if the class is private you can't inherit it...

A: There's no such thing as a private class, except in a very special case called an *inner* class, that we haven't looked at yet. But there are three things that can prevent a class from being subclassed.

The first is access control. Even though a class *can't* be marked `private`, a class *can* be non-public (what you get if you don't declare the class as `public`). A non-public class can be subclassed only by classes in the same package as the class. Classes in a different package won't be able to subclass (or even *use*, for that matter) the non-public class.

The second thing that stops a class from being subclassed is the keyword modifier `final`. A final class means that it's the end of the inheritance line. Nobody, ever, can extend a final class.

The third issue is that if a class has only private constructors (we'll look at constructors in chapter 9), it can't be subclassed.

Q: Why would you ever want to make a final class? What advantage would there be in preventing a class from being subclassed?

A: Typically, you won't make your classes final. But if you need security — the security of knowing that the methods will always work the way that you wrote them (because they can't be overridden), a final class will give you that. A lot of classes in the Java API are final for that reason. The `String` class, for example, is final because, well, imagine the havoc if somebody came along and changed the way `Strings` behave!

Q: Can you make a *method* final, without making the whole *class* final?

A: If you want to protect a specific method from being overridden, mark the *method* with the `final` modifier. Mark the whole *class* as `final` if you want to guarantee that *none* of the methods in that class will ever be overridden.

Keeping the contract: rules for overriding

When you override a method from a superclass, you're agreeing to fulfill the contract. The contract that says, for example, "I take no arguments and I return a boolean." In other words, the arguments and return types of your overriding method must look to the outside world *exactly* like the overridden method in the superclass.

The methods *are* the contract.

If polymorphism is going to work, the Toaster's version of the overridden method from Appliance has to work at runtime.

Remember, the compiler looks at the reference type to decide whether you can call a particular method on that reference. With an Appliance reference to a Toaster, the compiler cares only if class *Appliance* has the method you're invoking on an Appliance reference. But at runtime, the JVM looks not at the *reference* type (*Appliance*) but at the actual *Toaster* object on the heap. So if the compiler has already *approved* the method call, the only way it can work is if the overriding method has the same arguments and return types. Otherwise, someone with an Appliance reference will call *turnOn()* as a no-arg method, even though there's a version in *Toaster* that takes an *int*. Which one is called at runtime? The one in *Appliance*. In other words, *the turnOn(int level) method in Toaster is not an override!*

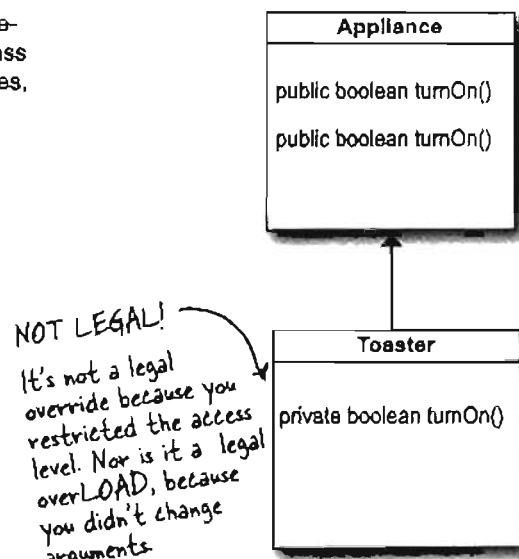
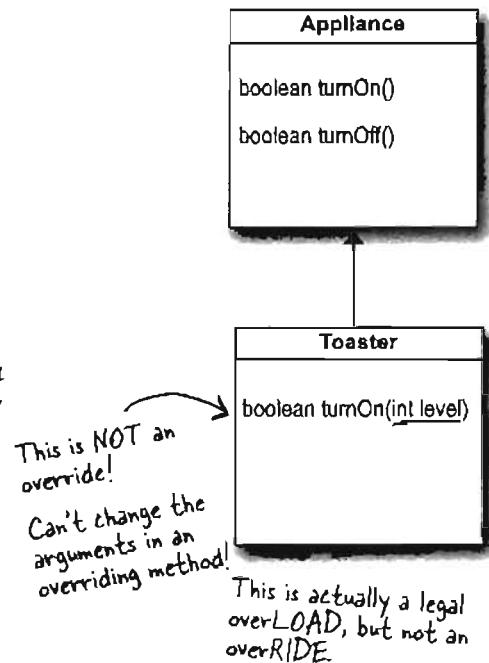
Arguments must be the same, and return types must be compatible.

The contract of superclass defines how other code can use a method. Whatever the superclass takes as an argument, the subclass overriding the method must use that same argument. And whatever the superclass declares as a return type, the overriding method must declare either the same type, or a subclass type. Remember, a subclass object is guaranteed to be able to do anything its superclass declares, so it's safe to return a subclass where the superclass is expected.

The method can't be less accessible.

That means the access level must be the same, or friendlier. That means you can't, for example, override a public method and make it private. What a shock that would be to the code invoking what it thinks (at compile time) is a public method, if suddenly at runtime the JVM slammed the door shut because the overriding version called at runtime is private!

So far we've learned about two access levels: private and public. The other two are in the deployment chapter (Release your Code) and appendix B. There's also another rule about overriding related to exception handling, but we'll wait until the chapter on exceptions (Risky Behavior) to cover that.



Overloading a method

Method overloading is nothing more than having two methods with the same name but different argument lists. Period. There's no polymorphism involved with overloaded methods!

Overloading lets you make multiple versions of a method, with different argument lists, for convenience to the callers. For example, if you have a method that takes only an int, the calling code has to convert, say, a double into an int before calling your method. But if you overloaded the method with another version that takes a double, then you've made things easier for the caller. You'll see more of this when we look into constructors in the object lifecycle chapter.

Since an overloading method isn't trying to fulfill the polymorphism contract defined by its superclass, overloaded methods have much more flexibility.

● **The return types can be different.**

You're free to change the return types in overloaded methods, as long as the argument lists are different.

● **You can't change ONLY the return type.**

If only the return type is different, it's not a valid overload—the compiler will assume you're trying to override the method. And even that won't be legal unless the return type is a subtype of the return type declared in the superclass. To overload a method, you MUST change the argument list, although you can change the return type to anything.

● **You can vary the access levels in any direction.**

You're free to overload a method with a method that's more restrictive. It doesn't matter, since the new method isn't obligated to fulfill the contract of the overloaded method.

An overloaded method is just a different method that happens to have the same method name. It has nothing to do with inheritance and polymorphism. An overloaded method is NOT the same as an overridden method.

Legal examples of method overloading:

```
public class Overloads {
    String uniqueID;

    public int addNums(int a, int b) {
        return a + b;
    }

    public double addNums(double a, double b) {
        return a + b;
    }

    public void setUniqueID(String theID) {
        // lots of validation code, and then:
        uniqueID = theID;
    }

    public void setUniqueID(int ssNumber) {
        String numString = "" + ssNumber;
        setUniqueID(numString);
    }
}
```

exercise: Mixed Messages



Exercise

Mixed Messages

a = 6; → 56
b = 5; → 11
a = 5; → 65

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

the program:

```
class A {  
    int ivar = 7;  
    void m1() {  
        System.out.print("A's m1, ");  
    }  
    void m2() {  
        System.out.print("A's m2, ");  
    }  
    void m3() {  
        System.out.print("A's m3, ");  
    }  
}  
  
class B extends A {  
    void m1() {  
        System.out.print("B's m1, ");  
    }  
}
```

```
class C extends B {  
    void m3() {  
        System.out.print("C's m3, "+(ivar + 6));  
    }  
}  
  
public class Mixed2 {  
    public static void main(String [] args) {  
        A a = new A();  
        B b = new B();  
        C c = new C();  
        A a2 = new C();  
    }  
}
```

candidate code
goes here
(three lines)

code

candidates:

b.m1(); }
c.m2(); }
a.m3(); }

c.m1(); }
c.m2(); }
c.m3(); }

a.m1(); }
b.m2(); }
c.m3(); }

a2.m1(); }
a2.m2(); }
a2.m3(); }

output:

A's m1, A's m2, C's m3, 6

B's m1, A's m2, A's m3,

A's m1, B's m2, A's m3,

B's m1, A's m2, C's m3, 13

B's m1, C's m2, A's m3,

B's m1, A's m2, C's m3, 6

A's m1, A's m2, C's m3, 13



BE the Compiler

Which of the A-B pairs of methods listed on the right, if inserted into the classes on the left, would compile and produce the output shown? (The A method inserted into class Monster, the B method inserted into class Vampire.)

```
public class MonsterTestDrive {
    public static void main(String [] args) {
        Monster [] ma = new Monster[3];
        ma[0] = new Vampire();
        ma[1] = new Dragon();
        ma[2] = new Monster();
        for(int x = 0; x < 3; x++) {
            ma[x].frighten(x);
        }
    }
}
```

```
class Monster {
    A
}
```

```
class Vampire extends Monster {
    B
}
```

```
class Dragon extends Monster {
    boolean frighten(int degree) {
        System.out.println("breath fire");
        return true;
    }
}
```

```
File Edit Window Help Save Yourself
$ java MonsterTestDrive
a bite?
breath fire
arrrgh
```

- 1 **A** boolean frighten(int d) {
 System.out.println("arrrgh");
 return true;
 }
 B boolean frighten(int x) {
 System.out.println("a bite?");
 return false;
 }

- 2 **A** boolean frighten(int x) {
 System.out.println("arrrgh");
 return true;
 }
 B int frighten(int f) {
 System.out.println("a bite?");
 return 1;
 }

- 3 **A** boolean frighten(int x) {
 System.out.println("arrrgh");
 return false;
 }
 B boolean scare(int x) {
 System.out.println("a bite?");
 return true;
 }

- 4 **A** boolean frighten(int z) {
 System.out.println("arrrgh");
 return true;
 }
 B boolean frighten(byte b) {
 System.out.println("a bite?");
 return true;
 }

puzzle: Pool Puzzle



Pool Puzzle

Your **Job** is to take code snippets from the pool and place them into the blank lines in the code. You may use the same snippet more than once, and you might not need to use all the snippets. Your **goal** is to make a set of classes that will compile and run together as a program. Don't be fooled – this one's harder than it looks.

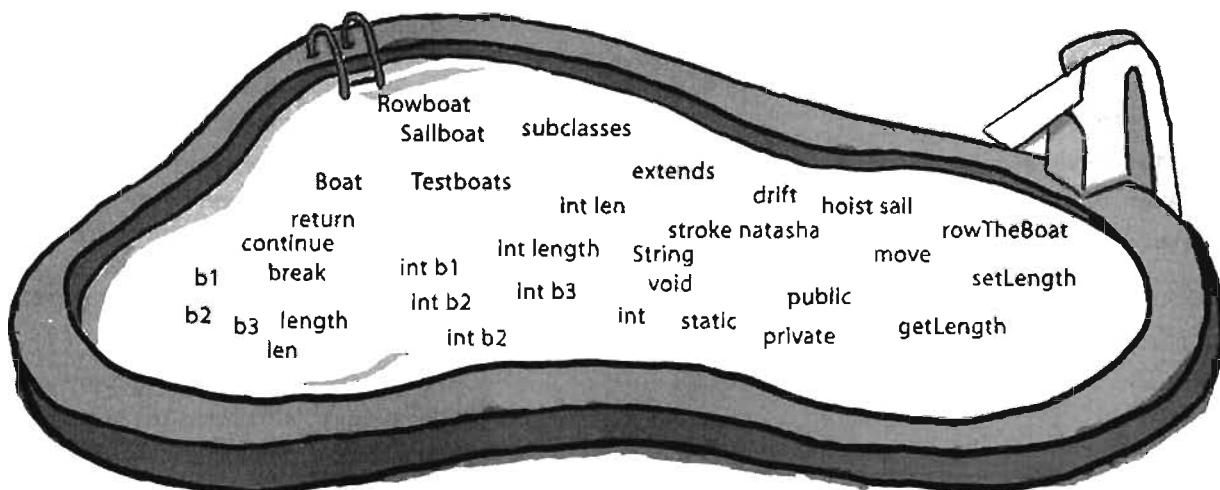
```
public class Rowboat _____ {
    public _____ rowTheBoat() {
        System.out.print("stroke natasha");
    }
}

public class _____ {
    private int _____;
    _____ void _____(_____) {
        length = len;
    }
    public int getLength() {
        _____;
    }
    public _____ move() {
        System.out.print("_____");
    }
}
```

```
public class TestBoats {
    _____ main(String[] args) {
        _____ b1 = new Boat();
        Sailboat b2 = new _____();
        Rowboat _____ = new Rowboat();
        b2.setLength(32);
        b1._____();
        b3._____();
        _____ .move();
    }
}

public class _____ Boat {
    public _____() {
        System.out.print("_____");
    }
}
```

OUTPUT: drift drift hoist sail





Exercise Solutions

BE the Compiler

Set 1 will work.

Set 2 will not compile because of Vampire's return type (int).

The Vampire's frighten() method (B) is not a legal override OR overload of Monster's frighten() method. Changing ONLY the return type is not enough to make a valid overload, and since an int is not compatible with a boolean, the method is not a valid override. (Remember, if you change ONLY the return type, it must be to a return type that is compatible with the superclass version's return type, and then it's an override.)

Sets 3 and 4 will compile, but produce:

arrrgh

breath fire

arrrgh

Remember, class Vampire did not *override* class Monster's frighten() method. (The frighten() method in Vampire's set 4 takes a byte, not an int.)

**code
candidates:**

```
b.m1();
c.m2();
a.m3();
```

output:

A's m1, A's m2, C's m3, 6

**Mixed
Messages**

```
c.m1();
c.m2();
c.m3();
```

B's m1, A's m2, A's m3,

```
a.m1();
b.m2();
c.m3();
```

A's m1, B's m2, A's m3,

```
a2.m1();
a2.m2();
a2.m3();
```

B's m1, C's m2, A's m3,

B's m1, A's m2, C's m3, 6

A's m1, A's m2, C's m3, 13

puzzle answers



```
public class Rowboat extends Boat {
    public void rowTheBoat() {
        System.out.print("stroke natasha");
    }
}
public class Boat {
    private int length ;
    public void setLength ( int len ) {
        length = len;
    }
    public int getLength() {
        return length ;
    }
    public void move() {
        System.out.print("drift ");
    }
}
```

```
public class TestBoats {
    public static void main(String[] args){
        Boat b1 = new Boat();
        Sailboat b2 = new Sailboat();
        Rowboat b3 = new Rowboat();
        b2.setLength(32);
        b1.move();
        b3.move();
        b2.move();
    }
}
public class Sailboat extends Boat {
    public void move() {
        System.out.print("hoist sail ");
    }
}
```

OUTPUT: drift drift hoist sail

8 interfaces and abstract classes

Serious Polymorphism

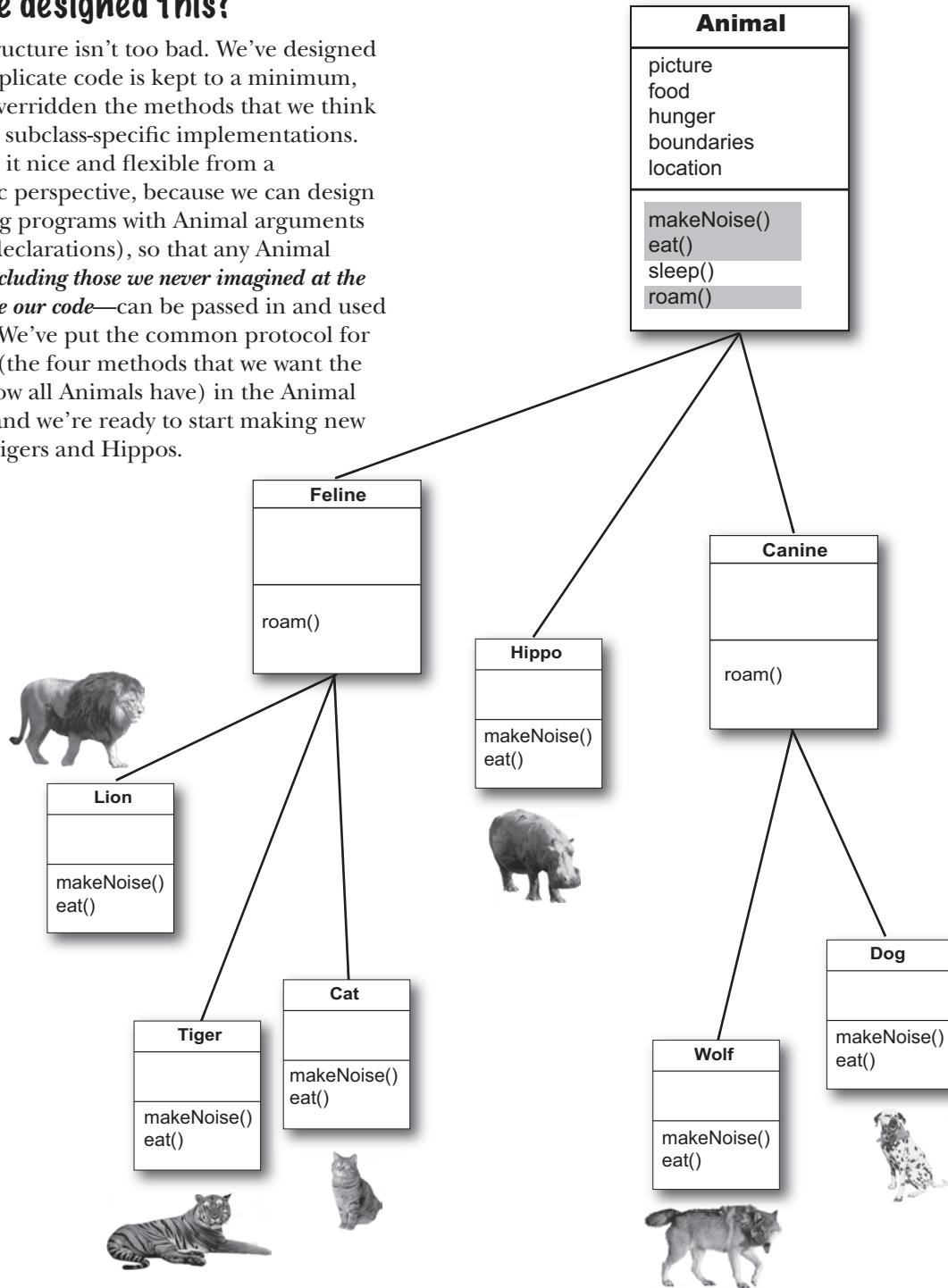


Inheritance is just the beginning. To exploit polymorphism, we need interfaces (and not the GUI kind). We need to go beyond simple inheritance to a level of flexibility and extensibility you can get only by designing and coding to interface specifications. Some of the coolest parts of Java wouldn't even be possible without interfaces, so even if you don't design with them yourself, you still have to use them. But you'll *want* to design with them. You'll *need* to design with them. **You'll wonder how you ever lived without them.** What's an interface? It's a 100% abstract class. What's an abstract class? It's a class that can't be instantiated. What's that good for? You'll see in just a few moments. But if you think about the end of the last chapter, and how we used polymorphic arguments so that a single Vet method could take Animal subclasses of all types, well, that was just scratching the surface. Interfaces are the **poly** in polymorphism. The **ab** in abstract. The **caffeine** in Java.

designing with inheritance

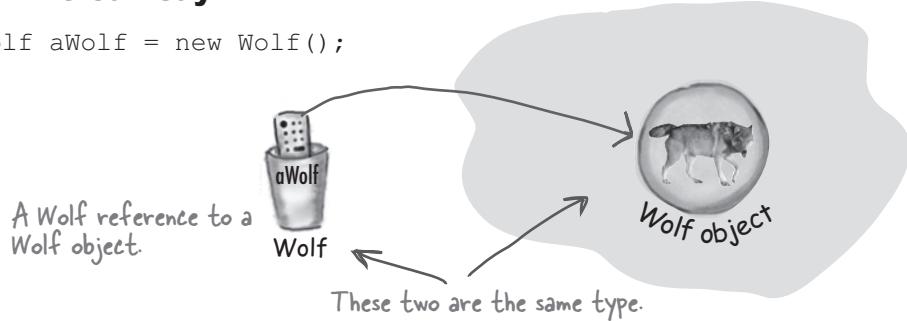
Did we forget about something when we designed this?

The class structure isn't too bad. We've designed it so that duplicate code is kept to a minimum, and we've overridden the methods that we think should have subclass-specific implementations. We've made it nice and flexible from a polymorphic perspective, because we can design Animal-using programs with Animal arguments (and array declarations), so that any Animal subtype—*including those we never imagined at the time we wrote our code*—can be passed in and used at runtime. We've put the common protocol for all Animals (the four methods that we want the world to know all Animals have) in the Animal superclass, and we're ready to start making new Lions and Tigers and Hippos.

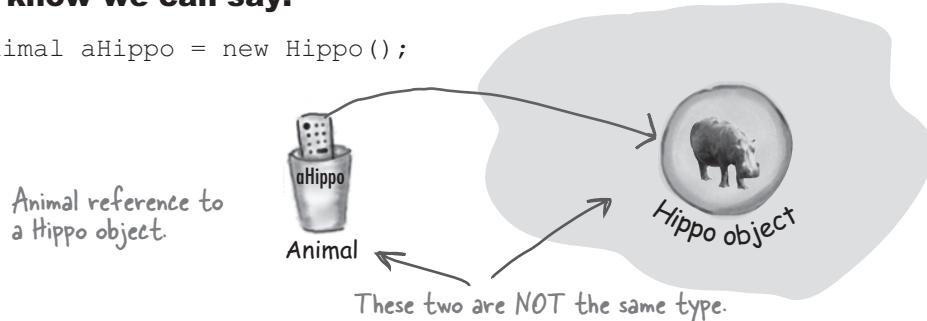


We know we can say:

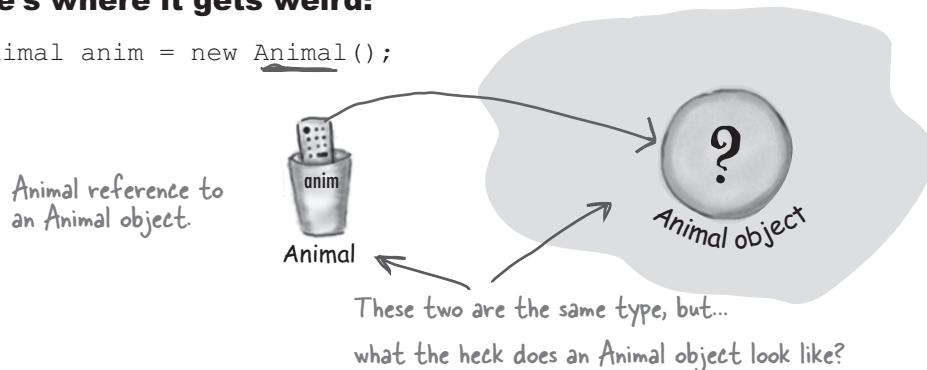
```
Wolf aWolf = new Wolf();
```

**And we know we can say:**

```
Animal aHippo = new Hippo();
```

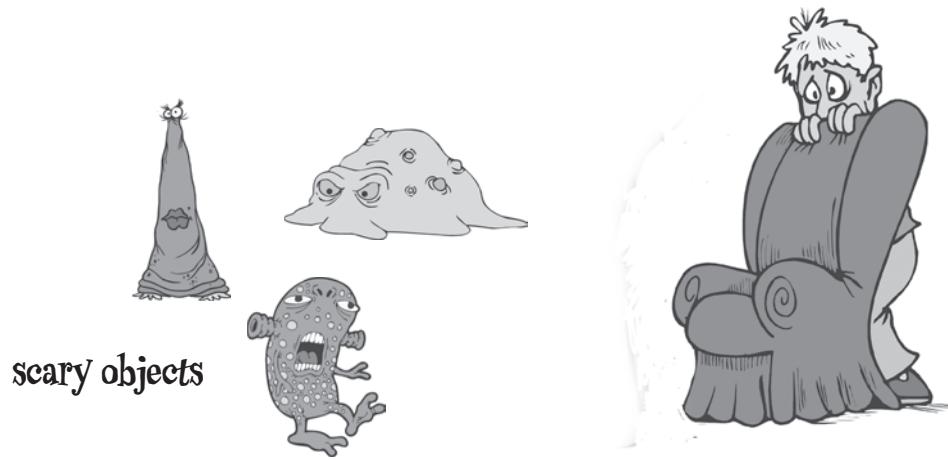
**But here's where it gets weird:**

```
Animal anim = new Animal();
```



when objects go bad

What does a new Animal() object look like?



What are the instance variable values?

Some classes just should not be instantiated!

It makes sense to create a Wolf object or a Hippo object or a Tiger object, but what exactly *is* an Animal object? What shape is it? What color, size, number of legs...

Trying to create an object of type Animal is like a **nightmare Star Trek™ transporter accident**. The one where somewhere in the beam-me-up process something bad happened to the buffer.

But how do we deal with this? We *need* an Animal class, for inheritance and polymorphism. But we want programmers to instantiate only the less abstract *subclasses* of class Animal, not Animal itself. We want Tiger objects and Lion objects, **not Animal objects**.

Fortunately, there's a simple way to prevent a class from ever being instantiated. In other words, to stop anyone from saying “**new**” on that type. By marking the class as **abstract**, the compiler will stop any code, anywhere, from ever creating an instance of that type.

You can still use that abstract type as a reference type. In fact, that's a big part of why you have that abstract class in the first place (to use it as a polymorphic argument or return type, or to make a polymorphic array).

When you're designing your class inheritance structure, you have to decide which classes are *abstract* and which are *concrete*. Concrete classes are those that are specific enough to be instantiated. A *concrete* class just means that it's OK to make objects of that type.

Making a class abstract is easy—put the keyword **abstract** before the class declaration:

```
abstract class Canine extends Animal {  
    public void roam() { }  
}
```

The compiler won't let you instantiate an abstract class

An abstract class means that nobody can ever make a new instance of that class. You can still use that abstract class as a declared reference type, for the purpose of polymorphism, but you don't have to worry about somebody making objects of that type. The compiler *guarantees* it.

```
abstract public class Canine extends Animal
{
    public void roam() { }

    public class MakeCanine {
        public void go() {
            Canine c; } // This is OK, because you can always assign
            // a subclass object to a superclass reference,
            // even if the superclass is abstract.

            c = new Dog();
            c = new Canine(); // ← class Canine is marked abstract,
                            // so the compiler will NOT let you do this.

        }
    }
}
```

```
File Edit Window Help BeamMeUp
% javac MakeCanine.java
MakeCanine.java:5: Canine is abstract;
cannot be instantiated
    c = new Canine();
               ^
1 error
```

An **abstract class** has virtually* no use, no value, no purpose in life, unless it is **extended**.

With an abstract class, the guys doing the work at runtime are **instances of a subclass** of your abstract class.

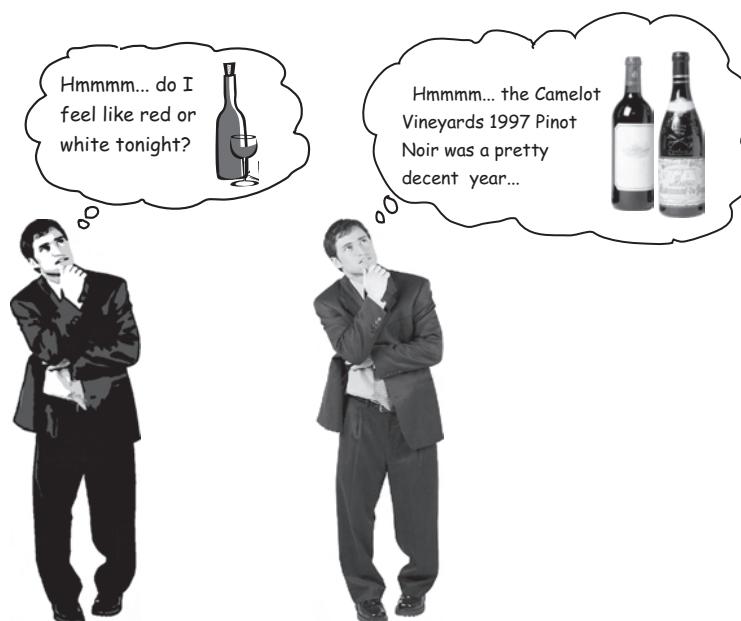
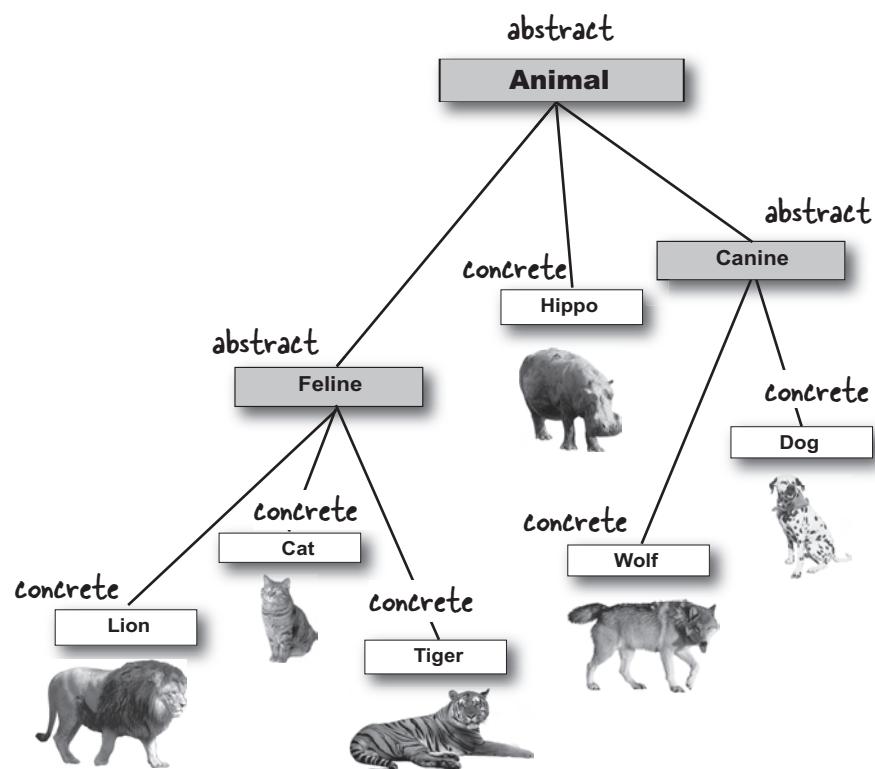
*There is an exception to this—an abstract class can have static members (see chapter 10).

abstract and concrete classes

Abstract vs. Concrete

A class that's not abstract is called a *concrete* class. In the Animal inheritance tree, if we make Animal, Canine, and Feline abstract, that leaves Hippo, Wolf, Dog, Tiger, Lion, and Cat as the concrete subclasses.

Flip through the Java API and you'll find a lot of abstract classes, especially in the GUI library. What does a GUI Component look like? The Component class is the superclass of GUI-related classes for things like buttons, text areas, scrollbars, dialog boxes, you name it. You don't make an instance of a generic *Component* and put it on the screen, you make a JButton. In other words, you instantiate only a *concrete subclass* of Component, but never Component itself.



abstract or concrete?

How do you know when a class should be abstract? **Wine** is probably abstract. But what about **Red** and **White**? Again probably abstract (for some of us, anyway). But at what point in the hierarchy do things become concrete?

Do you make **PinotNoir** concrete, or is it abstract too? It looks like the Camelot Vineyards 1997 Pinot Noir is probably concrete no matter what. But how do you know for sure?

Look at the Animal inheritance tree above. Do the choices we've made for which classes are abstract and which are concrete seem appropriate? Would you change anything about the Animal inheritance tree (other than adding more Animals, of course)?

Abstract methods

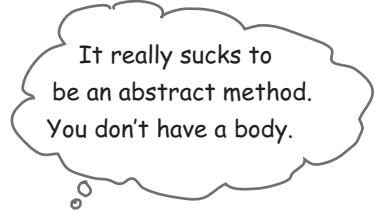
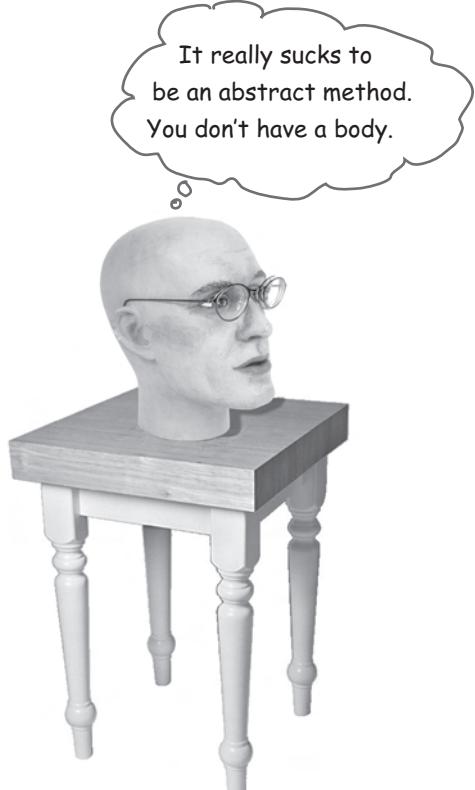
Besides classes, you can mark *methods* abstract, too. An abstract class means the class must be *extended*; an abstract method means the method must be *overridden*. You might decide that some (or all) behaviors in an abstract class don't make any sense unless they're implemented by a more specific subclass. In other words, you can't think of any generic method implementation that could possibly be useful for subclasses. What would a generic *eat()* method look like?

An abstract method has no body!

Because you've already decided there isn't any code that would make sense in the abstract method, you won't put in a method body. So no curly braces—just end the declaration with a semicolon.

```
public abstract void eat();
```

No method body!
End it with a semicolon.

If you declare an abstract method, you MUST mark the class abstract as well. You can't have an abstract method in a non-abstract class.

If you put even a single abstract method in a class, you have to make the class abstract. But you *can* mix both abstract and non-abstract methods in the abstract class.

there are no
Dumb Questions

Q: What is the *point* of an abstract method? I thought the whole point of an abstract class was to have common code that could be inherited by subclasses.

A: Inheritable method implementations (in other words, methods with actual *bodies*) are A Good Thing to put in a superclass. *When it makes sense*. And in an abstract class, it often *doesn't* make sense, because you can't come up with any generic code that subclasses would find useful. The point of an abstract method is that even though you haven't put in any actual method code, you've still defined part of the *protocol* for a group of subtypes (subclasses).

Q: Which is good because...

A: Polymorphism! Remember, what you want is the ability to use a superclass type (often abstract) as a method argument, return type, or array type. That way, you get to add new subtypes (like a new Animal subclass) to your program without having to rewrite (or add) new methods to deal with those new types. Imagine how you'd have to change the Vet class, if it didn't use Animal as its argument type for methods. You'd have to have a separate method for every single Animal subclass! One that takes a Lion, one that takes a Wolf, one that takes a... you get the idea. So with an abstract method, you're saying, "All subtypes of this type have THIS method." for the benefit of polymorphism.

you must implement abstract methods

You MUST implement all abstract methods



Abstract methods don't have a body; they exist solely for polymorphism. That means the first concrete class in the inheritance tree must implement *all* abstract methods.

You can, however, pass the buck by being abstract yourself. If both Animal and Canine are abstract, for example, and both have abstract methods, class Canine does not have to implement the abstract methods from Animal. But as soon as we get to the first concrete subclass, like Dog, that subclass must implement *all* of the abstract methods from both Animal and Canine.

But remember that an abstract class can have both abstract and *non-abstract* methods, so Canine, for example, could implement an abstract method from Animal, so that Dog didn't have to. But if Canine says nothing about the abstract methods from Animal, Dog has to implement all of Animal's abstract methods.

When we say "you must implement the abstract method", that means you *must provide a body*. That means you must create a non-abstract method in your class with the same method signature (name and arguments) and a return type that is compatible with the declared return type of the abstract method. What you put *in* that method is up to you. All Java cares about is that the method is *there*, in your concrete subclass.



Abstract vs. Concrete Classes

Let's put all this abstract rhetoric into some concrete use. In the middle column we've listed some classes. Your job is to imagine applications where the listed class might be concrete, and applications where the listed class might be abstract. We took a shot at the first few to get you going. For example, class Tree would be abstract in a tree nursery program, where differences between an Oak and an Aspen matter. But in a golf simulation program, Tree might be a concrete class (perhaps a subclass of Obstacle), because the program doesn't care about or distinguish between different types of trees. (There's no one right answer; it depends on your design.)

Concrete

golf course simulation

satellite photo application

Sample class

Tree

House

Town

Football Player

Chair

Customer

Sales Order

Book

Store

Supplier

Golf Club

Carburetor

Oven

Abstract

tree nursery application

architect application

coaching application

polymorphism examples

Polymorphism in action

Let's say that we want to write our *own* kind of list class, one that will hold Dog objects, but pretend for a moment that we don't know about the ArrayList class. For the first pass, we'll give it just an `add()` method. We'll use a simple Dog array (`Dog []`) to keep the added Dog objects, and give it a length of 5. When we reach the limit of 5 Dog objects, you can still call the `add()` method but it won't do anything. If we're *not* at the limit, the `add()` method puts the Dog in the array at the next available index position, then increments that next available index (`nextIndex`).

Building our own Dog-specific list

(Perhaps the world's worst attempt at making our own ArrayList kind of class, from scratch.)

version
1

MyDogList
Dog[] dogs int nextIndex
add(Dog d)

```
public class MyDogList {
    private Dog [] dogs = new Dog[5]; Use a plain old Dog array behind the scenes.
    private int nextIndex = 0; We'll increment this each time a new Dog is added.
    public void add(Dog d) {
        if (nextIndex < dogs.length) { If we're not already at the limit of the dogs array, add the Dog and print a message.
            dogs[nextIndex] = d;
            System.out.println("Dog added at " + nextIndex);
            nextIndex++; increment, to give us the next index to use
        }
    }
}
```

Uh-oh, now we need to keep Cats, too.

We have a few options here:

- 1) Make a separate class, MyCatList, to hold Cat objects. Pretty clunky.
- 2) Make a single class, DogAndCatList, that keeps two different arrays as instance variables and has two different add() methods: addCat(Cat c) and addDog(Dog d). Another clunky solution.
- 3) Make heterogeneous AnimalList class, that takes *any* kind of Animal subclass (since we know that if the spec changed to add Cats, sooner or later we'll have some *other* kind of animal added as well). We like this option best, so let's change our class to make it more generic, to take Animals instead of just Dogs. We've highlighted the key changes (the logic is the same, of course, but the type has changed from Dog to Animal everywhere in the code).

Building our own Animal-specific list

```
version (2)
MyAnimalList
Animal[] animals
int nextIndex
add(Animal a)

public class MyAnimalList {
    private Animal[] animals = new Animal[5];
    private int nextIndex = 0;

    public void add(Animal a) {
        if (nextIndex < animals.length) {
            animals[nextIndex] = a;
            System.out.println("Animal added at " + nextIndex);
            nextIndex++;
        }
    }
}
```

Don't panic. We're not making a new Animal object; we're making a new array object, of type Animal. (Remember, you cannot make a new instance of an abstract type, but you CAN make an array object declared to HOLD that type.)

```
public class AnimalTestDrive{
    public static void main (String[] args) {
        MyAnimalList list = new MyAnimalList();
        Dog a = new Dog();
        Cat c = new Cat();
        list.add(a);
        list.add(c);
    }
}
```

```
File Edit Window Help Harm
% java AnimalTestDrive
Animal added at 0
Animal added at 1
```

the ultimate superclass: `Object`

What about non-Animals? Why not make a class generic enough to take anything?

You know where this is heading. We want to change the type of the array, along with the `add()` method argument, to something *above* Animal. Something even *more* generic, *more* abstract than Animal. But how can we do it? We don't *have* a superclass for Animal.

Then again, maybe we do...

Remember those methods of `ArrayList`? Look how the `remove`, `contains`, and `indexOf` method all use an object of type... `Object`!

Every class in Java extends class `Object`.

Class `Object` is the mother of all classes; it's the superclass of *everything*.

Even if you take advantage of polymorphism, you still have to create a class with methods that take and return *your* polymorphic type. Without a common superclass for everything in Java, there'd be no way for the developers of Java to create classes with methods that could take *your* custom types... *types they never knew about when they wrote the `ArrayList` class*.

So you were making subclasses of class `Object` from the very beginning and you didn't even know it. *Every class you write extends Object*, without your ever having to say it. But you can think of it as though a class you write looks like this:

```
public class Dog extends Object { }
```

But wait a minute, Dog *already* extends something, `Canine`. That's OK. The compiler will make `Canine` extend `Object` instead. Except `Canine` extends `Animal`. No problem, then the compiler will just make `Animal` extend `Object`.

Any class that doesn't explicitly extend another class, implicitly extends `Object`.

So, since Dog extends Canine, it doesn't *directly* extend `Object` (although it does extend it indirectly), and the same is true for Canine, but `Animal` *does* directly extend `Object`.

version
3

(These are just a few of the methods in `ArrayList`...there are many more.)

ArrayList

boolean remove(Object elem)
Removes the object at the index parameter. Returns 'true' if the element was in the list.

boolean contains(Object elem)
Returns 'true' if there's a match for the object parameter.

boolean isEmpty()
Returns 'true' if the list has no elements.

int indexOf(Object elem)
Returns either the index of the object parameter, or -1.

Object get(int index)
Returns the element at this position in the list.

boolean add(Object elem)
Adds the element to the list (returns 'true').

// more

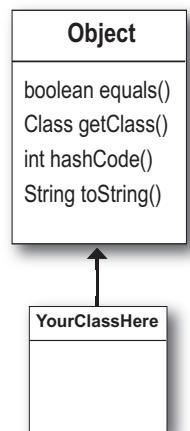
Many of the `ArrayList` methods use the ultimate polymorphic type, `Object`. Since every class in Java is a subclass of `Object`, these `ArrayList` methods can take anything!

(Note: as of Java 5.0, the `get()` and `add()` methods actually look a little different than the ones shown here, but for now this is the way to think about it. We'll get into the full story a little later.)

So what's in this ultra-super-megaclass Object?

If you were Java, what behavior would you want *every* object to have? Hmm... let's see... how about a method that lets you find out if one object is equal to another object? What about a method that can tell you the actual class type of that object? Maybe a method that gives you a hashcode for the object, so you can use the object in hashtables (we'll talk about Java's hashtables in chapter 17 and appendix B). Oh, here's a good one—a method that prints out a String message for that object.

And what do you know? As if by magic, class Object does indeed have methods for those four things. That's not all, though, but these are the ones we really care about.



Just SOME of the methods of class Object.

Every class you write inherits all the methods of class Object. The classes you've written inherited methods you didn't even know you had.

① equals(Object o)

```

Dog a = new Dog();
Cat c = new Cat();

if (a.equals(c)) {
    System.out.println("true");
} else {
    System.out.println("false");
}
  
```

```

File Edit Window Help Stop
% java TestObject
false
  
```

Tells you if two objects are considered 'equal' (we'll talk about what 'equal' really means in appendix B).

③ hashCode()

```

Cat c = new Cat();
System.out.println(c.hashCode());
  
```

```

File Edit Window Help Drop
% java TestObject
8202111
  
```

Prints out a hashcode for the object (for now, think of it as a unique ID).

② getClass()

```

Cat c = new Cat();
System.out.println(c.getClass());
  
```

```

File Edit Window Help Faint
% java TestObject
class Cat
  
```

Gives you back the class that object was instantiated from.

④ toString()

```

Cat c = new Cat();
System.out.println(c.toString());
  
```

```

File Edit Window Help LapseIntoComa
% java TestObject
Cat@7d277f
  
```

Prints out a String message with the name of the class and some other number we rarely care about.

Object and abstract classes

^{there are no}
Dumb Questions

Q: Is class Object abstract?

A: No. Well, not in the formal Java sense anyway. Object is a non-abstract class because it's got method implementation code that all classes can inherit and use out-of-the-box, without having to override the methods.

Q: Then *can* you override the methods in Object?

A: Some of them. But some of them are marked `final`, which means you can't override them. You're encouraged (strongly) to override `hashCode()`, `equals()`, and `toString()` in your own classes, and you'll learn how to do that a little later in the book. But some of the methods, like `getClass()`, do things that must work in a specific, guaranteed way.

Q: If `ArrayList` methods are generic enough to use Object, then what does it mean to say `ArrayList<DotCom>`? I thought I was restricting the `ArrayList` to hold only DotCom objects?

A: You were restricting it. Prior to Java 5.0, `ArrayLists` couldn't be restricted. They were all essentially what you get in Java 5.0 today if you write `ArrayList<Object>`. In other words, ***an ArrayList restricted to anything that's an Object***, which means *any* object in Java, instantiated from *any* class type! We'll cover the details of this new `<type>` syntax later in the book.

Q: OK, back to class Object being non-abstract (so I guess that means it's concrete), HOW can you let somebody make an Object object? Isn't that just as weird as making an Animal object?

A: Good question! Why is it acceptable to make a new Object instance? Because sometimes you just want a generic object to use as, well, as an object. A *lightweight* object. By far, the most common use of an instance of type Object is for thread synchronization (which you'll learn about in chapter 15). For now, just stick that on the back burner and assume that you will rarely make objects of type Object, even though you *can*.

Q: So is it fair to say that the main purpose for type Object is so that you can use it for a polymorphic argument and return type? Like in `ArrayList`?

A: The Object class serves two main purposes: to act as a polymorphic type for methods that need to work on any class that you or anyone else makes, and to provide *real* method code that all objects in Java need at runtime (and putting them in class Object means all other classes inherit them). Some of the most important methods in Object are related to threads, and we'll see those later in the book.

Q: If it's so good to use polymorphic types, why don't you just make ALL your methods take and return type Object?

A: Ahhhh... think about what would happen. For one thing, you would defeat the whole point of 'type-safety', one of Java's greatest protection mechanisms for your code. With type-safety, Java guarantees that you won't ask the wrong object to do something you *meant* to ask of another object type. Like, ask a *Ferrari* (which you think is a *Toaster*) to *cook itself*. But the truth is, you *don't* have to worry about that fiery Ferrari scenario, even if you *do* use Object references for everything. Because when objects are referred to by an Object reference type, Java *thinks* it's referring to an instance of type Object. And that means the only methods you're allowed to call on that object are the ones declared in class Object! So if you were to say:

```
Object o = new Ferrari();  
o.goFast(); //Not legal!
```

You wouldn't even make it past the compiler.

Because Java is a strongly-typed language, the compiler checks to make sure that you're calling a method on an object that's actually capable of *responding*. In other words, you can call a method on an object reference *only* if the class of the reference type actually *has* the method. We'll cover this in much greater detail a little later, so don't worry if the picture isn't crystal clear.

Using polymorphic references of type Object has a price...

Before you run off and start using type Object for all your ultra-flexible argument and return types, you need to consider a little issue of using type Object as a reference. And keep in mind that we're not talking about making instances of type Object; we're talking about making instances of some other type, but using a reference of type Object.

When you put an object into an `ArrayList<Dog>`, it goes in as a Dog, and comes out as a Dog:

```
ArrayList<Dog> myDogArrayList = new ArrayList<Dog>(); ← Make an ArrayList declared
to hold Dog objects.
Dog aDog = new Dog(); ← Make a Dog.
myDogArrayList.add(aDog); ← Add the Dog to the list.
Dog d = myDogArrayList.get(0); ← Assign the Dog from the list to a new Dog reference variable.
                                (Think of it as though the get() method declares a Dog return
                                type because you used ArrayList<Dog>.)
```

But what happens when you declare it as `ArrayList<Object>`? If you want to make an `ArrayList` that will literally take *any* kind of Object, you declare it like this:

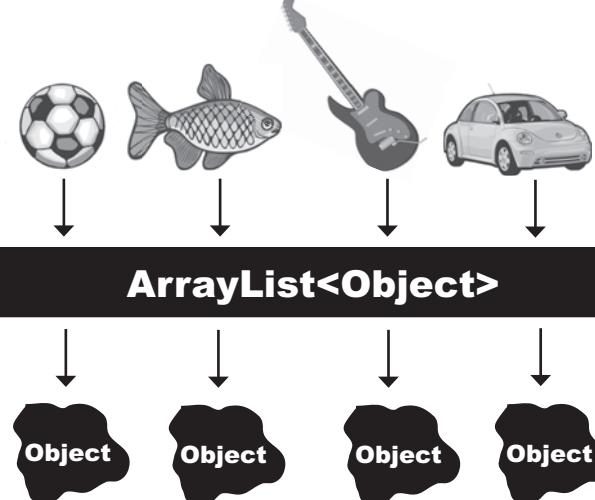
```
ArrayList<Object> myDogArrayList = new ArrayList<Object>(); ← Make an ArrayList declared
to hold any type of Object.
Dog aDog = new Dog(); ← Make a Dog. (These two steps are the same.)
myDogArrayList.add(aDog); ← Add the Dog to the list.
```

But what happens when you try to get the Dog object and assign it to a Dog reference?

 `Dog d = myDogArrayList.get(0);` NO!! Won't compile!! When you use `ArrayList<Object>`, the `get()` method returns type Object. The Compiler knows only that the object inherits from Object (somewhere in its inheritance tree) but it doesn't know it's a Dog!!

Everything comes out of an `ArrayList<Object>` as a reference of type Object, regardless of what the actual object is, or what the reference type was when you added the object to the list.

The objects go IN as **SoccerBall**, **Fish**, **Guitar**, and **Car**.



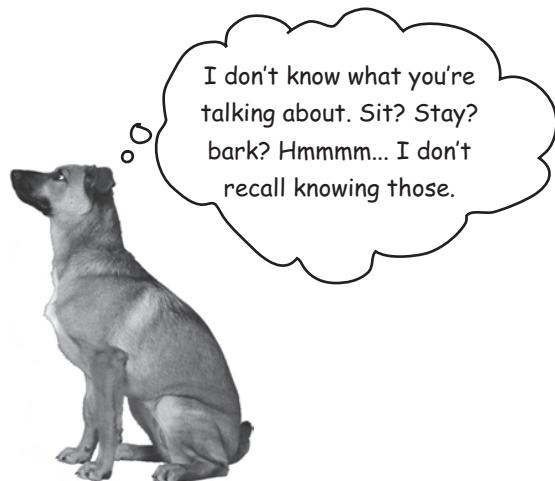
But they come OUT as though they were of type **Object**.

Objects come out of an `ArrayList<Object>` acting like they're generic instances of class Object. The Compiler cannot assume the object that comes out is of any type other than Object.

When a Dog loses its Dogness

When a Dog won't act like a Dog

The problem with having everything treated polymorphically as an Object is that the objects *appear* to lose (but not permanently) their true essence. *The Dog appears to lose its dogness.* Let's see what happens when we pass a Dog to a method that returns a reference to the same Dog object, but declares the return type as type Object rather than Dog.



BAD

```
public void go() {
    Dog aDog = new Dog();
    Dog sameDog = getobject(aDog);
}
```

This line won't work! Even though the method returned a reference to the very same Dog the argument referred to, the return type Object means the compiler won't let you assign the returned reference to anything but Object.

```
public Object getObject(Object o) {
    return o;
}
```

We're returning a reference to the same Dog, but as a return type of Object. This part is perfectly legal. Note: this is similar to how the get() method works when you have an ArrayList<Object> rather than an ArrayList<Dog>.

```
File Edit Window Help Remember
DogPolyTest.java:10: incompatible types
found   : java.lang.Object
required: Dog
    Dog sameDog = takeObjects(aDog);
                           ^
1 error
```

The compiler doesn't know that the thing returned from the method is actually a Dog, so it won't let you assign it to a Dog reference. (You'll see why on the next page.)

GOOD

```
public void go() {
    Dog aDog = new Dog();
    Object sameDog = getObject(aDog);
}

public Object getObject(Object o) {
    return o;
}
```

This works (although it may not be very useful, as you'll see in a moment) because you can assign ANYTHING to a reference of type Object, since every class passes the IS-A test for Object. Every object in Java is an instance of type Object, because every class in Java has Object at the top of its inheritance tree.

Objects don't bark.

So now we know that when an object is referenced by a variable declared as type Object, it can't be assigned to a variable declared with the actual object's type. And we know that this can happen when a return type or argument is declared as type Object, as would be the case, for example, when the object is put into an ArrayList of type Object using `ArrayList<Object>`. But what are the implications of this? Is it a problem to have to use an Object reference variable to refer to a Dog object? Let's try to call Dog methods on our Dog-That-Compiler-Thinks-Is-An-Object:

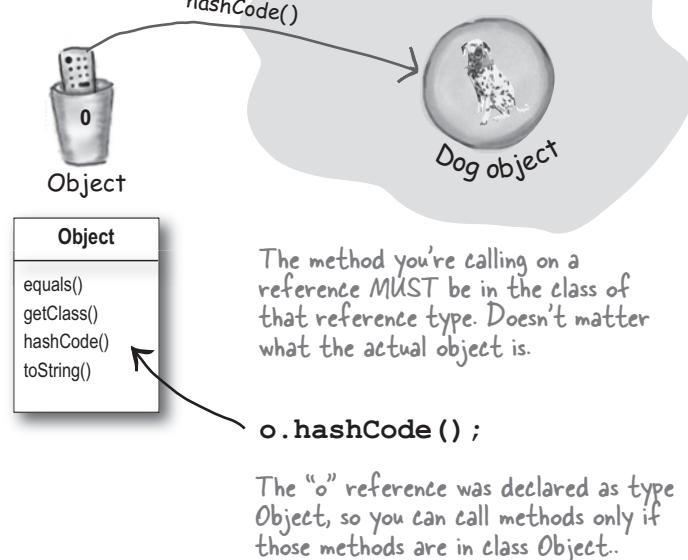
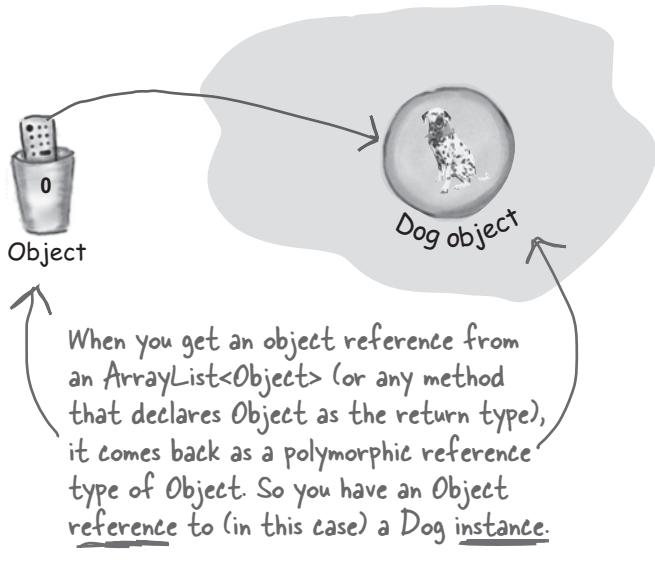
```
Object o = al.get(index);
int i = o.hashCode(); ← This is fine. Class Object has a
                           hashCode() method, so you can call
                           that method on ANY object in Java.
                           ← Can't do this!! The Object class has no idea what
                           it means to bark(). Even though YOU know it's
                           really a Dog at that index, the compiler doesn't..
```

Won't compile! → ~~o.bark();~~

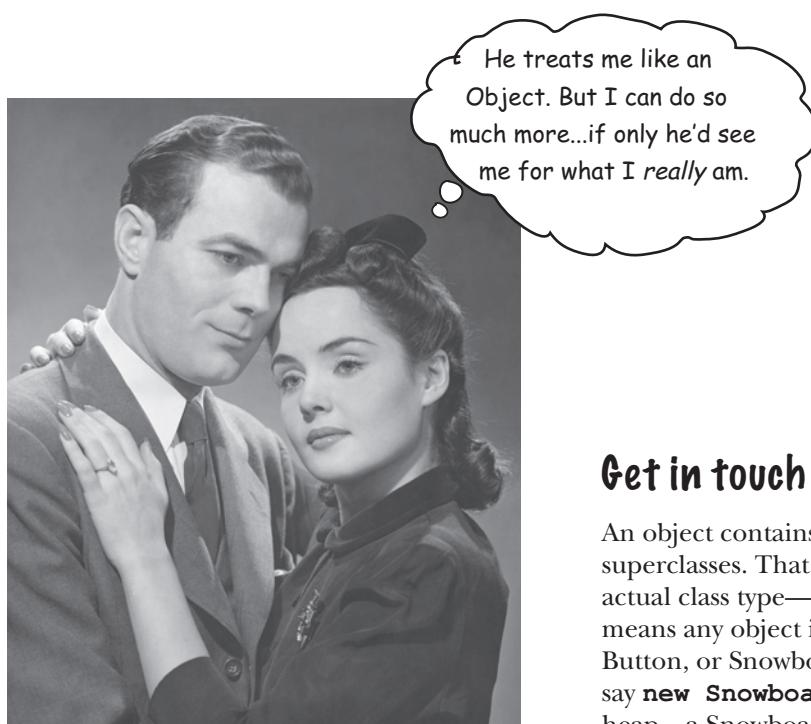
The compiler decides whether you can call a method based on the reference type, not the actual object type.

Even if you *know* the object is capable ("...but it really *is* a Dog, honest..."), the compiler sees it only as a generic Object. For all the compiler knows, you put a Button object out there. Or a Microwave object. Or some other thing that really doesn't know how to bark.

The compiler checks the class of the *reference type*—not the *object type*—to see if you can call a method using that reference.

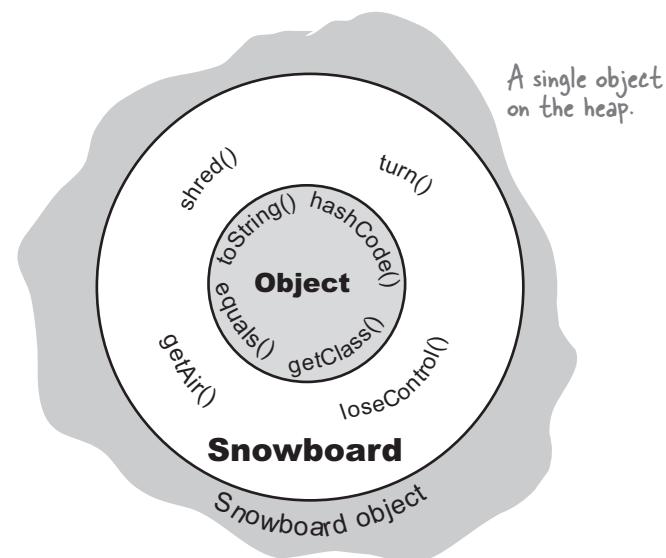
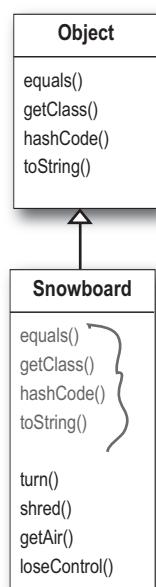


objects are Objects



Get in touch with your inner Object.

An object contains *everything* it inherits from each of its superclasses. That means *every* object—regardless of its actual class type—is *also* an instance of class `Object`. That means any object in Java can be treated not just as a `Dog`, `Button`, or `Snowboard`, but also as an `Object`. When you say `new Snowboard()`, you get a single object on the heap—a `Snowboard` object—but that `Snowboard` wraps itself around an inner core representing the `Object` (capital “O”) portion of itself.



There is only ONE object on the heap here. A `Snowboard` object. But it contains both the Snowboard class parts of itself and the Object class parts of itself.

**'Polymorphism' means
'many forms'.**

**You can treat a Snowboard as a
Snowboard or as an Object.**

If a reference is like a remote control, the remote control takes on more and more buttons as you move down the inheritance tree. A remote control (reference) of type Object has only a few buttons—the buttons for the exposed methods of class Object. But a remote control of type Snowboard includes all the buttons from class Object, plus any new buttons (for new methods) of class Snowboard. The more specific the class, the more buttons it may have.

Of course that's not always true; a subclass might not add any new methods, but simply override the methods of its superclass. The key point is that even if the *object* is of type Snowboard, an Object *reference* to the Snowboard object can't see the Snowboard-specific methods.

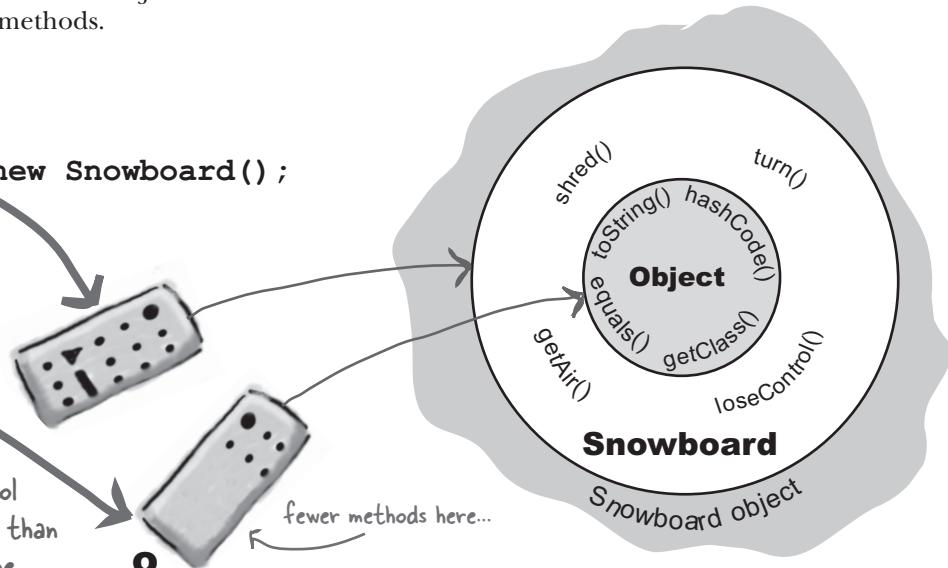
**When you put
an object in an
ArrayList<Object>, you
can treat it only as an
Object, regardless of
the type it was when
you put it in.**

**When you get a
reference from an
ArrayList<Object>, the
reference is always of
type Object.**

**That means you get an
Object remote control.**

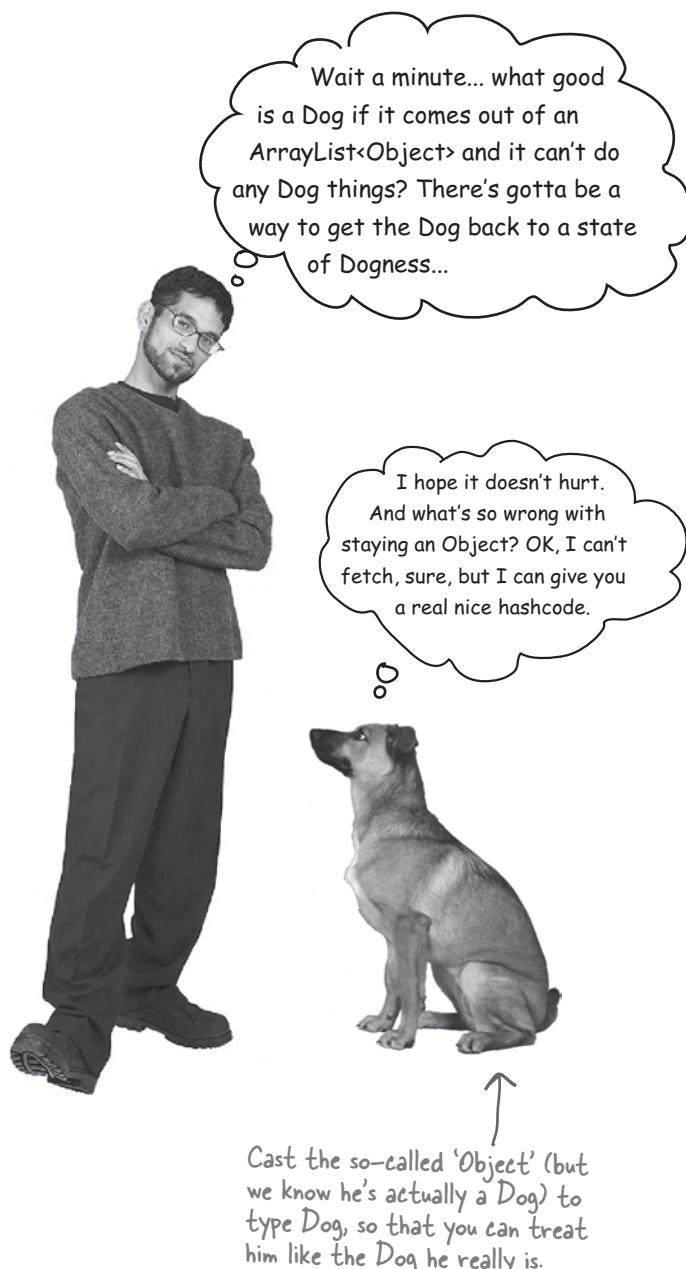
```
Snowboard s = new Snowboard();
Object o = s;
```

The Snowboard remote control (reference) has more buttons than an Object remote control. The Snowboard remote can see the full Snowboardness of the Snowboard object. It can access all the methods in Snowboard, including both the inherited Object methods and the methods from class Snowboard.

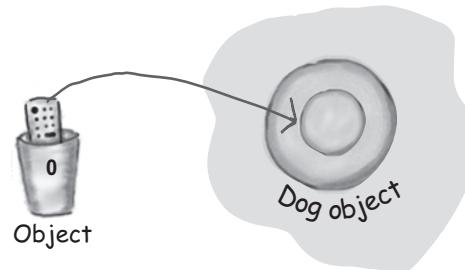


The Object reference can see only the Object parts of the Snowboard object. It can access only the methods of class Object. It has fewer buttons than the Snowboard remote control.

casting objects

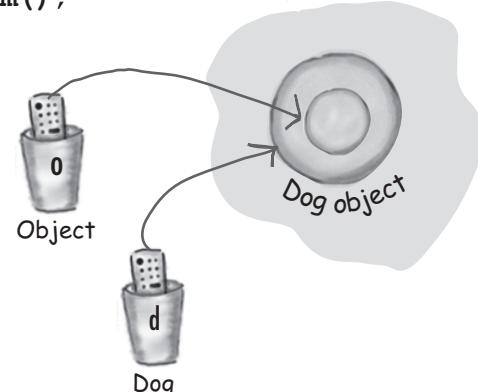


Casting an object reference back to its *real* type.



It's really still a Dog *object*, but if you want to call Dog-specific methods, you need a *reference* declared as type Dog. If you're *sure** the object is really a Dog, you can make a new Dog reference to it by copying the Object reference, and forcing that copy to go into a Dog reference variable, using a cast (Dog). You can use the new Dog reference to call Dog methods.

```
Object o = al.get(index);  
Dog d = (Dog) o; ← cast the Object back to  
d.roam(); a Dog we know is there.
```



*If you're *not* sure it's a Dog, you can use the `instanceof` operator to check. Because if you're wrong when you do the cast, you'll get a ClassCastException at runtime and come to a grinding halt.

```
if (o instanceof Dog) {  
    Dog d = (Dog) o;  
}
```

So now you've seen how much Java cares about the methods in the class of the reference variable.

You can call a method on an object only if the class of the reference variable has that method.

Think of the public methods in your class as your contract, your promise to the outside world about the things you can do.



When you write a class, you almost always *expose* some of the methods to code outside the class. To *expose* a method means you make a method *accessible*, usually by marking it public.

Imagine this scenario: you're writing code for a small business accounting program. A custom application for "Simon's Surf Shop". The good reuse user that you are, you found an Account class that appears to meet your needs perfectly, according to its documentation, anyway. Each account instance represents an individual customer's account with the store. So there you are minding your own business invoking the *credit()* and *debit()* methods on an account object when you realize you need to get a balance on an account. No problem—there's a *getBalance()* method that should do nicely.

Account
debit(double amt)
credit(double amt)
double getBalance()

Except... when you invoke the *getBalance()* method, the whole thing blows up at runtime. Forget the documentation, the class does not have that method. Yikes!

But that won't happen to you, because everytime you use the dot operator on a reference (*a.doStuff()*), the compiler looks at the *reference type* (the type 'a' was declared to be) and checks that class to guarantee the class has the method, and that the method does indeed take the argument you're passing and return the kind of value you're expecting to get back.

Just remember that the compiler checks the class of the reference variable, not the class of the actual *object* at the other end of the reference.

What if you need to change the contract?

OK, pretend you're a Dog. Your Dog class isn't the *only* contract that defines who you are. Remember, you inherit accessible (which usually means *public*) methods from all of your superclasses.

True, your Dog class defines a contract.

But not *all* of your contract.

Everything in class *Canine* is part of your contract.

Everything in class *Animal* is part of your contract.

Everything in class *Object* is part of your contract.

According to the IS-A test, you *are* each of those things—Canine, Animal, and Object.

But what if the person who designed your class had in mind the Animal simulation program, and now he wants to use you (class Dog) for a Science Fair Tutorial on Animal objects.

That's OK, you're probably reusable for that.

But what if later he wants to use you for a PetShop program? *You don't have any Pet behaviors.* A Pet needs methods like *beFriendly()* and *play()*.

OK, now pretend you're the Dog class programmer. No problem, right? Just add some more methods to the Dog class. You won't be breaking anyone else's code by *adding* methods, since you aren't touching the *existing* methods that someone else's code might be calling on Dog objects.

Can you see any drawbacks to that approach (adding Pet methods to the Dog class)?



Think about what **YOU** would do if **YOU** were the Dog class programmer and needed to modify the Dog so that it could do Pet things, too. We know that simply adding new Pet behaviors (methods) to the Dog class will work, and won't break anyone else's code.

But... this is a PetShop program. It has more than just Dogs! And what if someone wants to use your Dog class for a program that has *wild Dogs*? What do you think your options might be, and without worrying about how Java handles things, just try to imagine how you'd like to solve the problem of modifying some of your Animal classes to include Pet behaviors.

Stop right now and think about it, before you look at the next page where we begin to reveal everything.

(thus rendering the whole exercise completely useless, robbing you of your One Big Chance to burn some brain calories)

Let's explore some design options for reusing some of our existing classes in a PetShop program.

On the next few pages, we're going to walk through some possibilities. We're not yet worried about whether Java can actually *do* what we come up with. We'll cross that bridge once we have a good idea of some of the tradeoffs.

① Option one

We take the easy path, and put pet methods in class Animal.

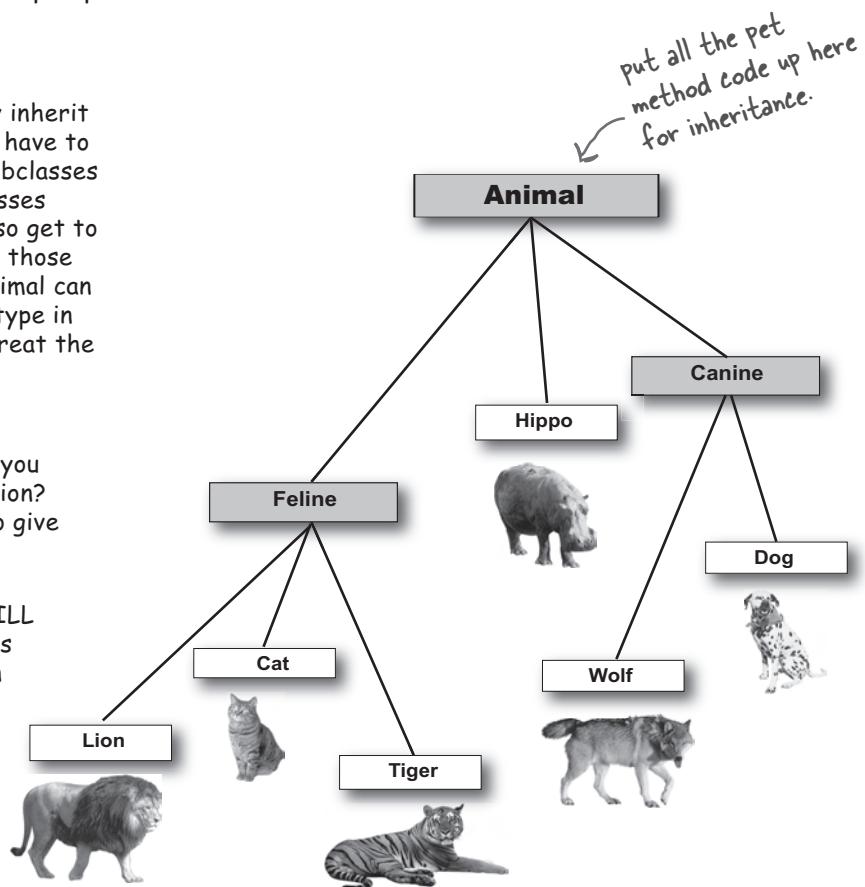
Pros:

All the Animals will instantly inherit the pet behaviors. We won't have to touch the existing Animal subclasses at all, and any Animal subclasses created in the future will also get to take advantage of inheriting those methods. That way, class Animal can be used as the polymorphic type in any program that wants to treat the Animals as pets.

Cons:

So... when was the last time you saw a Hippo at a pet shop? Lion? Wolf? Could be dangerous to give non-pets pet methods.

Also, we almost certainly WILL have to touch the pet classes like Dog and Cat, because (in our house, anyway) Dogs and Cats tend to implement pet behaviors VERY differently.



modifying existing classes

② Option two

We start with Option One, putting the pet methods in class Animal, but we make the methods abstract, forcing the Animal subclasses to override them.

Pros:

That would give us all the benefits of Option One, but without the drawback of having non-pet Animals running around with pet methods (like `beFriendly()`). All Animal classes would have the method (because it's in class Animal), but because it's abstract the non-pet Animal classes won't inherit any functionality. All classes MUST override the methods, but they can make the methods "do-nothings".

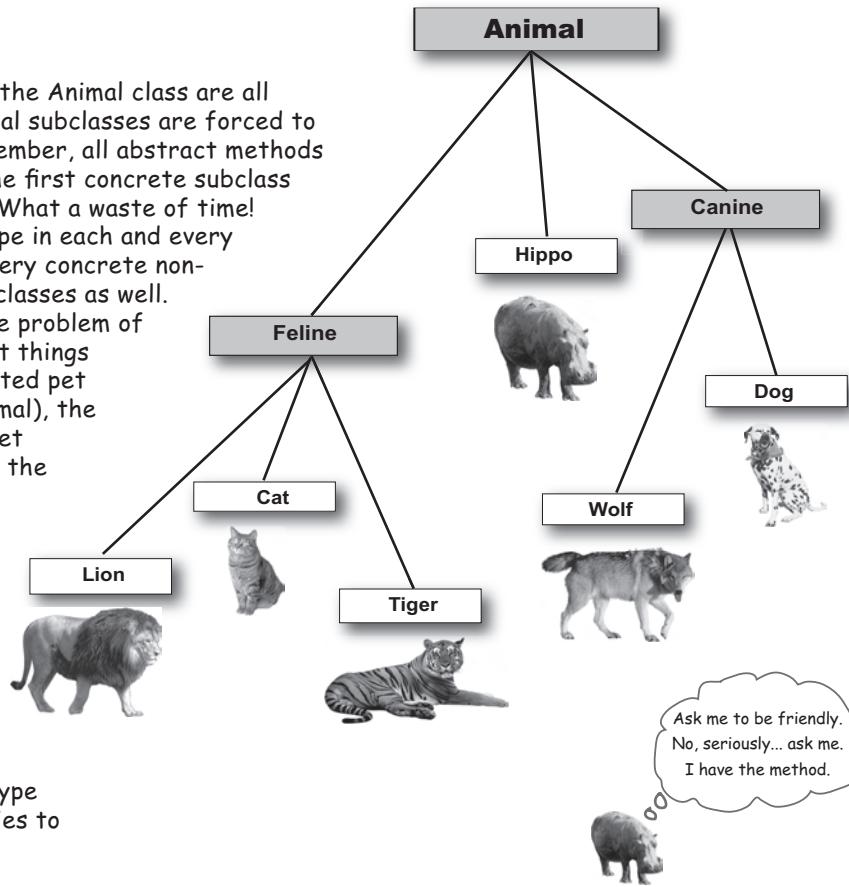
put all the pet methods up here, but with no implementations. Make all pet methods abstract.

Cons:

Because the pet methods in the Animal class are all abstract, the concrete Animal subclasses are forced to implement all of them. (Remember, all abstract methods MUST be implemented by the first concrete subclass down the inheritance tree.) What a waste of time! You have to sit there and type in each and every pet method into each and every concrete non-pet class, and all future subclasses as well.

And while this does solve the problem of non-pets actually DOING pet things (as they would if they inherited pet functionality from class Animal), the contract is bad. Every non-pet class would be announcing to the world that it, too, has those pet methods, even though the methods wouldn't actually DO anything when called.

This approach doesn't look good at all. It just seems wrong to stuff everything into class Animal that more than one Animal type might need, UNLESS it applies to ALL Animal subclasses.



③ Option three

Put the pet methods ONLY in the classes where they belong.

Pros:

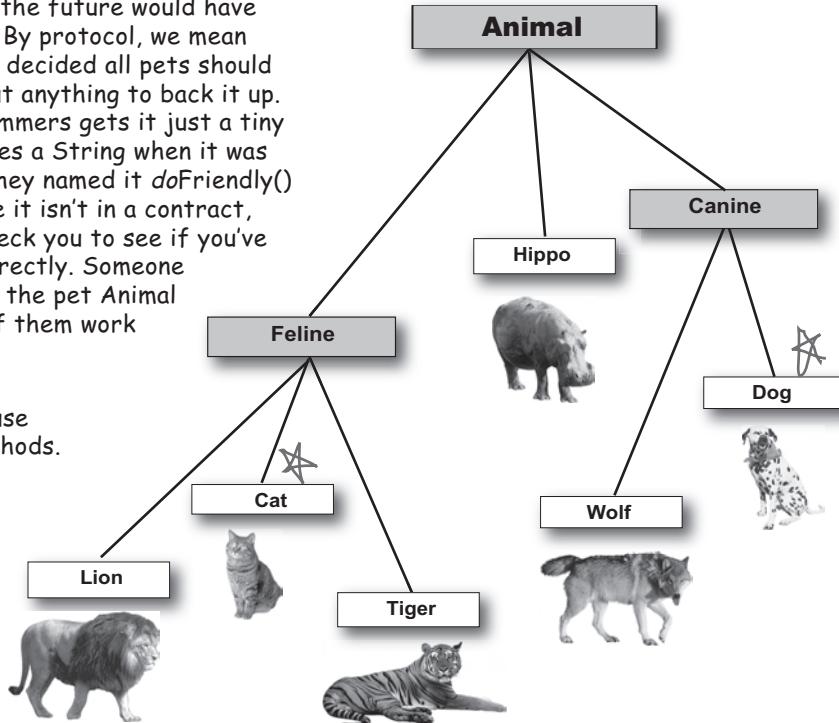
No more worries about Hippos greeting you at the door or licking your face. The methods are where they belong, and ONLY where they belong. Dogs can implement the methods and Cats can implement the methods, but nobody else has to know about them.

Cons:

Two Big Problems with this approach. First off, you'd have to agree to a protocol, and all programmers of pet Animal classes now and in the future would have to KNOW about the protocol. By protocol, we mean the exact methods that we've decided all pets should have. The pet contract without anything to back it up. But what if one of the programmers gets it just a tiny bit wrong? Like, a method takes a String when it was supposed to take an int? Or they named it *doFriendly()* instead of *beFriendly()*? Since it isn't in a contract, the compiler has no way to check you to see if you've implemented the methods correctly. Someone could easily come along to use the pet Animal classes and find that not all of them work quite right.

And second, you don't get to use polymorphism for the pet methods. Every class that needs to use pet behaviors would have to know about each and every class! In other words, you can't use *Animal* as the polymorphic type now, because the compiler won't let you call a Pet method on an *Animal* reference (even if it's really a Dog object) because class *Animal* doesn't have the method.

~~Put the pet methods ONLY in the Animal classes that can be pets, instead of in Animal.~~

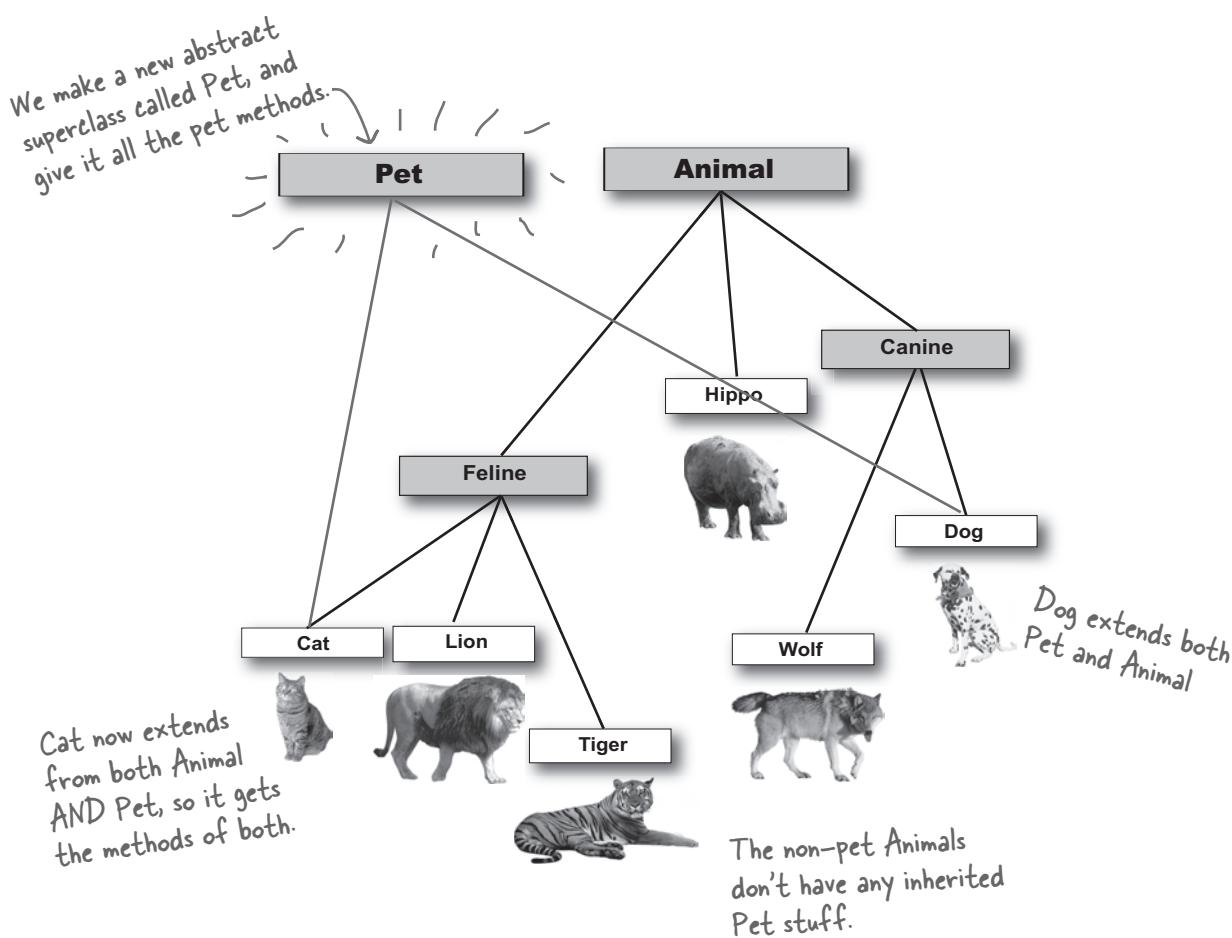


multiple inheritance?

So what we **REALLY** need is:

- A way to have pet behavior in **just** the pet classes
- A way to guarantee that all pet classes have all of the same methods defined (same name, same arguments, same return types, no missing methods, etc.), without having to cross your fingers and hope all the programmers get it right.
- A way to take advantage of polymorphism so that all pets can have their pet methods called, without having to use arguments, return types, and arrays for each and every pet class.

It looks like we need TWO superclasses at the top



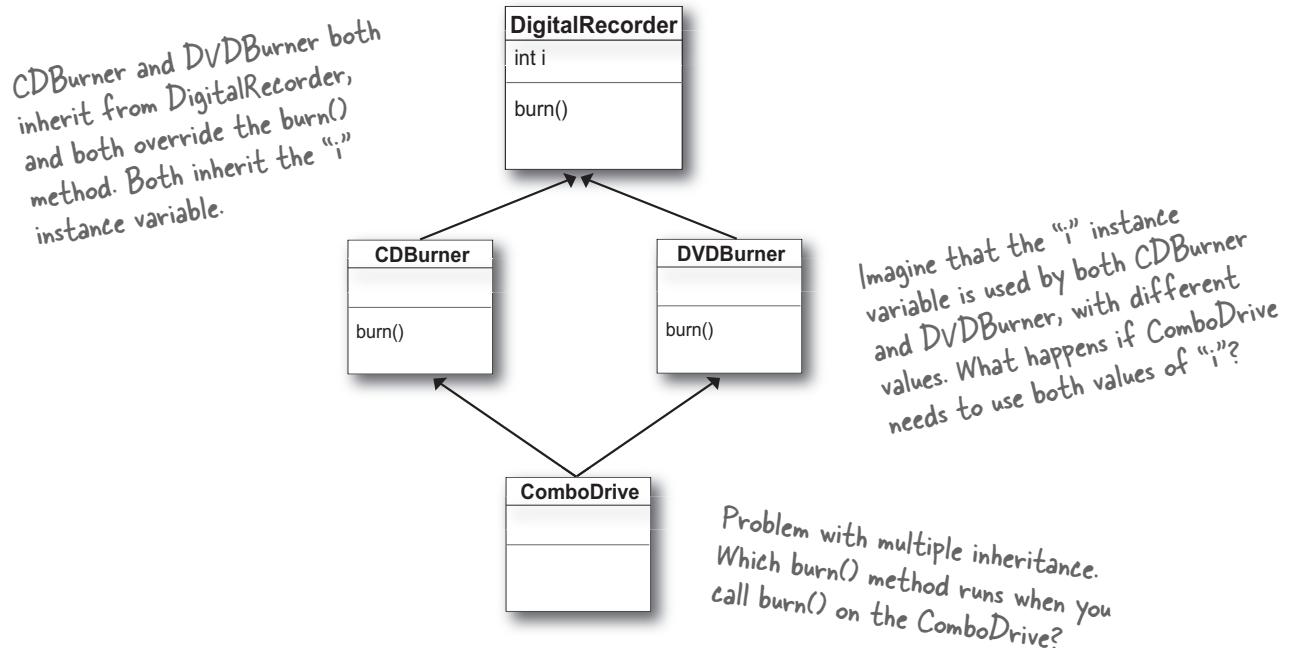
There's just one problem with the "two superclasses" approach...

It's called “multiple inheritance” and it can be a Really Bad Thing.

That is, if it were possible to do in Java.

But it isn't, because multiple inheritance has a problem known as The Deadly Diamond of Death.

Deadly Diamond of Death



A language that allows the Deadly Diamond of Death can lead to some ugly complexities, because you have to have special rules to deal with the potential ambiguities. And extra rules means extra work for you both in *learning* those rules and watching out for those "special cases". Java is supposed to be *simple*, with consistent rules that don't blow up under some scenarios. So Java (unlike C++) protects you from having to think about the Deadly Diamond of Death. But that brings us back to the original problem!
How do we handle the Animal/Pet thing?

interfaces

Interface to the rescue!

Java gives you a solution. An *interface*. Not a *GUI* interface, not the generic use of the *word* interface as in, “That’s the public interface for the Button class API,” but the Java keyword **interface**.

A Java interface solves your multiple inheritance problem by giving you much of the polymorphic *benefits* of multiple inheritance without the pain and suffering from the Deadly Diamond of Death (DDD).

The way in which interfaces side-step the DDD is surprisingly simple: **make all the methods abstract!** That way, the subclass **must** implement the methods (remember, abstract methods *must* be implemented by the first concrete subclass), so at runtime the JVM isn’t confused about *which* of the two inherited versions it’s supposed to call.

Pet
abstract void beFriendly(); abstract void play();

A Java interface is like a 100% pure abstract class.

All methods in an interface are abstract, so any class that IS-A Pet MUST implement (i.e. override) the methods of Pet.

To DEFINE an interface:

```
public interface Pet {...}
```

↑
Use the keyword “interface” instead of “class”

To IMPLEMENT an interface:

```
public class Dog extends Canine implements Pet {...}
```

↑
Use the keyword “implements” followed by the interface name. Note that when you implement an interface you still get to extend a class

Making and Implementing the Pet interface

You say 'interface' instead of 'class' here

```
public interface Pet {
    public abstract void beFriendly();
    public abstract void play();
}
```

interface methods are implicitly public and abstract, so typing in 'public' and 'abstract' is optional (in fact, it's not considered 'good style' to type the words in, but we did here just to reinforce it, and because we've never been slaves to fashion...)

All interface methods are abstract, so they MUST end in semicolons. Remember, they have no body!

Dog IS-A Animal
and Dog IS-A Pet

```
public class Dog extends Canine implements Pet {
```

```
    public void beFriendly() {...}
```

```
    public void play() {...}
```

```
    public void roam() {...}
```

```
    public void eat() {...} ← These are just normal
                                overriding methods.
```

You say 'implements' followed by the name of the interface.

You SAID you are a Pet, so you MUST implement the Pet methods. It's your instead of semicolons.

^{there are no} Dumb Questions

Q: Wait a minute, interfaces don't really give you multiple inheritance, because you can't put any implementation code in them. If all the methods are abstract, what does an interface really buy you?

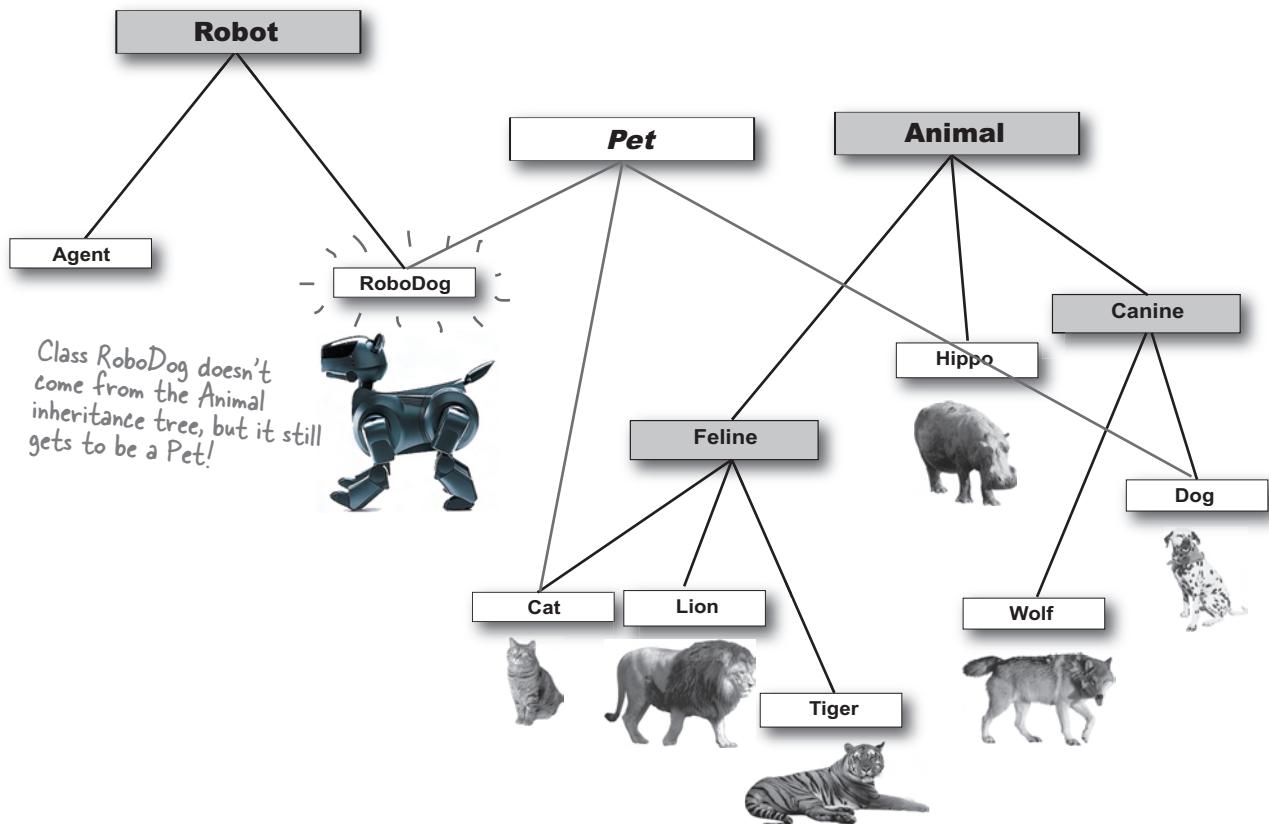
A: Polymorphism, polymorphism, polymorphism. Interfaces are the ultimate in flexibility, because if you use interfaces instead of concrete subclasses (or even abstract superclass types) as arguments and return

types, you can pass anything that implements that interface. And think about it—with an interface, a class doesn't have to come from just one inheritance tree. A class can extend one class, and implement an interface. But another class might implement the same interface, yet come from a completely different inheritance tree! So you get to treat an object by the role it plays, rather than by the class type from which it was instantiated. In fact, if you wrote your code to use interfaces, you wouldn't even have to give anyone a superclass that they had

to extend. You could just give them the interface and say, "Here, I don't care what kind of class inheritance structure you come from, just implement this interface and you'll be good to go."

The fact that you can't put in implementation code turns out not to be a problem for most good designs, because most interface methods wouldn't make sense if implemented in a generic way. In other words, most interface methods would need to be overridden even if the methods weren't forced to be abstract.

Classes from different inheritance trees can implement the same interface.



When you use a *class* as a polymorphic type (like an array of type Animal or a method that takes a Canine argument), the objects you can stick in that type must be from the same inheritance tree. But not just anywhere in the inheritance tree; the objects must be from a class that is a subclass of the polymorphic type. An argument of type Canine can accept a Wolf and a Dog, but not a Cat or a Hippo.

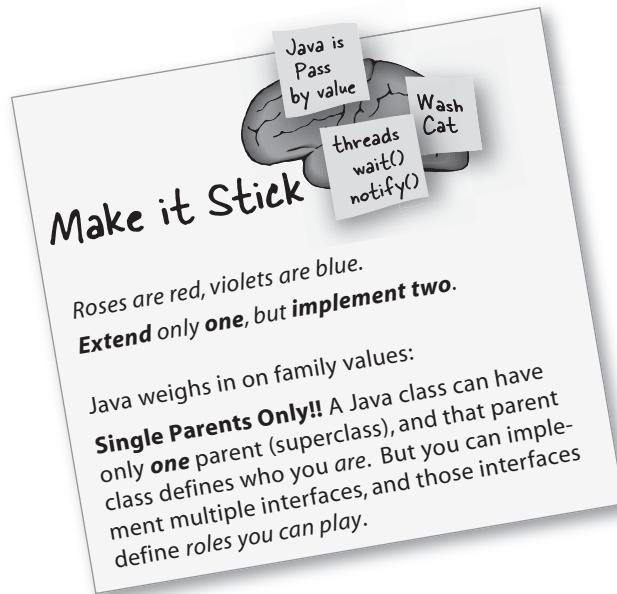
But when you use an *interface* as a polymorphic type (like an array of Pets), the objects can be from *anywhere* in the inheritance tree. The only requirement is that the objects are from a class that *implements* the interface. Allowing classes in different inheritance trees to implement a common interface is crucial in the Java API. Do you want an object to be able to save its state to a file? Implement the Serializable interface. Do you need objects to run

their methods in a separate thread of execution? Implement Runnable. You get the idea. You'll learn more about Serializable and Runnable in later chapters, but for now, remember that classes from *any* place in the inheritance tree might need to implement those interfaces. Nearly *any* class might want to be saveable or runnable.

Better still, a class can implement multiple interfaces!

A Dog object IS-A Canine, and IS-A Animal, and IS-A Object, all through inheritance. But a Dog IS-A Pet through interface implementation, and the Dog might implement other interfaces as well. You could say:

```
public class Dog extends Animal implements Pet, Saveable, Paintable { ... }
```



How do you know whether to make a class, a subclass, an **abstract class**, or an **interface**?

- ▶ Make a class that doesn't extend anything (other than Object) when your new class doesn't pass the IS-A test for any other type.
- ▶ Make a subclass (in other words, *extend* a class) only when you need to make a **more specific** version of a class and need to override or add new behaviors.
- ▶ Use an abstract class when you want to define a **template** for a group of subclasses, and you have at least *some* implementation code that all subclasses could use. Make the class abstract when you want to guarantee that nobody can make objects of that type.
- ▶ Use an interface when you want to define a **role** that other classes can play, regardless of where those classes are in the inheritance tree.

using super

Invoking the superclass version of a method

Q: What if you make a concrete subclass and you need to override a method, but you want the behavior in the superclass version of the method? In other words, what if you don't need to replace the method with an override, but you just want to add to it with some additional specific code.

A: Ahhh... think about the meaning of the word 'extends'. One area of good OO design looks at how to design concrete code that's *meant* to be overridden. In other words, you write method code in, say, an abstract class, that does work that's generic enough to support typical concrete implementations. But, the concrete code isn't enough to handle *all* of the subclass-specific work. So the subclass overrides the method and *extends* it by adding the rest of the code. The keyword super lets you invoke a superclass version of an overridden method, from within the subclass.

If method code inside a BuzzwordReport subclass says:
super.runReport();

the runReport() method inside the superclass Report will run

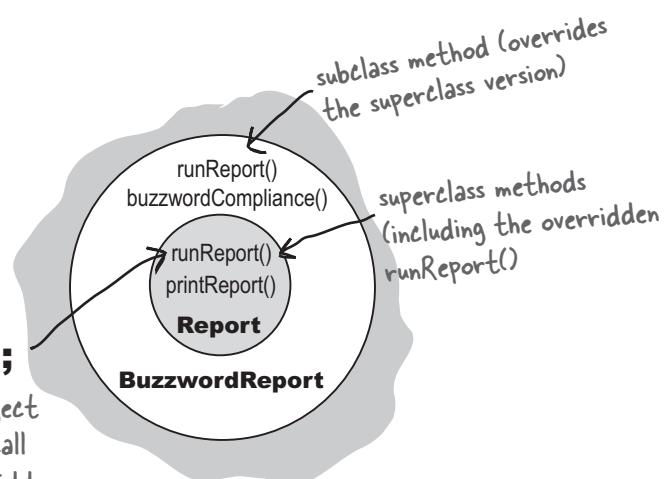
super.runReport();

A reference to the subclass object (BuzzwordReport) will always call the subclass version of an overridden method. That's polymorphism. But the subclass code can call **super.runReport()** to invoke the superclass version.

```
abstract class Report {  
    void runReport() {  
        // set-up report  
    }  
    void printReport() {  
        // generic printing  
    }  
}  
  
class BuzzwordsReport extends Report {  
  
    void runReport() {  
        super.runReport();  
        buzzwordCompliance();  
        printReport();  
    }  
    void buzzwordCompliance() { ... }  
}
```

superclass version of the method does important stuff that subclasses could use

call superclass version, then come back and do some subclass-specific stuff



The **super** keyword is really a reference to the superclass portion of an object. When subclass code uses **super**, as in **super.runReport()**, the superclass version of the method will run.



BULLET POINTS

- ▶ When you don't want a class to be instantiated (in other words, you don't want anyone to make a new object of that class type) mark the class with the **abstract** keyword.
- ▶ An abstract class can have both abstract and non-abstract methods.
- ▶ If a class has even one abstract method, the class must be marked abstract.
- ▶ An abstract method has no body, and the declaration ends with a semicolon (no curly braces).
- ▶ All abstract methods must be implemented in the first concrete subclass in the inheritance tree.
- ▶ Every class in Java is either a direct or indirect subclass of class **Object** (`java.lang.Object`).
- ▶ Methods can be declared with Object arguments and/or return types.
- ▶ You can call methods on an object *only* if the methods are in the class (or interface) used as the *reference* variable type, regardless of the actual *object* type. So, a reference variable of type Object can be used only to call methods defined in class Object, regardless of the type of the object to which the reference refers.
- ▶ A reference variable of type Object can't be assigned to any other reference type without a *cast*. A cast can be used to assign a reference variable of one type to a reference variable of a subtype, but at runtime the cast will fail if the object on the heap is NOT of a type compatible with the cast.
Example: `Dog d = (Dog) x.getObject(aDog);`
- ▶ All objects come out of an `ArrayList<Object>` as type Object (meaning, they can be referenced only by an Object reference variable, unless you use a *cast*).
- ▶ Multiple inheritance is not allowed in Java, because of the problems associated with the "Deadly Diamond of Death". That means you can extend only one class (i.e. you can have only one immediate superclass).
- ▶ An interface is like a 100% pure abstract class. It defines *only* abstract methods.
- ▶ Create an interface using the **interface** keyword instead of the word **class**.
- ▶ Implement an interface using the keyword **implements**
Example: `Dog implements Pet`
- ▶ Your class can implement multiple interfaces.
- ▶ A class that implements an interface *must* implement all the methods of the interface, since **all interface methods are implicitly public and abstract**.
- ▶ To invoke the superclass version of a method from a subclass that's overridden the method, use the **super** keyword. Example: `super.runReport();`

Q: There's still something strange here... you never explained how it is that `ArrayList<Dog>` gives back Dog references that don't need to be cast, yet the `ArrayList` class uses Object in its methods, not Dog (or DotCom or anything else). What's the special trick going on when you say `ArrayList<Dog>`?

A: You're right for calling it a special trick. In fact it is a special trick that `ArrayList<Dog>` gives back Dogs without you having to do any cast, since it looks like `ArrayList` methods don't know anything about Dogs, or any type besides Object.

The short answer is that *the compiler puts in the cast for you!* When you say `ArrayList<Dog>`, there is no special class that has methods to take and return Dog objects, but instead the `<Dog>` is a signal to the compiler that you want the compiler to let you put ONLY Dog objects in and to stop you if you try to add any other type to the list. And since the compiler stops you from adding anything but Dogs to the `ArrayList`, the compiler also knows that it's safe to cast anything that comes out of that `ArrayList` do a Dog reference. In other words, using `ArrayList<Dog>` saves you from having to cast the Dog you get back. But it's much more important than that... because remember, a cast can fail at runtime, and wouldn't you rather have your errors happen at compile time rather than, say, when your customer is using it for something critical?

But there's a lot more to this story, and we'll get into all the details in the Collections chapter.

exercise: What's the Picture?



Here's your chance to demonstrate your artistic abilities. On the left you'll find sets of class and interface declarations. Your job is to draw the associated class diagrams on the right. We did the first one for you. Use a dashed line for "implements" and a solid line for "extends".

Given:

1) public interface Foo { }
public class Bar implements Foo { }

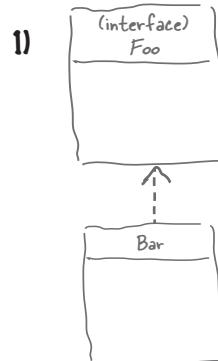
2) public interface Vinn { }
public abstract class Vout implements Vinn { }

3) public abstract class Muffie implements Whuffie { }
public class Fluffie extends Muffie { }
public interface Whuffie { }

4) public class Zoop { }
public class Boop extends Zoop { }
public class Goop extends Boop { }

5) public class Gamma extends Delta implements Epsilon { }
public interface Epsilon { }
public interface Beta { }
public class Alpha extends Gamma implements Beta { }
public class Delta { }

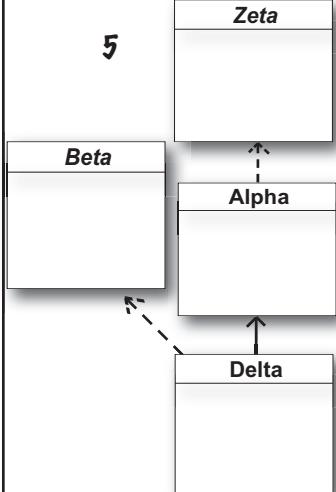
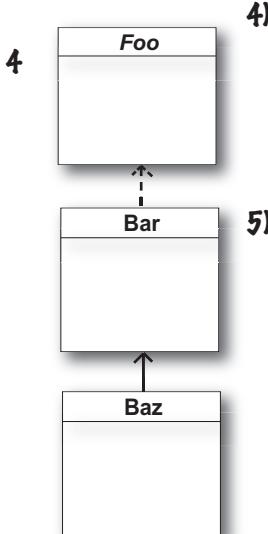
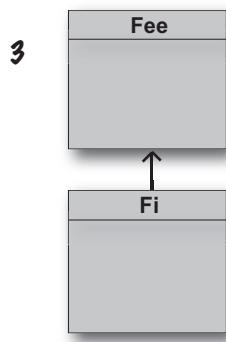
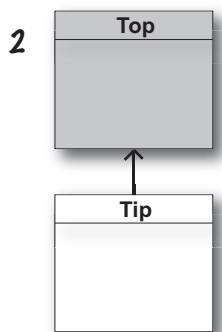
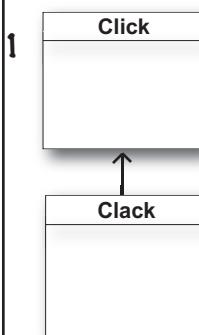
What's the Picture ?



3)

4)

5)

**Given:****What's the Declaration ?**

1) public class Click { }

public class Clack extends Click { }

2)

3)

4)

5)

KEY	
	extends
	implements
	class
	interface
	abstract class

puzzle: Pool Puzzle



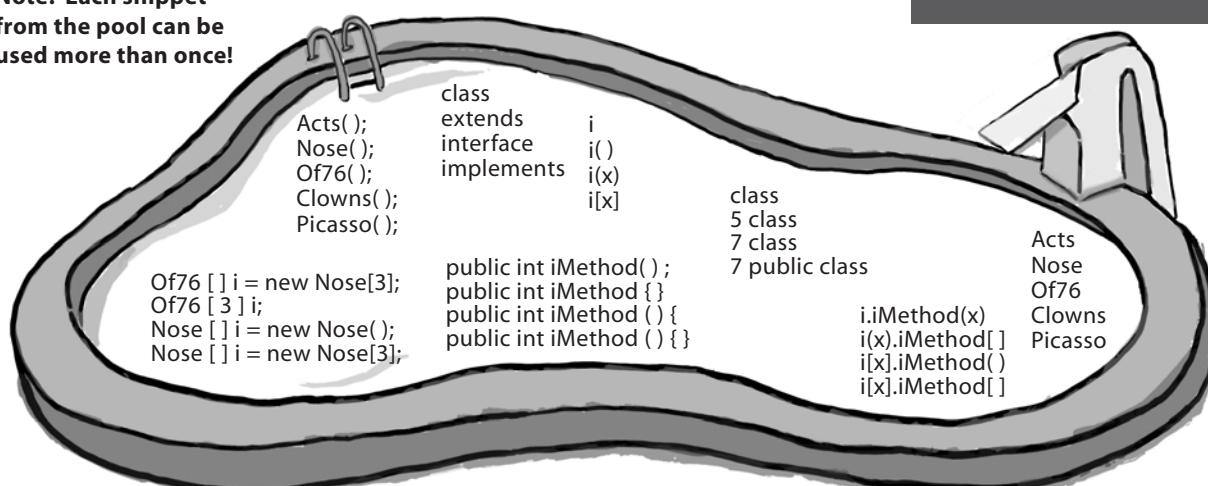
Pool Puzzle



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

```
_____ Nose {  
}  
  
abstract class Picasso implements _____ {  
    _____  
    return 7;  
}  
}  
  
class _____ { }  
  
class _____ {  
    _____  
    return 5;  
}  
}
```

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



```
public _____ extends Clowns {  
  
public static void main(String [] args) {  
  
    i[0] = new _____  
    i[1] = new _____  
    i[2] = new _____  
    for(int x = 0; x < 3; x++) {  
        System.out.println(_____  
            + " " + _____.getClass( ) );  
    }  
}
```

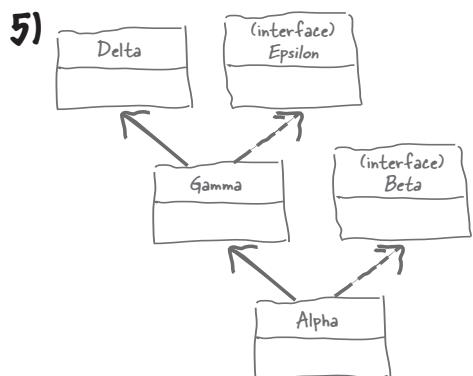
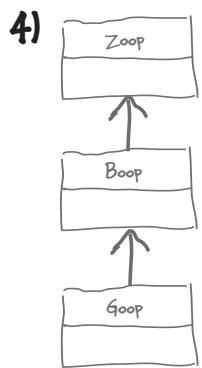
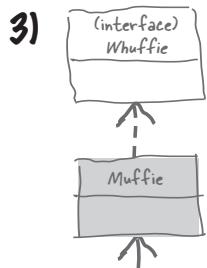
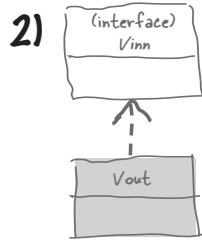
Output

```
File Edit Window Help BeAfraid  
%java _____  
5 class Acts  
7 class Clowns  
_____ Of76
```



Exercise Solutions

What's the Picture ?



What's the Declaration ?

2) `public abstract class Top { }`
`public class Tip extends Top { }`

3) `public abstract class Fee { }`
`public abstract class Fi extends Fee { }`

4) `public interface Foo { }`
`public class Bar implements Foo { }`
`public class Baz extends Bar { }`

5) `public interface Zeta { }`
`public class Alpha implements Zeta { }`
`public interface Beta { }`
`public class Delta extends Alpha implements Beta { }`

puzzle solution



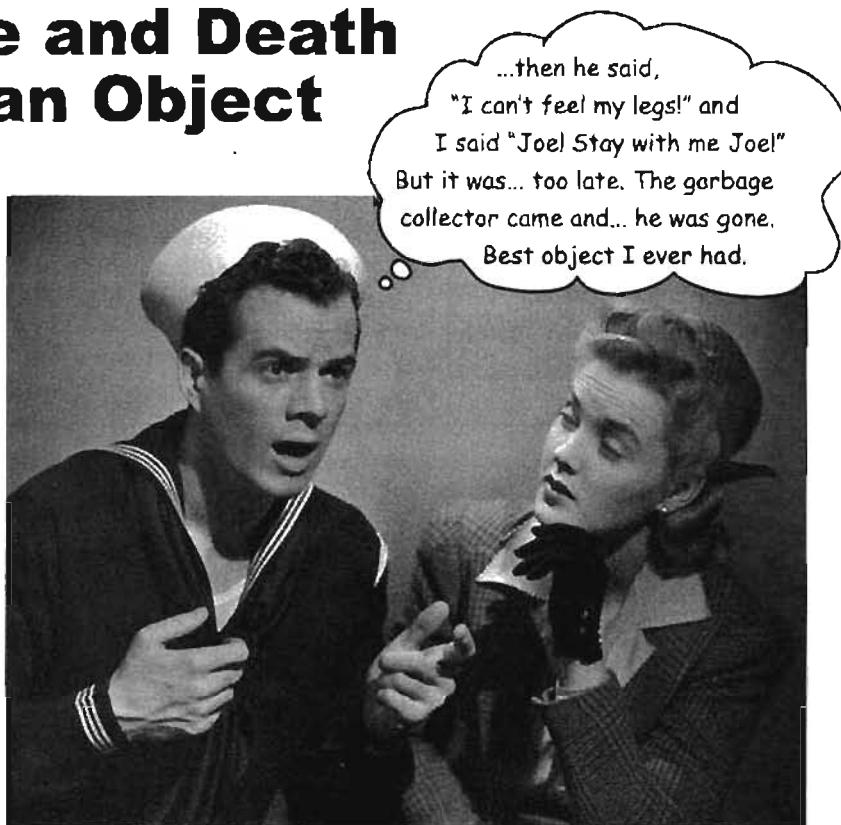
```
interface Nose {  
    public int iMethod();  
}  
abstract class Picasso implements Nose {  
    public int iMethod() {  
        return 7;  
    }  
}  
class Clowns extends Picasso {}  
  
class Acts extends Picasso {  
    public int iMethod() {  
        return 5;  
    }  
}
```

```
public class Of76 extends Clowns {  
    public static void main(String [] args) {  
        Nose [ ] i = new Nose [3];  
        i[0] = new Acts();  
        i[1] = new Clowns();  
        i[2] = new Of76();  
        for(int x = 0; x < 3; x++) {  
            System.out.println( i[x].iMethod()  
                + " " + i[x].getClass() );  
        }  
    }  
}
```

Output

```
File Edit Window Help KillTheMime  
%java Of76  
5 class Acts  
7 class Clowns  
7 class Of76
```

Life and Death of an Object



...then he said,
"I can't feel my legs!" and
I said "Joel Stay with me Joel"
But it was... too late. The garbage
collector came and... he was gone.
Best object I ever had.

Objects are born and objects die. You're in charge of an object's lifecycle. You decide when and how to **construct** it. You decide when to **destroy** it. Except you don't actually *destroy* the object yourself, you simply *abandon* it. But once it's abandoned, the heartless **Garbage Collector (gc)** can vaporize it, reclaiming the memory that object was using. If you're gonna write Java, you're gonna create objects. Sooner or later, you're gonna have to let some of them go, or risk running out of RAM. In this chapter we look at how objects are created, where they live while they're alive, and how to keep or abandon them efficiently. That means we'll talk about the heap, the stack, scope, constructors, super constructors, null references, and more. Warning: this chapter contains material about object death that some may find disturbing. Best not to get too attached.

the stack and the heap

The Stack and the Heap: where things live

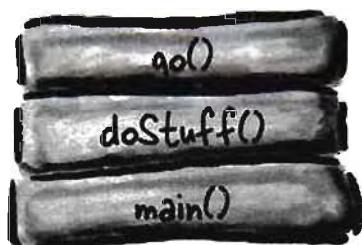
Before we can understand what really happens when you create an object, we have to step back a bit. We need to learn more about where everything lives (and for how long) in Java. That means we need to learn more about the Stack and the Heap. In Java, we (programmers) care about two areas of memory—the one where objects live (the heap), and the one where method invocations and local variables live (the stack). When a JVM starts up, it gets a chunk of memory from the underlying OS, and uses it to run your Java program. How much memory, and whether or not you can tweak it, is dependent on which version of the JVM (and on which platform) you're

running. But usually you *won't* have anything to say about it. And with good programming, you probably *won't* care (more on that a little later).

We know that all *objects* live on the garbage-collectible heap, but we haven't yet looked at where *variables* live. And where a variable lives depends on what *kind* of variable it is. And by "kind", we don't mean *type* (i.e. primitive or object reference). The two *kinds* of variables whose lives we care about now are *instance* variables and *local* variables. Local variables are also known as *stack* variables, which is a big clue for where they live.

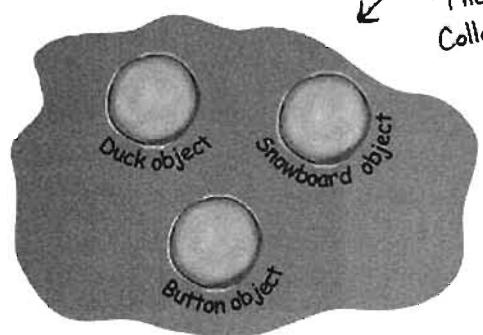
The Stack

Where method invocations and local variables live



The Heap

Where ALL objects live



Instance Variables

Instance variables are declared inside a *class* but not inside a *method*. They represent the "fields" that each individual object has (which can be filled with different values for each instance of the class). Instance variables live inside the object they belong to.

```
public class Duck {  
    int size; // Every Duck has a "size"  
              // instance variable.  
}
```

Local Variables

Local variables are declared inside a *method*, including *method parameters*. They're temporary, and live only as long as the method is on the stack (in other words, as long as the method has not reached the closing curly brace).

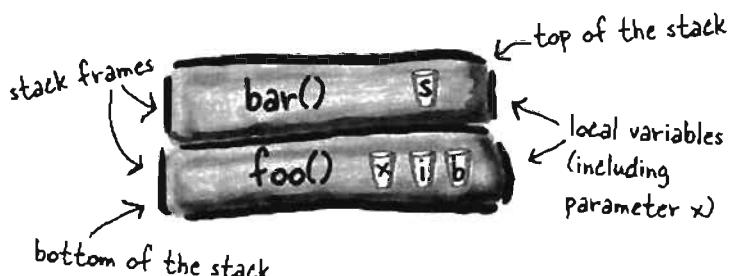
```
public void foo(int x) {  
    int i = x + 3; // The parameter x and  
    boolean b = true; // the variables i and b  
                      // are all local variables.  
}
```

Methods are stacked

When you call a method, the method lands on the top of a call stack. That new thing that's actually pushed onto the stack is the **stack frame**, and it holds the state of the method including which line of code is executing, and the values of all local variables.

The method at the *top* of the stack is always the currently-running method for that stack (for now, assume there's only one stack, but in chapter 14 we'll add more.) A method stays on the stack until the method hits its closing curly brace (which means the method's done). If method `foo()` calls method `bar()`, method `bar()` is stacked on top of method `foo()`.

A call stack with two methods



The method on the top of the stack is always the currently-executing method.

```
public void doStuff() {
    boolean b = true;
    go(4);
}

public void go(int x) {
    int z = x + 24;
    crazy();
    // imagine more code here
}

public void crazy() {
    char c = 'a';
}
```

A stack scenario

The code on the left is a snippet (we don't care what the rest of the class looks like) with three methods. The first method (`doStuff()`) calls the second method (`go()`), and the second method calls the third (`crazy()`). Each method declares one local variable within the body of the method, and method `go()` also declares a parameter variable (which means `go()` has two local variables).

- ❶ Code from another class calls `doStuff()`, and `doStuff()` goes into a stack frame at the top of the stack. The boolean variable named 'b' goes on the `doStuff()` stack frame.



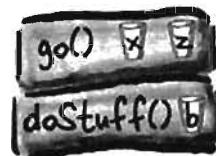
- ❷ `doStuff()` calls `go()`. `go()` is pushed on top of the stack. Variables 'x' and 'z' are in the `go()` stack frame.



- ❸ `go()` calls `crazy()`. `crazy()` is now on the top of the stack, with variable 'c' in the frame.



- ❹ `crazy()` completes, and its stack frame is popped off the stack. Execution goes back to the `go()` method, and picks up at the line following the call to `crazy()`.



object references on the stack

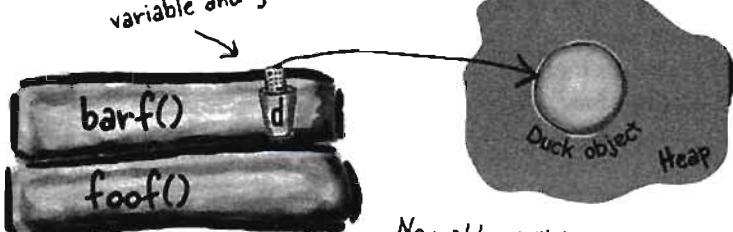
What about local variables that are objects?

Remember, a non-primitive variable holds a *reference* to an object, not the object itself. You already know where objects live—on the heap. It doesn't matter where they're declared or created. *If the local variable is a reference to an object, only the variable (the reference/remote control) goes on the stack.*

The object itself still goes in the heap.

```
public class StackRef {  
    public void foof() {  
        barf();  
    }  
  
    public void barf() {  
        Duck d = new Duck(24);  
    }  
}
```

barf() declares and creates a new
Duck reference variable 'd' (since it's
declared inside the method, it's a local
variable and goes on the stack.)



No matter WHERE the object reference variable is declared (inside a method vs. as an instance variable of a class) the object always always always goes on the heap.

there are no
Dumb Questions

Q: One more time, WHY are we learning the whole stack/heap thing? How does this help me? Do I really need to learn about it?

A: Knowing the fundamentals of the Java Stack and Heap is crucial if you want to understand variable scope, object creation issues, memory management, threads, and exception handling. We cover threads and exception handling in later chapters but the others you'll learn in this one. You do not need to know anything about how the Stack and Heap are implemented in any particular JVM and/or platform. Everything you need to know about the Stack and Heap is on this page and the previous one. If you nail these pages, all the other topics that depend on your knowing this stuff will go much, much, much easier. Once again, some day you will SO thank us for shoving Stacks and Heaps down your throat.

BULLET POINTS

- ▶ Java has two areas of memory we care about: the Stack and the Heap.
- ▶ Instance variables are variables declared inside a class but outside any method.
- ▶ Local variables are variables declared inside a method or method parameter.
- ▶ All local variables live on the stack, in the frame corresponding to the method where the variables are declared.
- ▶ Object reference variables work just like primitive variables—if the reference is declared as a local variable, it goes on the stack.
- ▶ All objects live in the heap, regardless of whether the reference is a local or instance variable.

If local variables live on the stack, where do instance variables live?

When you say `new CellPhone()`, Java has to make space on the Heap for that `CellPhone`. But how *much* space? Enough for the object, which means enough to house all of the object's instance variables. That's right, instance variables live on the Heap, inside the object they belong to.

Remember that the *values* of an object's instance variables live inside the object. If the instance variables are all primitives, Java makes space for the instance variables based on the primitive type. An `int` needs 32 bits, a `long` 64 bits, etc. Java doesn't care about the value inside primitive variables; the bit-size of an `int` variable is the same (32 bits) whether the value of the `int` is 32,000,000 or 32.

But what if the instance variables are *objects*? What if `CellPhone HAS-A Antenna`? In other words, `CellPhone` has a reference variable of type `Antenna`.

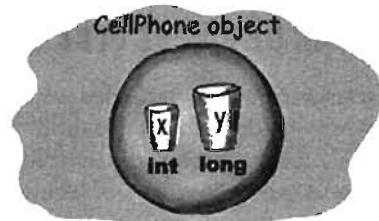
When the new object has instance variables that are object references rather than primitives, the real question is: does the object need space for all of the objects it holds references to? The answer is, *not exactly*. No matter what, Java has to make space for the instance variable *values*. But remember that a reference variable value is not the whole *object*, but merely a *remote control* to the object. So if `CellPhone` has an instance variable declared as the non-primitive type `Antenna`, Java makes space within the `CellPhone` object only for the `Antenna`'s *remote control* (i.e. reference variable) but not the `Antenna` *object*.

Well then when does the `Antenna object` get space on the Heap? First we have to find out when the `Antenna` object itself is created. That depends on the instance variable declaration. If the instance variable is declared but no object is assigned to it, then only the space for the reference variable (the remote control) is created.

```
private Antenna ant;
```

No actual `Antenna` object is made on the heap unless or until the reference variable is assigned a new `Antenna` object.

```
private Antenna ant = new Antenna();
```

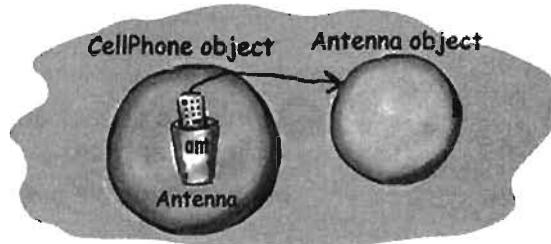


Object with two primitive instance variables.
Space for the variables lives in the object



Object with one non-primitive instance variable—a reference to an `Antenna` object, but no actual `Antenna` object. This is what you get if you declare the variable but don't initialize it with an actual `Antenna` object.

```
public class CellPhone {
    private Antenna ant;
}
```



Object with one non-primitive instance variable, and the `Antenna` variable is assigned a new `Antenna` object

```
public class CellPhone {
    private Antenna ant = new Antenna();
}
```

object creation

The miracle of object creation

Now that you know where variables and objects live, we can dive into the mysterious world of object creation. Remember the three steps of object declaration and assignment: declare a reference variable, create an object, and assign the object to the reference.

But until now, step two—where a miracle occurs and the new object is “born”—has remained a Big Mystery. Prepare to learn the facts of object life. *Hope you’re not squeamish.*

Review the 3 steps of object declaration, creation and assignment:

Make a new reference variable of a class or interface type.

1 Declare a reference variable

Duck myDuck = new Duck();

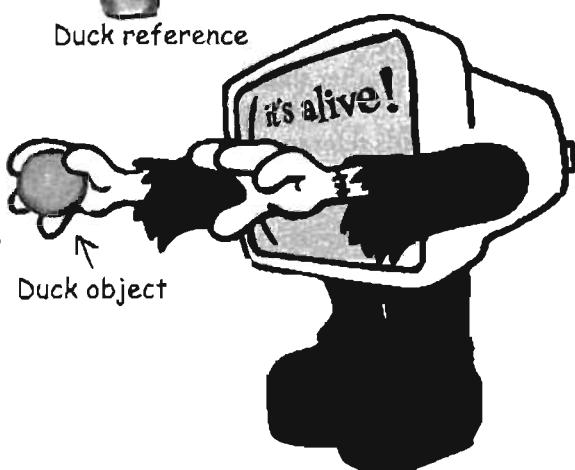


Duck reference

A miracle occurs here.

2 Create an object

Duck myDuck = new Duck();

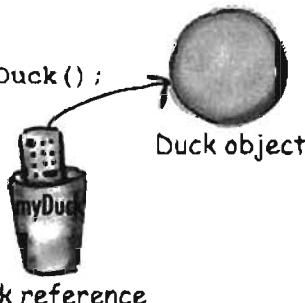


Duck object

Assign the new object to the reference.

3 Link the object and the reference

Duck myDuck = new Duck();



Duck object

Are we calling a method named Duck()?**Because it sure looks like it.**

Duck myDuck = new Duck();

It looks like we're calling
a method named Duck(),
because of the parentheses.

No.**We're calling the Duck constructor.**

A constructor *does* look and feel a lot like a method, but it's not a method. It's got the code that runs when you say `new`. In other words, *the code that runs when you instantiate an object*.

The only way to invoke a constructor is with the keyword `new` followed by the class name. The JVM finds that class and invokes the constructor in that class. (OK, technically this isn't the *only* way to invoke a constructor. But it's the only way to do it from *outside* a constructor. You *can* call a constructor from within another constructor, with restrictions, but we'll get into all that later in the chapter.)

But where is the constructor?**If we didn't write it, who did?**

A CONSTRUCTOR has the code that runs when you instantiate an object. In other words, the code that runs when you say `new` on a class type.

Every class you create has a constructor, even if you don't write it yourself.

You can write a constructor for your class (we're about to do that), but if you don't, *the compiler writes one for you!*

Here's what the compiler's default constructor looks like:

```
public Duck() {  
}
```

Notice something missing? How is this different from a method?

Where's the return type?
If this were a method,
you'd need a return type
between "public" and
"Duck()".

```
public Duck() {  
    // constructor code goes here  
}
```

Its name is the same as the
class name. That's mandatory.

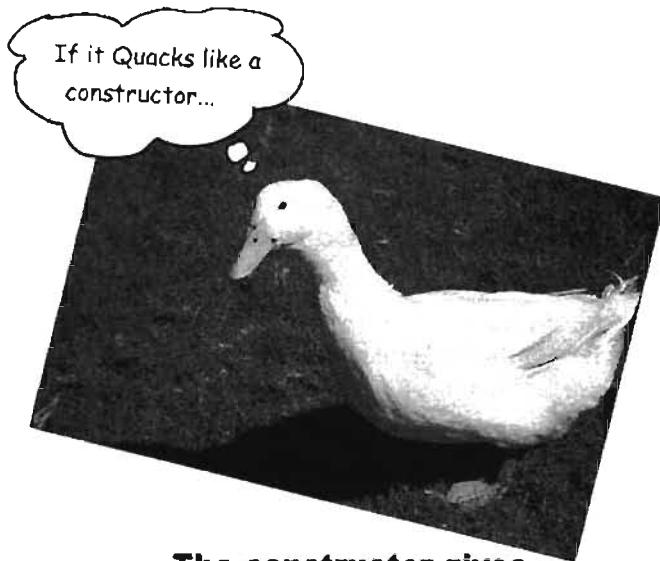
constructing a new Duck

Construct a Duck

The key feature of a constructor is that it runs *before* the object can be assigned to a reference. That means you get a chance to step in and do things to get the object ready for use. In other words, before anyone can use the remote control for an object, the object has a chance to help construct itself. In our Duck constructor, we're not doing anything useful, but it still demonstrates the sequence of events.

```
public class Duck {  
  
    public Duck() {  
        System.out.println("Quack");  
    }  
}
```

↑
Constructor code.



The constructor gives you a chance to step into the middle of `new`.

```
public class UseADuck {  
  
    public static void main (String[] args) {  
        Duck d = new Duck();  
    }  
}
```

↑
This calls the Duck constructor.

```
File Edit Window Help Quack  
% java UseADuck  
Quack
```



Sharpen your pencil

A constructor lets you jump into the middle of the object creation step—into the middle of `new`. Can you imagine conditions where that would be useful? Which of these might be useful in a Car class constructor, if the Car is part of a Racing Game? Check off the ones that you came up with a scenario for.

- Increment a counter to track how many objects of this class type have been made.
- Assign runtime-specific state (data about what's happening NOW).
- Assign values to the object's important instance variables.
- Get and save a reference to the object that's *creating* the new object.
- Add the object to an ArrayList.
- Create HAS-A objects.
- _____ (your idea here)

Initializing the state of a new Duck

Most people use constructors to initialize the state of an object. In other words, to make and assign values to the object's instance variables.

```
public Duck() {
    size = 34;
}
```

That's all well and good when the Duck class *developer* knows how big the Duck object should be. But what if we want the programmer who is *using* Duck to decide how big a particular Duck should be?

Imagine the Duck has a size instance variable, and you want the programmer using your Duck class to set the size of the new Duck. How could you do it?

Well, you could add a setSize() setter method to the class. But that leaves the Duck temporarily without a size*, and forces the Duck user to write *two* statements—one to create the Duck, and one to call the setSize() method. The code below uses a setter method to set the initial size of the new Duck.

```
public class Duck {
    int size; ← instance variable

    public Duck() {
        System.out.println("Quack"); ← constructor
    }

    public void setSize(int newSize) { ← setter method
        size = newSize;
    }
}
```

```
public class UseADuck {

    public static void main (String[] args) {
        Duck d = new Duck();
        d.setSize(42); ← There's a bad thing here. The Duck is alive at
                        this point in the code, but without a size!*
    }
}
```

*Instance variables do have a default value. 0 or 0.0 for numeric primitives, false for booleans, and null for references.

there are no
Dumb Questions

Q: Why do you need to write a constructor if the compiler writes one for you?

A: If you need code to help initialize your object and get it ready for use, you'll have to write your own constructor. You might, for example, be dependent on input from the user before you can finish making the object ready. There's another reason you might have to write a constructor, even if you don't need any constructor code yourself. It has to do with your superclass constructor, and we'll talk about that in a few minutes.

Q: How can you tell a constructor from a method? Can you also have a method that's the same name as the class?

A: Java lets you declare a method with the same name as your class. That doesn't make it a constructor, though. The thing that separates a method from a constructor is the return type. Methods *must* have a return type, but constructors cannot have a return type.

Q: Are constructors inherited? If you don't provide a constructor but your superclass does, do you get the superclass constructor instead of the default?

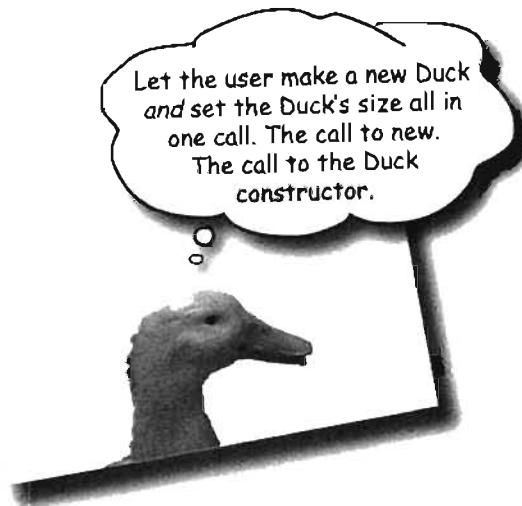
A: Nope. Constructors are not inherited. We'll look at that in just a few pages.

initializing object state

Using the constructor to initialize important Duck state*

If an object shouldn't be used until one or more parts of its state (instance variables) have been initialized, don't let anyone get ahold of a Duck object until you're finished initializing! It's usually way too risky to let someone make—and get a reference to—a new Duck object that isn't quite ready for use until that someone turns around and calls the `setSize()` method. How will the Duck-user even *know* that he's required to call the setter method after making the new Duck?

The best place to put initialization code is in the constructor. And all you need to do is make a constructor with arguments.



```
public class Duck {  
    int size;  
  
    public Duck(int duckSize) {  
        System.out.println("Quack");  
        size = duckSize; ← Use the argument value to set  
        System.out.println("size is " + size);  
    }  
}
```

```
public class UseADuck {  
  
    public static void main (String[] args) {  
        Duck d = new Duck(42); ← Pass a value to the  
        } ← constructor.  
    }  
  
    This time there's only one statement. We make the new Duck and set its size in one statement.
```

File Edit Window Help Hank
% java UseADuck
Quack
size is 42

*Not to imply that not all Duck state is not unimportant.

Make it easy to make a Duck

Be sure you have a no-arg constructor

What happens if the Duck constructor takes an argument? Think about it. On the previous page, there's only *one* Duck constructor—and it takes an int argument for the *size* of the Duck. That might not be a big problem, but it does make it harder for a programmer to create a new Duck object, especially if the programmer doesn't *know* what the size of a Duck should be. Wouldn't it be helpful to have a default size for a Duck, so that if the user doesn't know an appropriate size, he can still make a Duck that works?

Imagine that you want Duck users to have TWO options for making a Duck—one where they supply the Duck size (as the constructor argument) and one where they don't specify a size and thus get your default Duck size.

You can't do this cleanly with just a single constructor. Remember, if a method (or constructor—same rules) has a parameter, you *must* pass an appropriate argument when you invoke that method or constructor. You can't just say, "If someone doesn't pass anything to the constructor, then use the default size", because they won't even be able to compile without sending an int argument to the constructor call. You could do something clunkly like this:

```
public class Duck {
    int size;

    public Duck(int newSize) {
        if (newSize == 0) { ←
            size = 27;
        } else {
            size = newSize;
        }
    }
}
```

If the parameter value is zero, give the new Duck a default size; otherwise use the parameter value for the size. NOT a very good solution.

But that means the programmer making a new Duck object has to *know* that passing a "0" is the protocol for getting the default Duck size. Pretty ugly. What if the other programmer doesn't know that? Or what if he really *does* want a zero-size Duck? (Assuming a zero-sized Duck is allowed. If you don't want zero-sized Duck objects, put validation code in the constructor to prevent it.) The point is, it might not always be possible to distinguish between a genuine "I want zero for the size" constructor argument and a "I'm sending zero so you'll give me the default size, whatever that is" constructor argument.

You really want TWO ways to make a new Duck:

```
public class Duck2 {
    int size;

    public Duck2() {
        // supply default size
        size = 27;
    }

    public Duck2(int duckSize) {
        // use duckSize parameter
        size = duckSize;
    }
}
```

To make a Duck when you know the size:

Duck2 d = new Duck2(15);

To make a Duck when you do not know the size:

Duck2 d2 = new Duck2();

So this two-options-to-make-a-Duck idea needs two constructors. One that takes an int and one that doesn't. **If you have more than one constructor in a class, it means you have Overloaded constructors.**

Doesn't the compiler always make a no-arg constructor for you? No!

You might think that if you write *only* a constructor with arguments, the compiler will see that you don't have a no-arg constructor, and stick one in for you. But that's not how it works. The compiler gets involved with constructor-making *only if you don't say anything at all about constructors.*

If you write a constructor that takes arguments, and you still want a no-arg constructor, you'll have to build the no-arg constructor yourself!

As soon as you provide a constructor, ANY kind of constructor, the compiler backs off and says, "OK Buddy, looks like you're in charge of constructors now."

If you have more than one constructor in a class, the constructors **MUST have different argument lists.**

The argument list includes the order and types of the arguments. As long as they're different, you can have more than one constructor. You can do this with methods as well, but we'll get to that in another chapter.



OK, let's see here... "You have the right to your own constructor." Makes sense.

"If you cannot afford a constructor, one will be provided for you by the compiler." Good to know.

Overloaded constructors means you have more than one constructor in your class.

To compile, each constructor must have a different argument list!

The class below is legal because all four constructors have different argument lists. If you had two constructors that took only an int, for example, the class wouldn't compile. What you name the parameter variable doesn't count. It's the variable type (int, Dog, etc.) and order that matters. You can have two constructors that have identical types, as long as the order is different. A constructor that takes a String followed by an int, is not the same as one that takes an int followed by a String.

Four different constructors
means four different ways to
make a new mushroom.



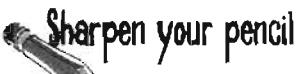
these two have the
same args, but in
different order, so
it's OK

```
public class Mushroom {
    public Mushroom(int size) {} ← when you know the size, but you
    public Mushroom() {} ← when you don't know anything
    public Mushroom(boolean isMagic) {} ← when you know if it's magic or not,
                                         but don't know the size
    {public Mushroom(boolean isMagic, int size) {} } ← when you know
    {public Mushroom(int size, boolean isMagic) {} } ← whether or not it's
                                                       magic, AND you know
                                                       the size as well
}
```

BULLET POINTS

- ▶ Instance variables live within the object they belong to, on the Heap.
- ▶ If the instance variable is a reference to an object, both the reference and the object it refers to are on the Heap.
- ▶ A constructor is the code that runs when you say `new` on a class type.
- ▶ A constructor must have the same name as the class, and must not have a return type.
- ▶ You can use a constructor to initialize the state (i.e. the instance variables) of the object being constructed.
- ▶ If you don't put a constructor in your class, the compiler will put in a default constructor.
- ▶ The default constructor is always a no-arg constructor.
- ▶ If you put a constructor—any constructor—in your class, the compiler will not build the default constructor.
- ▶ If you want a no-arg constructor, and you've already put in a constructor with arguments, you'll have to build the no-arg constructor yourself.
- ▶ Always provide a no-arg constructor if you can, to make it easy for programmers to make a working object. Supply default values.
- ▶ Overloaded constructors means you have more than one constructor in your class.
- ▶ Overloaded constructors must have different argument lists.
- ▶ You cannot have two constructors with the same argument lists. An argument list includes the order and/or type of arguments.
- ▶ Instance variables are assigned a default value, even when you don't explicitly assign one. The default values are 0/0/false for primitives, and null for references.

overloaded constructors



Match the new Duck() call with the constructor that runs when that Duck is instantiated. We did the easy one to get you started.

```
public class TestDuck {  
  
    public static void main(String[] args){  
  
        int weight = 8;  
        float density = 2.3F;  
        String name = "Donald";  
        long[] feathers = {1,2,3,4,5,6};  
        boolean canFly = true;  
        int airspeed = 22;  
  
        Duck[] d = new Duck[7];  
  
        d[0] = new Duck();  
  
        d[1] = new Duck(density, weight);  
  
        d[2] = new Duck(name, feathers);  
  
        d[3] = new Duck(canFly);  
  
        d[4] = new Duck(3.3F, airspeed);  
  
        d[5] = new Duck(false);  
  
        d[6] = new Duck(airspeed, density);  
    }  
}
```

```
class Duck {  
  
    int pounds = 6;  
    float floatability = 2.1F;  
    String name = "Generic";  
    long[] feathers = {1,2,3,4,5,6,7};  
    boolean canFly = true;  
    int maxSpeed = 25;  
  
    public Duck() {  
        System.out.println("type 1 duck");  
    }  
  
    public Duck(boolean fly) {  
        canFly = fly;  
        System.out.println("type 2 duck");  
    }  
  
    public Duck(String n, long[] f) {  
        name = n;  
        feathers = f;  
        System.out.println("type 3 duck");  
    }  
  
    public Duck(int w, float f) {  
        pounds = w;  
        floatability = f;  
        System.out.println("type 4 duck");  
    }  
  
    public Duck(float density, int max) {  
        floatability = density;  
        maxSpeed = max;  
        System.out.println("type 5 duck");  
    }  
}
```

Q: Earlier you said that it's good to have a no-argument constructor so that if people call the no-arg constructor, we can supply default values for the "missing" arguments. But aren't there times when it's impossible to come up with defaults? Are there times when you should not have a no-arg constructor in your class?

A: You're right. There are times when a no-arg constructor doesn't make sense. You'll see this in the Java API—some classes don't have a no-arg constructor. The Color class, for example, represents a... color. Color objects are used to, for example, set or change the color of a screen font or GUI button. When you make a Color instance, that instance is of a particular color (you know, Death-by-Chocolate Brown, Blue-Screen-of-Death Blue, Scandalous Red, etc.). If you make a Color object, you must specify the color in some way.

```
Color c = new Color(3,45,200);
```

(We're using three ints for RGB values here. We'll get into using Color later, in the Swing chapters.) Otherwise, what would you get? The Java API programmers could have decided that if you call a no-arg Color constructor you'll get a lovely shade of mauve. But good taste prevailed. If you try to make a Color without supplying an argument:

```
Color c = new Color();
```

The compiler freaks out because it can't find a matching no-arg constructor in the Color class.

```
File Edit Window Help StopBeingStupid  
cannot resolve symbol  
:constructor Color()  
location: class  
java.awt.Color  
Color c = new Color();  
^  
1 error
```

Nanoreview: four things to remember about constructors

- A constructor is the code that runs when somebody says `new` on a class type

```
Duck d = new Duck();
```

- A constructor must have the same name as the class, and no return type

```
public Duck(int size) { }
```

- If you don't put a constructor in your class, the compiler puts in a default constructor. The default constructor is always a no-arg constructor.

```
public Duck() { }
```

- You can have more than one constructor in your class, as long as the argument lists are different. Having more than one constructor in a class means you have overloaded constructors.

```
public Duck() { }  
public Duck(int size) { }  
public Duck(String name) { }  
public Duck(String name, int size) { }
```

Doing all the Brain Barbells has been shown to produce a 42% increase in neuron size. And you know what they say, "Big neurons..."



What about superclasses?

**When you make a Dog,
should the Canine
constructor run too?**

**If the superclass is abstract,
should it even have a
constructor?**

We'll look at this on the next few pages, so stop now and think about the implications of constructors and superclasses.

~~there are no~~ Dumb Questions

Q: Do constructors have to be `public`?

A: No. Constructors can be `public`, `private`, or `default` (which means no access modifier at all). We'll look more at `default` access in chapter 16 and appendix B.

Q: How could a `private` constructor ever be useful? Nobody could ever call it, so nobody could ever make a new object!

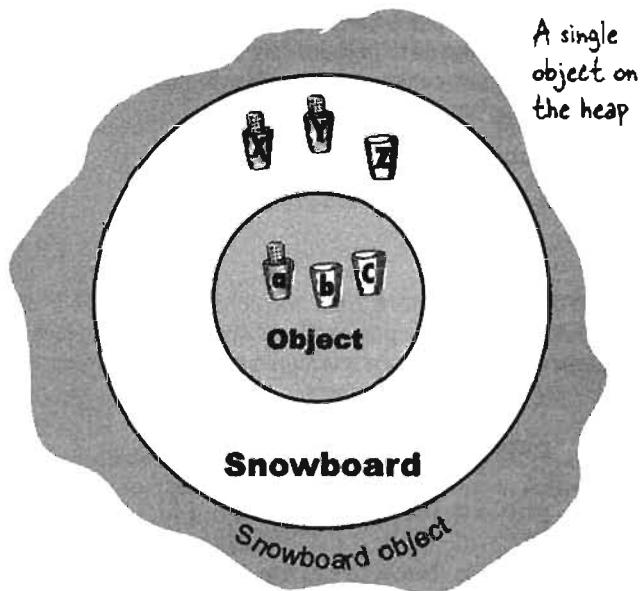
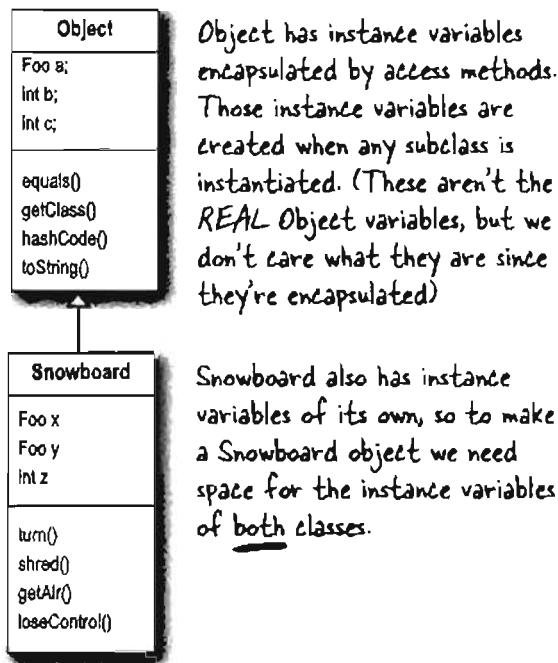
A: But that's not exactly right. Marking something `private` doesn't mean *nobody* can access it, it just means that *nobody outside the class* can access it. Bet you're thinking "Catch 22". Only code from the *same class* as the class-with-private-constructor can make a new object from that class, but without first making an object, how do you ever get to run code from that class in the first place? How do you ever get to anything in that class? *Patience grasshopper*. We'll get there in the next chapter.

space for an object's superclass parts

Wait a minute... we never DID talk about superclasses and inheritance and how that all fits in with constructors.

Here's where it gets fun. Remember from the last chapter, the part where we looked at the Snowboard object wrapping around an inner core representing the Object portion of the Snowboard class? The Big Point there was that every object holds not just its own declared instance variables, but also *everything from its superclasses* (which, at a minimum, means class Object, since *every* class extends Object).

So when an object is created (because somebody said `new`; there is *no other way* to create an object other than someone, somewhere saying `new` on the class type), the object gets space for *all* the instance variables, from all the way up the inheritance tree. Think about it for a moment... a superclass might have setter methods encapsulating a private variable. But that variable has to live *somewhere*. When an object is created, it's almost as though *multiple* objects materialize—the object being new'd and one object per each superclass. Conceptually, though, it's much better to think of it like the picture below, where the object being created has *layers* of itself representing each superclass.



There is only ONE object on the heap here. A Snowboard object. But it contains both the Snowboard parts of itself and the Object parts of itself. All instance variables from both classes have to be here.

The role of superclass constructors in an object's life.

All the constructors in an object's inheritance tree must run when you make a new object.

Let that sink in.

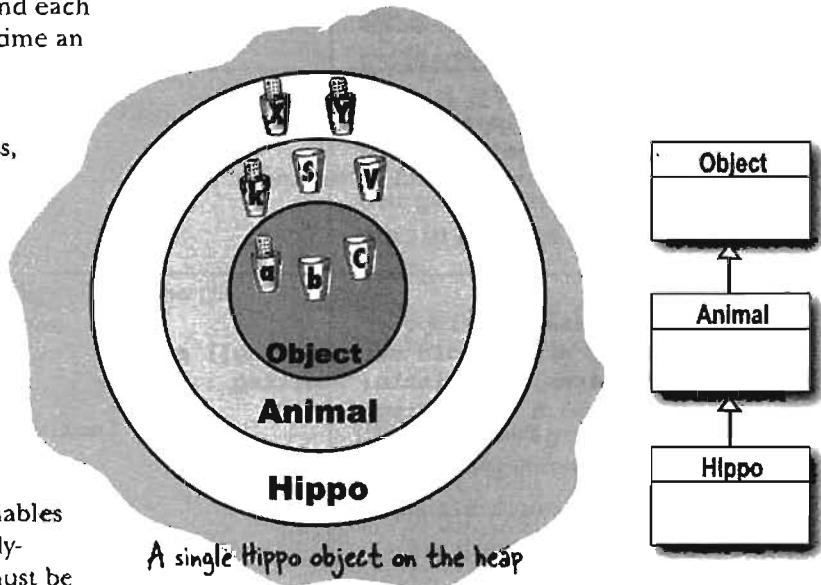
That means every superclass has a constructor (because every class has a constructor), and each constructor up the hierarchy runs at the time an object of a subclass is created.

Saying `new` is a Big Deal. It starts the whole constructor chain reaction. And yes, even abstract classes have constructors. Although you can never say `new` on an abstract class, an abstract class is still a superclass, so its constructor runs when someone makes an instance of a concrete subclass.

The super constructors run to build out the superclass parts of the object. Remember, a subclass might inherit methods that depend on superclass state (in other words, the value of instance variables in the superclass). For an object to be fully-formed, all the superclass parts of itself must be fully-formed, and that's why the super constructor *must* run. All instance variables from every class in the inheritance tree have to be declared and initialized. Even if `Animal` has instance variables that `Hippo` doesn't inherit (if the variables are private, for example), the `Hippo` still depends on the `Animal` methods that *use* those variables.

When a constructor runs, it immediately calls its superclass constructor, all the way up the chain until you get to the class `Object` constructor.

On the next few pages, you'll learn how superclass constructors are called, and how you can call them yourself. You'll also learn what to do if your superclass constructor has arguments!



A new **Hippo** object also **IS-A Animal** and **IS-A Object**. If you want to make a **Hippo**, you must also make the **Animal** and **Object** parts of the **Hippo**.

This all happens in a process called **Constructor Chaining**.

Making a Hippo means making the Animal and Object parts too...

```
public class Animal {
    public Animal() {
        System.out.println("Making an Animal");
    }
}

public class Hippo extends Animal {
    public Hippo() {
        System.out.println("Making a Hippo");
    }
}

public class TestHippo {
    public static void main (String[] args) {
        System.out.println("Starting...");
        Hippo h = new Hippo();
    }
}
```

Sharpen your pencil

What's the real output? Given the code on the left, what prints out when you run TestHippo? A or B? (the answer is at the bottom of the page)

A

```
File Edit Window Help Swear
% java TestHippo
Starting...
Making an Animal
Making a Hippo
```

B

```
File Edit Window Help Swear
% java TestHippo
Starting...
Making a Hippo
Making an Animal
```

- 1 Code from another class says `new Hippo()` and the `Hippo()` constructor goes into a stack frame at the top of the stack.



- 2 `Hippo()` invokes the superclass constructor which pushes the `Animal()` constructor onto the top of the stack.



- 3 `Animal()` invokes the superclass constructor which pushes the `Object()` constructor onto the top of the stack, since `Object` is the superclass of `Animal`.



- 4 `Object()` completes, and its stack frame is popped off the stack. Execution goes back to the `Animal()` constructor, and picks up at the line following `Animal`'s call to its superclass constructor.



The first one, A. The `Hippo()` constructor is invoked first, but it's the `Animal` constructor that finishes first.

How do you invoke a superclass constructor?

You might think that somewhere in, say, a Duck constructor, if Duck extends Animal you'd call Animal(). But that's not how it works:

```
public class Duck extends Animal {
    int size;

    public Duck(int newSize) {
        BAD! → Animal(); ← NO! This is not legal!
        size = newSize;
    }
}
```

The only way to call a super constructor is by calling *super()*. That's right—*super()* calls the *super constructor*.

What are the odds?

```
public class Duck extends Animal {
    int size;

    public Duck(int newSize) {
        super(); ← you just say super()
        size = newSize;
    }
}
```

A call to *super()* in your constructor puts the superclass constructor on the top of the Stack. And what do you think that superclass constructor does? *Calls its superclass constructor*. And so it goes until the Object constructor is on the top of the Stack. Once *Object()* finishes, it's popped off the Stack and the next thing down the Stack (the subclass constructor that called *Object()*) is now on top. *That constructor* finishes and so it goes until the original constructor is on the top of the Stack, where *it can now finish*.

And how is it that we've gotten away without doing it?

You probably figured that out.

Our good friend the compiler puts in a call to *super()* if you don't.

So the compiler gets involved in constructor-making in two ways:

① If you don't provide a constructor

The compiler puts one in that looks like:

```
public ClassName() {
    super();
}
```

② If you do provide a constructor but you do not put in the call to *super()*

The compiler will put a call to *super()* in each of your overloaded constructors.* The compiler-supplied call looks like:

```
super();
```

It always looks like that. The compiler-inserted call to *super()* is always a no-arg call. If the superclass has overloaded constructors, only the no-arg one is called.

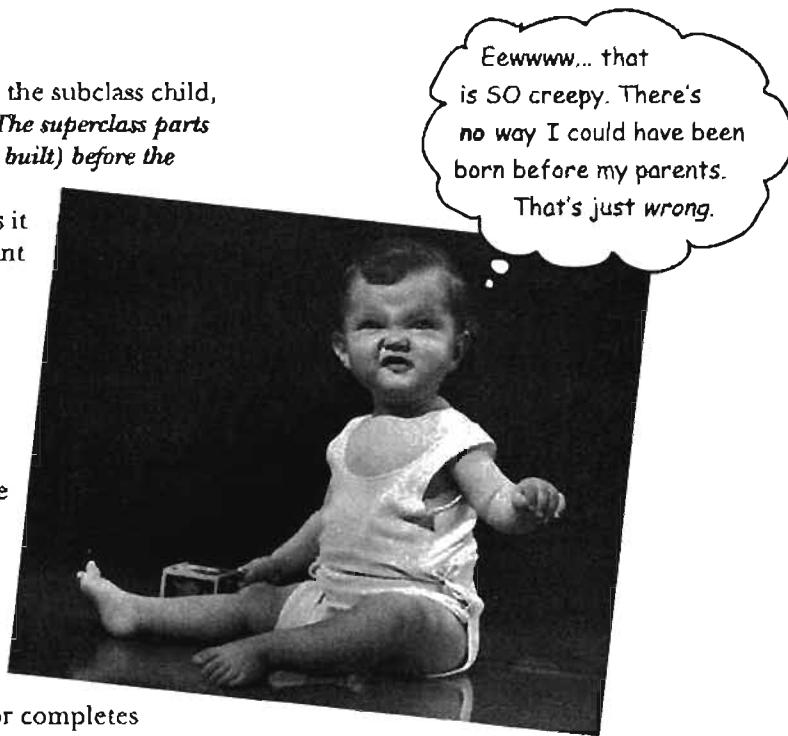
*Unless the constructor calls another overloaded constructor (you'll see that in a few pages).

Can the child exist before the parents?

If you think of a superclass as the parent to the subclass child, you can figure out which has to exist first. *The superclass parts of an object have to be fully-formed (completely built) before the subclass parts can be constructed.* Remember, the subclass object might depend on things it inherits from the superclass, so it's important that those inherited things be finished. No way around it. The superclass constructor must finish before its subclass constructor.

Look at the Stack series on page 248 again, and you can see that while the Hippo constructor is the *first* to be invoked (it's the first thing on the Stack), it's the *last* one to complete! Each subclass constructor immediately invokes its own superclass constructor, until the Object constructor is on the top of the Stack. Then Object's constructor completes and we bounce back down the Stack to Animal's constructor. Only after Animal's constructor completes do we finally come back down to finish the rest of the Hippo constructor. For that reason:

The call to super() must be the first statement in each constructor!



Possible constructors for class Boop	
<input checked="" type="checkbox"/> public Boop() { super(); ← }	<input checked="" type="checkbox"/> public Boop() { } ← These are OK because the compiler will put a call to super() in as the first statement
<input checked="" type="checkbox"/> public Boop(int i) { super(); ← size = i; }	<input checked="" type="checkbox"/> public Boop(int i) { size = i; ← }
	<input type="checkbox"/> public Boop(int i) { size = i; super(); ← BAD!! This won't compile! You can't explicitly put anything else. }

*There's an exception to this rule; you'll learn it on page 252.

Superclass constructors with arguments

What if the superclass constructor has arguments? Can you pass something in to the `super()` call? Of course. If you couldn't, you'd never be able to extend a class that didn't have a no-arg constructor. Imagine this scenario: all animals have a name. There's a `getName()` method in class `Animal` that returns the value of the `name` instance variable. The instance variable is marked private, but the subclass (in this case, `Hippo`) inherits the `getName()` method. So here's the problem: `Hippo` has a `getName()` method (through inheritance), but does not have the `name` instance variable. `Hippo` has to depend on the `Animal` part of himself to keep the `name` instance variable, and return it when someone calls `getName()` on a `Hippo` object. But... how does the `Animal` part get the name? The only reference `Hippo` has to the `Animal` part of himself is through `super()`, so that's the place where `Hippo` sends the `Hippo`'s name up to the `Animal` part of himself, so that the `Animal` part can store it in the private `name` instance variable.

```
public abstract class Animal {
    private String name;   ← All animals (including
                           subclasses) have a name

    public String getName() { ← A getter method that
        return name;       Hippo inherits
    }

    public Animal(String theName) {
        name = theName;   ← The constructor that
    }                      takes the name and assigns
                           it the name instance
                           variable

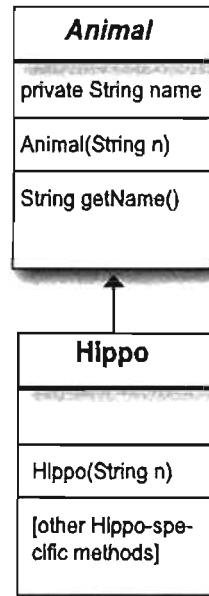
}

public class Hippo extends Animal {

    public Hippo(String name) {
        super(name);   ← Hippo constructor takes a name
    }
}                      ← it sends the name up the Stack to
                           the Animal constructor

}

public class MakeHippo {
    public static void main(String[] args) { Make a Hippo, passing the
        Hippo h = new Hippo("Buffy");   ← name "Buffy" to the Hippo
        System.out.println(h.getName());  constructor. Then call the
    }                                Hippo's inherited getName()
}
```



The Animal part of me needs to know my name, so I take a name in my own Hippo constructor, then pass the name to super()



```

File Edit Window Help Hide
% java MakeHippo
Buffy

```

calling overloaded constructors

Invoking one overloaded constructor from another

What if you have overloaded constructors that, with the exception of handling different argument types, all do the same thing? You know that you don't want *duplicate* code sitting in each of the constructors (pain to maintain, etc.), so you'd like to put the bulk of the constructor code (including the call to `super()`) in only *one* of the overloaded constructors. You want whichever constructor is first invoked to call The Real Constructor and let The Real Constructor finish the job of construction. It's simple: just say `this()`. Or `this(aString)`. Or `this(27, x)`. In other words, just imagine that the keyword `this` is a reference to the **current object**.

You can say `this()` only within a constructor, and it must be the first statement in the constructor!

But that's a problem, isn't it? Earlier we said that `super()` must be the first statement in the constructor. Well, that means you get a choice.

Every constructor can have a call to super() or this(), but never both!

```
class Mini extends Car {  
  
    Color color;  
  
    public Mini() {  
        this(Color.Red); ←  
    }  
  
    public Mini(Color c) {  
        super("Mini"); ←  
        color = c;  
        // more initialization  
    }  
  
    public Mini(int size) {  
        this(Color.Red); ←  
        super(size); ←  
    }  
}
```

The no-arg constructor supplies a default Color and calls the overloaded Real Constructor (the one that calls `super()`).

This is The Real Constructor that does The Real Work of initializing the object (including the call to `super()`)

Won't work!! Can't have `super()` and `this()` in the same constructor, because they each must be the first statement!

Use `this()` to call a constructor from another overloaded constructor in the same class.

The call to `this()` can be used only in a constructor, and must be the first statement in a constructor.

A constructor can have a call to `super()` OR `this()`, but never both!

```
File Edit Window Help Drva  
javac Mini.java  
Mini.java:16: call to super must  
be first statement in constructor  
        super();  
               ^
```

Sharpen your pencil

Some of the constructors in the SonOfBoo class will not compile. See if you can recognize which constructors are not legal. Match the compiler errors with the SonOfBoo constructors that caused them, by drawing a line from the compiler error to the "bad" constructor.

```
public class Boo {
    public Boo(int i) { }
    public Boo(String s) { }
    public Boo(String s, int i) { }
}
```

```
class SonOfBoo extends Boo {

    public SonOfBoo() {
        super("boo");
    }

    public SonOfBoo(int i) {
        super("Fred");
    }

    public SonOfBoo(String s) {
        super(42);
    }

    public SonOfBoo(int i, String s) {
    }

    public SonOfBoo(String a, String b, String c) {
        super(a,b);
    }

    public SonOfBoo(int i, int j) {
        super("man", j);
    }

    public SonOfBoo(int i, int x, int y) {
        super(i, "star");
    }
}
```



```
File Edit Window Help
%javac SonOfBoo.java
cannot resolve symbol
symbol : constructor Boo
(java.lang.String,java.lang.String)
```

```
File Edit Window Help Yadayadaya
%javac SonOfBoo.java
cannot resolve symbol
symbol : constructor Boo
(int,java.lang.String)
```

```
File Edit Window Help ImNotListening
%javac SonOfBoo.java
cannot resolve symbol
symbol:constructor Boo()
```

object lifespan

Now we know how an object is born, but how long does an object live?

An object's life depends entirely on the life of references referring to it. If the reference is considered "alive", the object is still alive on the Heap. If the reference dies (and we'll look at what that means in just a moment), the object will die.

So if an object's life depends on the reference variable's life, how long does a *variable* live?

That depends on whether the variable is a *local* variable or an *instance* variable. The code below shows the life of a local variable. In the example, the variable is a primitive, but variable lifetime is the same whether it's a primitive or reference variable.

```
public class TestLifeOne {  
  
    public void read() {  
        int s = 42; ← 's' is scoped to the  
        sleep(); method, so it can't be used  
    }  
  
    public void sleep() {  
        s = 7;  
    } ↑ BAD!! Not legal to  
    use 's' here!  
  
    sleep()  
    read() s  
  
    The variable 's' is alive, but in scope only within the  
    read() method. When sleep() completes and read() is  
    on top of the stack and running again, read() can  
    still see 's'. When read() completes and is popped off  
    the stack, 's' is dead. Pushing up digital daisies.  
}
```

❶ A local variable lives only within the method that declared the variable.

```
public void read() {  
    int s = 42;  
    // 's' can be used only  
    // within this method.  
    // When this method ends,  
    // 's' disappears completely.  
}
```

Variable 's' can be used only within the *read()* method. In other words, the *variable is in scope only within its own method*. No other code in the class (or any other class) can see 's'.

❷ An instance variable lives as long as the object does. If the object is still alive, so are its instance variables.

```
public class Life {  
    int size;  
  
    public void setSize(int s) {  
        size = s;  
        // 's' disappears at the  
        // end of this method,  
        // but 'size' can be used  
        // anywhere in the class  
    }  
}
```

Variable 's' (this time a method parameter) is in scope only within the *setSize()* method. But instance variable *size* is scoped to the life of the *object* as opposed to the life of the *method*.

The difference between **life** and **scope** for local variables:

Life

A local variable is *alive* as long as its Stack frame is on the Stack. In other words, *until the method completes*.

Scope

A local variable is in *scope* only within the method in which the variable was declared. When its own method calls another, the variable is alive, but not in scope until its method resumes. *You can use a variable only when it is in scope.*

```
public void doStuff() {
    boolean b = true;
    go(4);
}

public void go(int x) {
    int z = x + 24;
    crazy();
    // imagine more code here
}

public void crazy() {
    char c = 'a';
}
```



- 1 doStuff() goes on the Stack. Variable 'b' is alive and in scope.



- 2 go() pops onto the Stack. 'x' and 'z' are alive and in scope, and 'b' is alive but not in scope.



- 3 crazy() is pushed onto the Stack, with 'c' now alive and in scope. The other three variables are alive but out of scope.



- 4 crazy() completes and is popped off the Stack, so 'c' is out of scope and dead. When go() resumes where it left off, 'x' and 'z' are both alive and back in scope. Variable 'b' is still alive but out of scope (until go() completes).

While a local variable is alive, its state persists. As long as method doStuff() is on the Stack, for example, the 'b' variable keeps its value. But the 'b' variable can be used *only* while doStuff()'s Stack frame is at the top. In other words, you can use a local variable *only* while that local variable's method is actually running (as opposed to waiting for higher Stack frames to complete).

What about reference variables?

The rules are the same for primitives and references. A reference variable can be used only when it's in scope, which means you can't use an object's remote control unless you've got a reference variable that's in scope. The *real* question is,

"How does *variable* life affect *object* life?"

An object is alive as long as there are live references to it. If a reference variable goes out of scope but is still alive, the object it *refers* to is still alive on the Heap. And then you have to ask... "What happens when the Stack frame holding the reference gets popped off the Stack at the end of the method?"

If that was the *only* live reference to the object, the object is now abandoned on the Heap. The reference variable disintegrated with the Stack frame, so the abandoned object is now, *officially*, toast. The trick is to know the point at which an object becomes *eligible for garbage collection*.

Once an object is eligible for garbage collection (GC), you don't have to worry about reclaiming the memory that object was using. If your program gets low on memory, GC will destroy some or all of the eligible objects, to keep you from running out of RAM. You can still run out of memory, but *not* before all eligible objects have been hauled off to the dump. Your job is to make sure that you abandon objects (i.e., make them eligible for GC) when you're done with them, so that the garbage collector has something to reclaim. If you hang on to objects, GC can't help you and you run the risk of your program dying a painful out-of-memory death.

An object's life has no value, no meaning, no point, unless somebody has a reference to it.

If you can't get to it, you can't ask it to do anything and it's just a big fat waste of bits.

But if an object is unreachable, the Garbage Collector will figure that out. Sooner or later, that object's goin' down.



An object becomes eligible for GC when its last live reference disappears.

Three ways to get rid of an object's reference:

- ① The reference goes out of scope, permanently

```
void go() {  
    Life z = new Life();  
}
```

reference 'z' dies at
end of method

- ② The reference is assigned another object

```
Life z = new Life();  
z = new Life();
```

the first object is abandoned
when z is 'reprogrammed' to
a new object

- ③ The reference is explicitly set to null

```
Life z = new Life();  
z = null;
```

the first object is abandoned
when z is 'deprogrammed'.

Object-killer #1

Reference goes out of scope, permanently.



```
public class StackRef {
    public void foof() {
        barf();
    }

    public void barf() {
        Duck d = new Duck();
    }
}
```



- ➊ *foof()* is pushed onto the Stack, no variables are declared.

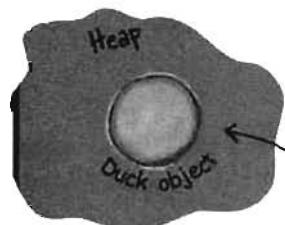


- ➋ *barf()* is pushed onto the Stack, where it declares a reference variable, and creates a new object assigned to that reference. The object is created on the Heap, and the reference is alive and in scope.



The new Duck goes on the Heap, and as long as *barf()* is running, the 'd' reference is alive and in scope, so the Duck is considered alive.

- ➌ *barf()* completes and pops off the Stack. Its frame disintegrates, so 'd' is now dead and gone. Execution returns to *foof()*, but *foof()* can't use 'd'.



Uh-oh. The 'd' variable went away when the *barf()* Stack frame was blown off the stack, so the Duck is abandoned. Garbage-collector bait.

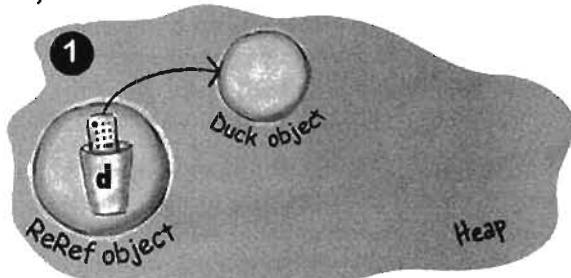
object lifecycle

Object-killer #2

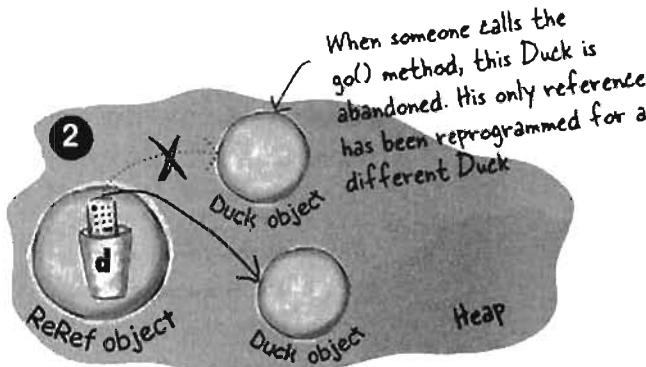
Assign the reference
to another object



```
public class ReRef {  
  
    Duck d = new Duck();  
  
    public void go() {  
        d = new Duck();  
    }  
}
```



The new Duck goes on the Heap, referenced by 'd'. Since 'd' is an instance variable, the Duck will live as long as the ReRef object that instantiated it is alive. Unless...



'd' is assigned a new Duck object, leaving the original (first) Duck object abandoned. That first Duck is now as good as dead.

Dude, all you had to do was reset the reference. Guess they didn't have memory management back then.



Object-killer #3

Explicitly set the reference to null



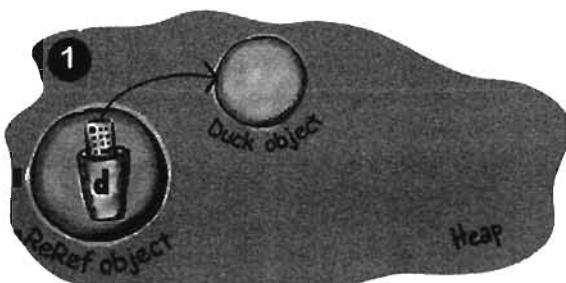
```
public class ReRef {
    Duck d = new Duck();
    public void go() {
        d = null;
    }
}
```

The meaning of null

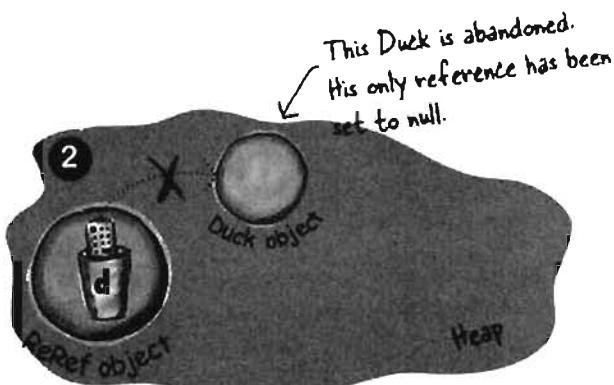
When you set a reference to `null`, you're deprogramming the remote control. In other words, you've got a remote control, but no TV at the other end. A null reference has bits representing 'null' (we don't know or care what those bits are, as long as the JVM knows).

If you have an unprogrammed remote control, in the real world, the buttons don't do anything when you press them. But in Java, you can't press the buttons (i.e. use the dot operator) on a null reference, because the JVM knows (this is a runtime issue, not a compiler error) that you're expecting a bark but there's no Dog there to do it!

If you use the dot operator on a null reference, you'll get a **NullPointerException** at runtime. You'll learn all about Exceptions in the Risky Behavior chapter.

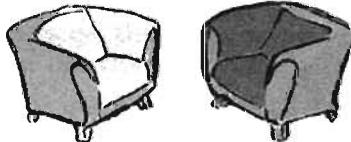


The new Duck goes on the Heap, referenced by 'd'. Since 'd' is an instance variable, the Duck will live as long as the ReRef object that instantiated it is alive. Unless...



'd' is set to null, which is just like having a remote control that isn't programmed to anything. You're not even allowed to use the dot operator on 'd' until it's reprogrammed (assigned an object).

Fireside Chats



Tonight's Talk: **An instance variable and a local variable discuss life and death (with remarkable civility)**

Instance Variable

I'd like to go first, because I tend to be more important to a program than a local variable. I'm there to support an object, usually throughout the object's entire life. After all, what's an object without *state*? And what is state? Values kept in *instance variables*.

No, don't get me wrong, I do understand your role in a method, it's just that your life is so short. So temporary. That's why they call you guys "temporary variables".

My apologies. I understand completely.

I never really thought about it like that. What are you doing while the other methods are running and you're waiting for your frame to be the top of the Stack again?

Local Variable

I appreciate your point of view, and I certainly appreciate the value of object state and all, but I don't want folks to be misled. Local variables are *really* important. To use your phrase, "After all, what's an object without behavior?" And what is behavior? Algorithms in methods. And you can bet your bits there'll be some *local variables* in there to make those algorithms work.

Within the local-variable community, the phrase "temporary variable" is considered derogatory. We prefer "local", "stack", "automatic", or "Scope-challenged".

Anyway, it's true that we don't have a long life, and it's not a particularly *good* life either. First, we're shoved into a Stack frame with all the other local variables. And then, if the method we're part of calls another method, another frame is pushed on top of us. And if *that* method calls *another* method... and so on. Sometimes we have to wait forever for all the other methods on top of the Stack to complete so that our method can run again.

Nothing. Nothing at all. It's like being in stasis—that thing they do to people in science fiction movies when they have to travel long distances. Suspended animation, really. We just sit there on hold. As long as our frame is still there, we're safe and the value we hold is secure, but it's a mixed blessing when our

Instance Variable

We saw an educational video about it once. Looks like a pretty brutal ending. I mean, when that method hits its ending curly brace, the frame is literally *blown* off the Stack! Now *that's* gotta hurt.

I live on the Heap, with the objects. Well, not *with* the objects, actually *in* an object. The object whose state I store. I have to admit life can be pretty luxurious on the Heap. A lot of us feel guilty, especially around the holidays.

OK, hypothetically, yes, if I'm an instance variable of the Collar and the Collar gets GC'd, then the Collar's instance variables would indeed be tossed out like so many pizza boxes. But I was told that this almost never happens.

They let us *drink*?

Local Variable

frame gets to run again. On the one hand, we get to be active again. On the other hand, the clock starts ticking again on our short lives. The more time our method spends running, the closer we get to the end of the method. We *all* know what happens then.

Tell me about it. In computer science they use the term *popped* as in "the frame was popped off the Stack". That makes it sound fun, or maybe like an extreme sport. But, well, you saw the footage. So why don't we talk about you? I know what my little Stack frame looks like, but where do *you* live?

But you don't *always* live as long as the object who declared you, right? Say there's a Dog object with a Collar instance variable. Imagine *you're* an instance variable of the Collar object, maybe a reference to a Buckle or something, sitting there all happy inside the Collar object who's all happy inside the Dog object. But... what happens if the Dog wants a new Collar, or *nulls* out its Collar instance variable? That makes the Collar object eligible for GC. So... if *you're* an instance variable inside the Collar, and the whole Collar is abandoned, what happens to *you*?

And you believed it? That's what they say to keep us motivated and productive. But aren't you forgetting something else? What if you're an instance variable inside an object, and that object is referenced *only* by a *local* variable? If I'm the only reference to the object you're in, when I go, you're coming with me. Like it or not, our fates may be connected. So I say we forget about all this and go get drunk while we still can. Carpe RAM and all that.

exercise: Be the Garbage Collector



BE the Garbage Collector



Which of the lines of code on the right, if added to the class on the left at point A, would cause exactly one additional object to be eligible for the Garbage Collector? (Assume that point A (`//call more methods`) will execute for a long time, giving the Garbage Collector time to do its stuff.)

```
public class GC {  
    public static GC doStuff() {  
        GC newGC = new GC();  
        doStuff2(newGC);  
        return newGC;  
    }  
}
```

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

```
        GC gc1;  
        GC gc2 = new GC();  
        GC gc3 = new GC();  
        GC gc4 = gc3;  
        gc1 = doStuff();  
    }  
    // call more methods  
}
```

```
    public static void doStuff2(GC copyGC) {  
        GC localGC  
    }  
}
```

- 1 copyGC = null;
- 2 gc2 = null;
- 3 newGC = gc3;
- 4 gc1 = null;
- 5 newGC = null;
- 6 gc4 = null;

- 7 gc3 = gc2;
- 8 gc1 = gc4;
- 9 gc3 = null;



Popular Objects

```

class Bees {
    Honey [] beeHA;
}

class Raccoon {
    Kit k;
    Honey rh;
}

class Kit {
    Honey kh;
}

class Bear {
    Honey hunny;
}

public class Honey {
    public static void main(String [] args) {
        Honey honeyPot = new Honey();
        Honey [] ha = {honeyPot, honeyPot, honeyPot, honeyPot};
        Bees bl = new Bees();
        bl.beeRA = ha;
        Bear [] ba = new Bear[5];
        for (int x=0; x < 5; x++) {
            ba[x] = new Bear();
            ba[x].hunny = honeyPot;
        }
        Kit k = new Kit();
        k.kh = honeyPot;
        Raccoon r = new Raccoon(); // Here's a new Raccoon object!
        r.rh = honeyPot; // Here's its reference variable 'r'.
        r.k = k;
        k = null;
    } // end of main
}

```

In this code example, several new objects are created. Your challenge is to find the object that is 'most popular', i.e. the one that has the most reference variables referring to it. Then list how *many* total references there are for that object, and what they are! We'll start by pointing out one of the new objects, and its reference variable.

Good Luck!

puzzle: Five Minute Mystery



Five-Minute Mystery



"We've run the simulation four times, and the main module's temperature consistently drifts out of nominal towards cold", Sarah said, exasperated. "We installed the new temp-bots last week. The readings on the radiator bots, designed to cool the living quarters, seem to be within spec, so we've focused our analysis on the heat retention bots, the bots that help to warm the quarters." Tom sighed, at first it had seemed that nano-technology was going to really put them ahead of schedule. Now, with only five weeks left until launch, some of the orbiter's key life support systems were still not passing the simulation gauntlet.

"What ratios are you simulating?", Tom asked.

"Well if I see where you're going, we already thought of that", Sarah replied. "Mission control will not sign off on critical systems if we run them out of spec. We are required to run the v3 radiator bot's SimUnits in a 2:1 ratio with the v2 radiator's SimUnits", Sarah continued. "Overall, the ratio of retention bots to radiator bots is supposed to run 4:3."

"How's power consumption Sarah?", Tom asked. Sarah paused, "Well that's another thing, power consumption is running higher than anticipated. We've got a team tracking that down too, but because the nanos are wireless it's been hard to isolate the power consumption of the radiators from the retention bots." "Overall power consumption ratios", Sarah continued, "are designed to run 3:2 with the radiators pulling more power from the wireless grid."

"OK Sarah", Tom said "Let's take a look at some of the simulation initiation code. We've got to find this problem, and find it quick!"

```
import java.util.*;
class V2Radiator {
    V2Radiator(ArrayList list) {
        for(int x=0; x<5; x++) {
            list.add(new SimUnit("V2Radiator"));
        }
    }
}

class V3Radiator extends V2Radiator {
    V3Radiator(ArrayList llist) {
        super(llist);
        for(int g=0; g<10; g++) {
            llist.add(new SimUnit("V3Radiator"));
        }
    }
}

class RetentionBot {
    RetentionBot(ArrayList rlist) {
        rlist.add(new SimUnit("Retention"));
    }
}
```

Five-Minute Mystery continued...

```

public class TestLifeSupportSim {
    public static void main(String [] args) {
        ArrayList aList = new ArrayList();
        V2Radiator v2 = new V2Radiator(aList);
        V3Radiator v3 = new V3Radiator(aList);
        for(int z=0; z<20; z++) {
            RetentionBot ret = new RetentionBot(aList);
        }
    }
}

class SimUnit {
    String botType;
    SimUnit(String type) {
        botType = type;
    }
    int powerUse() {
        if ("Retention".equals(botType)) {
            return 2;
        } else {
            return 4;
        }
    }
}

```

Tom gave the code a quick look and a small smile creped across his lips. I think I've found the problem Sarah, and I bet I know by what percentage your power usage readings are off too!

What did Tom suspect? How could he guess the power readings errors, and what few lines of code could you add to help debug this program?



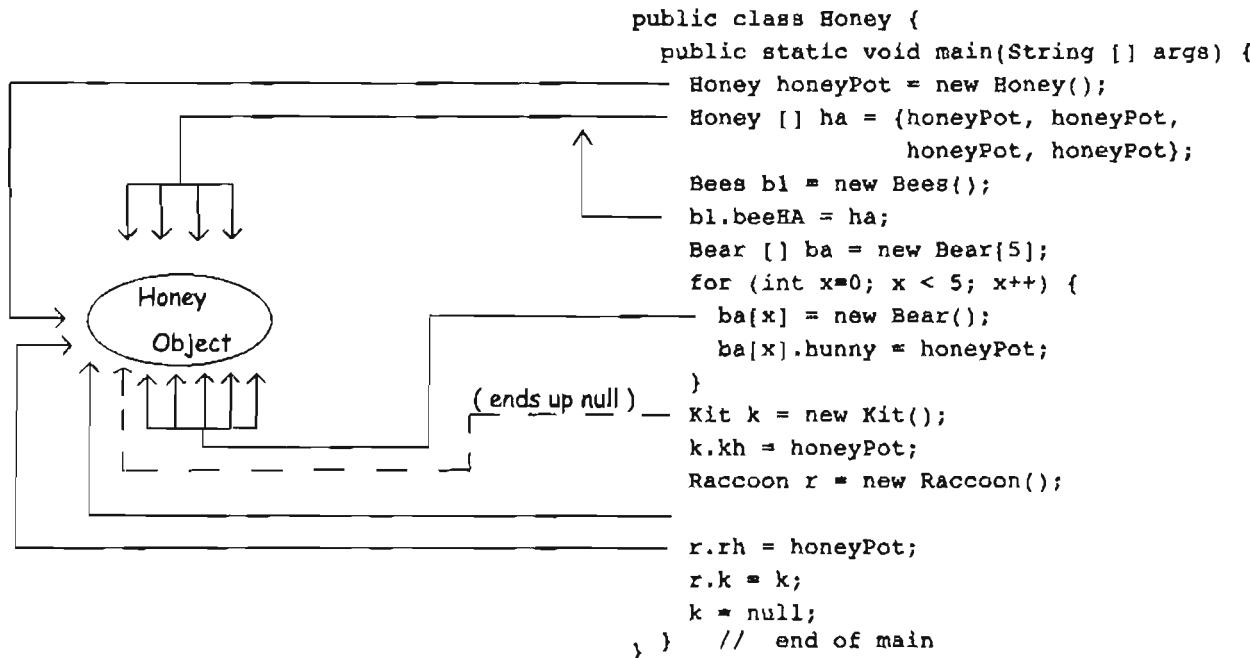
Exercise Solutions

G.C.

- 1 copyGC = null; No - this line attempts to access a variable that is out of scope.
- 2 gc2 = null; OK - gc2 was the only reference variable referring to that object.
- 3 newGC = gc3; No - another out of scope variable.
- 4 gc1 = null; OK - gc1 had the only reference because newGC is out of scope.
- 5 newGC = null; No - newGC is out of scope.
- 6 gc4 = null; No - gc3 is still referring to that object.
- 7 gc3 = gc2; No - gc4 is still referring to that object.
- 8 gc1 = gc4; OK - Reassigning the only reference to that object.
- 9 gc3 = null; No - gc4 is still referring to that object.

Popular Objects

It probably wasn't too hard to figure out that the Honey object first referred to by the honeyPot variable is by far the most 'popular' object in this class. But maybe it was a little trickier to see that all of the variables that point from the code to the Honey object refer to the **same object!** There are a total of 12 active references to this object right before the main() method completes. The k.kh variable is valid for a while, but k gets nulled at the end. Since r.k still refers to the Kit object, r.k.kh (although never explicitly declared), refers to the object!



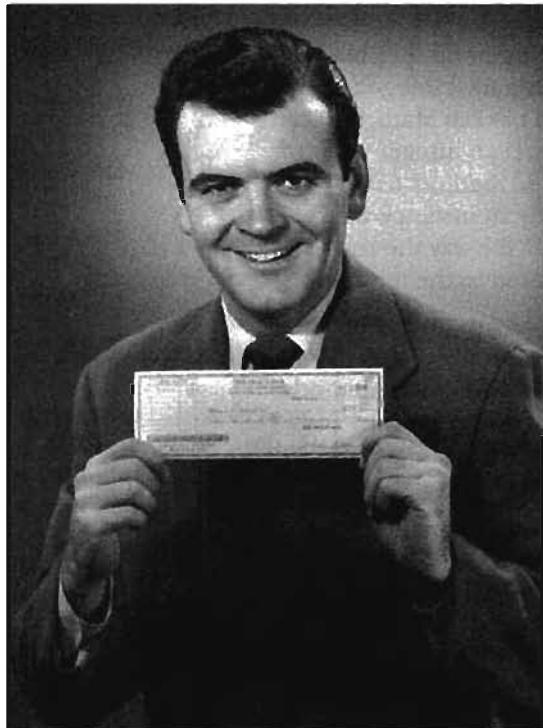


Five-Minute Mystery Solution

Tom noticed that the constructor for the V2Radiator class took an ArrayList. That meant that every time the V3Radiator constructor was called, it passed an ArrayList in its super() call to the V2Radiator constructor. That meant that an extra five V2Radiator SimUnits were created. If Tom was right, total power use would have been 120, not the 100 that Sarah's expected ratios predicted.

Since all the Bot classes create SimUnits, writing a constructor for the SimUnit class, that printed out a line everytime a SimUnit was created, would have quickly highlighted the problem!

Numbers Matter



Do the Math. But there's more to working with numbers than just doing primitive arithmetic. You might want to get the absolute value of a number, or round a number, or find the larger of two numbers. You might want your numbers to print with exactly two decimal places, or you might want to put commas into your large numbers to make them easier to read. And what about working with dates? You might want to print dates in a variety of ways, or even manipulate dates to say things like, "add three weeks to today's date". And what about parsing a String into a number? Or turning a number into a String? You're in luck. The Java API is full of handy number-tweaking methods ready and easy to use. But most of them are **static**, so we'll start by learning what it means for a variable or method to be static, including constants in Java—static *final* variables.

Math methods

MATH methods: as close as you'll ever get to a *global* method

Except there's no global *anything* in Java. But think about this: what if you have a method whose behavior doesn't depend on an instance variable value. Take the round() method in the Math class, for example. It does the same thing every time—rounds a floating point number (the argument to the method) to the nearest integer. Every time. If you had 10,000 instances of class Math, and ran the round(42.2) method, you'd get an integer value of 42. Every time. In other words, the method acts on the argument, but is never affected by an instance variable state. The only value that changes the way the round() method runs is the argument passed to the method!

Doesn't it seem like a waste of perfectly good heap space to make an instance of class Math simply to run the round() method? And what about *other* Math methods like min(), which takes two numerical primitives and returns the smaller of the two. Or max(). Or abs(), which returns the absolute value of a number.

These methods never use instance variable values. In fact the Math class doesn't have any instance variables. So there's nothing to be gained by making an instance of class Math. So guess what? You don't have to. As a matter of fact, you can't.

If you try to make an instance of class Math:

```
Math mathObject = new Math();
```

You'll get this error:

```
File Edit Window Help IwasToldThereWouldBeNoMath
*javac TestMath

TestMath.java:3: Math() has private
access in java.lang.Math
    Math mathObject = new Math();
                           ^
1 error
```

← This error shows that the Math constructor is marked private! That means you can NEVER say 'new' on the Math class to make a new Math object.

Methods in the Math class don't use any instance variable values. And because the methods are 'static', you don't need to have an instance of Math. All you need is the Math class.

```
int x = Math.round(42.2);
int y = Math.min(56,12);
int z = Math.abs(-343);
```



These methods never use instance variables, so their behavior doesn't need to know about a specific object

The difference between regular (non-static) and static methods

Java is object-oriented, but once in a while you have a special case, typically a utility method (like the Math methods), where there is no need to have an instance of the class. The keyword `static` lets a method run *without any instance of the class*. A static method means "behavior not dependent on an instance variable, so no instance/object is required. Just the class."

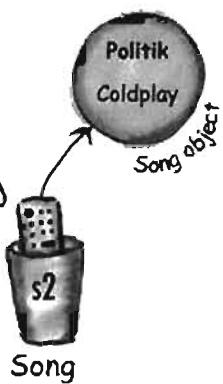
regular (non-static) method

```
public class Song {
    String title; ← Instance variable value affects
                    the behavior of the play()
    public Song(String t) method.
        title = t;
    }
    public void play() {
        SoundPlayer player = new SoundPlayer();
        player.playSound(title);
    }
}
```

Song
title
play()

The current value of the 'title' instance variable is the song that plays when you call `play()`.

two instances
of class Song

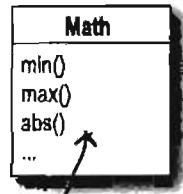


`s2.play();`
Calling `play()` on this reference will cause "Politik" to play.

`s3.play();`
Calling `play()` on this reference will cause "My Way" to play.

static method

```
public [REDACTED] int min(int a, int b){
    //returns the lesser of a and b
}
```



No instance variables.
The method behavior
doesn't change with
instance variable state.

`Math.min(42, 36);`

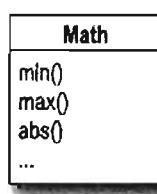
Use the Class name, rather
than a reference variable
name.



NO OBJECTS!!
Absolutely NO OBJECTS
anywhere in this picture!

static methods

Call a static method using a class name



```
Math.min(88,86);
```

Call a non-static method using a reference variable name



```
Song t2 = new Song();  
t2.play();
```

What it means to have a class with static methods.

Often (although not always), a class with static methods is not meant to be instantiated. In Chapter 8 we talked about abstract classes, and how marking a class with the **abstract** modifier makes it impossible for anyone to say 'new' on that class type. In other words, *it's impossible to instantiate an abstract class*.

But you can restrict other code from instantiating a *non-abstract* class by marking the constructor **private**. Remember, a *method* marked private means that only code from within the class can invoke the method. A *constructor* marked private means essentially the same thing—only code from within the class can invoke the constructor. Nobody can say 'new' from *outside* the class. That's how it works with the Math class, for example. The constructor is private, you cannot make a new instance of Math. The compiler knows that your code doesn't have access to that private constructor.

This does *not* mean that a class with one or more static methods should never be instantiated. In fact, every class you put a main() method in is a class with a static method in it!

Typically, you make a main() method so that you can launch or test another class, nearly always by instantiating a class in main, and then invoking a method on that new instance.

So you're free to combine static and non-static methods in a class, although even a single non-static method means there must be *some* way to make an instance of the class. The only ways to get a new object are through 'new' or deserialization (or something called the Java Reflection API that we don't go into). No other way. But exactly *who* says new can be an interesting question, and one we'll look at a little later in this chapter.

Static methods can't use non-static (instance) variables!

Static methods run without knowing about any particular instance of the static method's class. And as you saw on the previous pages, there might not even be any instances of that class. Since a static method is called using the *class* (`Math.random()`) as opposed to an *instance reference* (`t2.play()`), a static method can't refer to any instance variables of the class. The static method doesn't know which instance's variable value to use.

If you try to compile this code:

```
public class Duck {
    private int size;

    public static void main (String[] args) {
        System.out.println("Size of duck is " + size);
    }

    public void setSize(int s) {
        size = s;
    }

    public int getSize() {
        return size;
    }
}
```

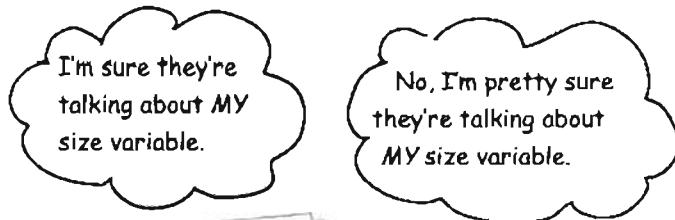
Which Duck?
Whose size?

If there's a Duck on
the heap somewhere, we
don't know about it

You'll get this error:

```
File Edit Window Help Quack
% javac Duck.java
Duck.java:6: non-static variable
size cannot be referenced from a
static context
        System.out.println("Size
of duck is " + size);
               ^
```

If you try to use an instance variable from inside a static method, the compiler thinks, "I don't know which object's instance variable you're talking about!" If you have ten Duck objects on the heap, a static method doesn't know about any of them.



static methods

Static methods can't use non-static methods, either!

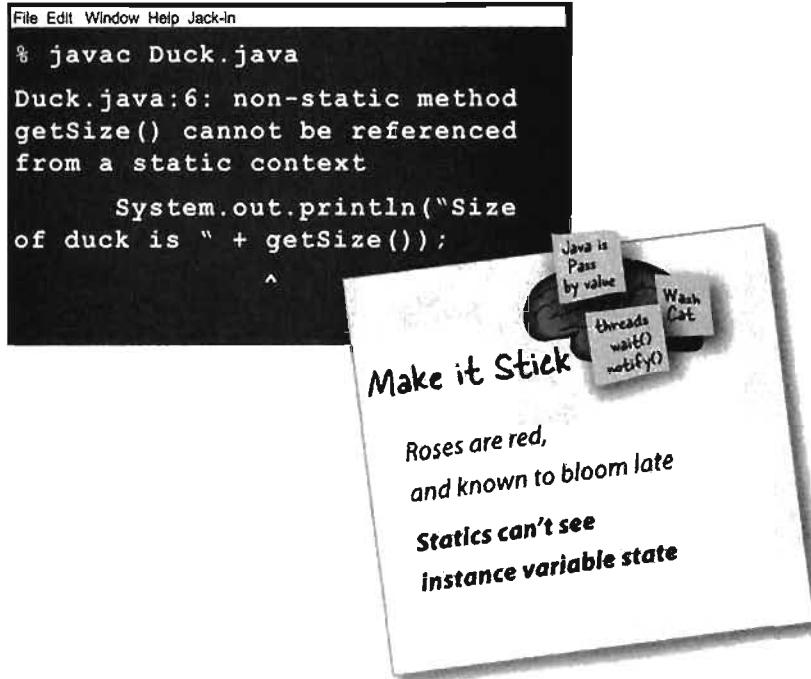
What do non-static methods do? *They usually use instance variable state to affect the behavior of the method.* A getName() method returns the value of the name variable. Whose name? The object used to invoke the getName() method.

This won't compile:

```
public class Duck {  
    private int size;  
  
    public static void main (String[] args) {  
        System.out.println("Size is " + getSize());  
    }  
  
    public void setSize(int s) {  
        size = s;  
    }  
    public int getSize() {  
        return size;  
    }  
}
```

Calling getSize() just postpones the inevitable—getSize() uses the size instance variable.)

Back to the same problem... whose size?



there are no Dumb Questions

Q: What if you try to call a non-static method from a static method, but the non-static method doesn't use any instance variables. Will the compiler allow that?

A: No. The compiler knows that whether you do or do not use instance variables in a non-static method, you *can*. And think about the implications... if you were allowed to compile a scenario like that, then what happens if in the future you want to change the implementation of that non-static method so that one day it does use an instance variable? Or worse, what happens if a subclass overrides the method and uses an instance variable in the overriding version?

Q: I could swear I've seen code that calls a static method using a reference variable instead of the class name.

A: You *can* do that, but as your mother always told you, "Just because it's legal doesn't mean it's good." Although it works to call a static method using any instance of the class, it makes for misleading (less-readable) code. You *can* say,

```
Duck d = new Duck();  
String[] s = {};  
d.main(s);
```

This code is legal, but the compiler just resolves it back to the real class anyway ("OK, *d* is of type Duck, and *main()* is static, so I'll call the static *main()* in class Duck"). In other words, using *d* to invoke *main()* doesn't imply that *main()* will have any special knowledge of the object that *d* is referencing. It's just an alternate way to invoke a static method, but the method is still static!

Static variable: value is the same for ALL instances of the class

Imagine you wanted to count how many Duck instances are being created while your program is running. How would you do it? Maybe an instance variable that you increment in the constructor?

```
class Duck {
    int duckCount = 0;
    public Duck() {
        duckCount++;
    }
}
```

this would always set
duckCount to 1 each time
a Duck was made

No, that wouldn't work because duckCount is an instance variable, and starts at 0 for each Duck. You could try calling a method in some other class, but that's kludgy. You need a class that's got only a single copy of the variable, and all instances share that one copy.

That's what a static variable gives you: a value shared by all instances of a class. In other words, one value per *class*, instead of one value per *instance*.

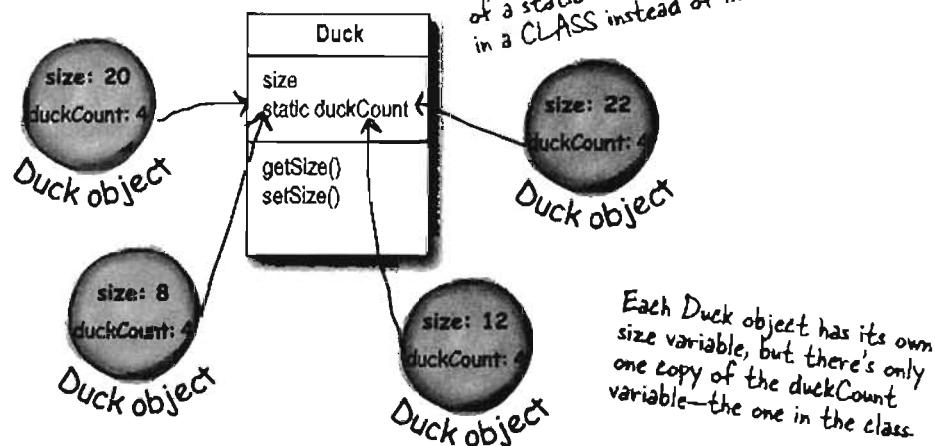
```
public class Duck {
    private int size;
    private static int duckCount = 0;

    public Duck() {
        duckCount++; ← Now it will keep
    }
}

public void setSize(int s) {
    size = s;
}

public int getSize() {
    return size;
}
```

The static duckCount variable is initialized ONLY when the class is first loaded, NOT each time a new instance is made.



static variables



Static variables are shared.

All instances of the same class share a single copy of the static variables.

instance variables: 1 per **instance**
static variables: 1 per **class**



Brain Barbell

Earlier in this chapter, we saw that a **private constructor** means that the class can't be instantiated from code running outside the class. In other words, only code from within the class can make a new instance of a class with a private constructor. (There's a kind of chicken-and-egg problem here.)

What If you want to write a class in such a way that only **ONE** instance of it can be created, and anyone who wants to use an instance of the class will always use that one, single instance?

Initializing a static variable

Static variables are initialized when a *class is loaded*. A class is loaded because the JVM decides it's time to load it. Typically, the JVM loads a class because somebody's trying to make a new instance of the class, for the first time, or use a static method or variable of the class. As a programmer, you also have the option of telling the JVM to load a class, but you're not likely to need to do that. In nearly all cases, you're better off letting the JVM decide when to *load* the class.

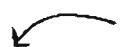
And there are two guarantees about static initialization:

Static variables in a class are initialized before any *object* of that class can be created.

Static variables in a class are initialized before any *static method* of the class runs.

```
class Player {
    static int playerCount = 0;
    private String name;
    public Player(String n) {
        name = n;
        playerCount++;
    }
}

public class PlayerTestDrive {
    public static void main(String[] args) {
        System.out.println(Player.playerCount);
        Player one = new Player("Tiger Woods");
        System.out.println(Player.playerCount);
    }
}
```



The playerCount is initialized when the class is loaded. We explicitly initialized it to 0, but we don't need to since 0 is the default value for ints. Static variables get default values just like instance variables.

Default values for declared but uninitialized static and instance variables are the same:
 primitive integers (long, short, etc.): 0
 primitive floating points (float, double): 0.0
 boolean: false
 object references: null

↑ Access a static variable just like a static method—with the class name.

Static variables are initialized when the class is loaded. If you don't explicitly initialize a static variable (by assigning it a value at the time you declare it), it gets a default value, so int variables are initialized to zero, which means we didn't need to explicitly say "playerCount = 0". Declaring, but not initializing, a static variable means the static variable will get the default value for that variable type, in exactly the same way that instance variables are given default values when declared.

All static variables in a class are initialized before any object of that class can be created.

```
File Edit Window Help What?
% java PlayerTestDrive
0 ← before any instances are made
1 ← after an object is created
```

static final constants

static final variables are constants

A variable marked **final** means that—once initialized—it can never change. In other words, the value of the static final variable will stay the same as long as the class is loaded. Look up `Math.PI` in the API, and you'll find:

```
public static final double PI = 3.141592653589793;
```

The variable is marked **public** so that any code can access it.

The variable is marked **static** so that you don't need an instance of class `Math` (which, remember, you're not allowed to create).

The variable is marked **final** because `PI` doesn't change (as far as Java is concerned).

There is no other way to designate a variable as a constant, but there is a naming convention that helps you to recognize one.

Constant variable names should be in all caps!

static initializer

Initialize a **final static** variable:

- At the time you declare it:

```
public class Foo {  
    public static final int FOO_X = 25;  
}  
  
OR  
  
notice the naming convention -- static  
final variables are constants, so the  
name should be all uppercase, with an  
underline separating the words
```

- In a static initializer:

```
public class Bar {  
    public static final double BAR_SIGN;  
  
    {  
        BAR_SIGN = (double) Math.random();  
    }  
}
```

this code runs as soon as the class is loaded, before any static method is called and even before any static variable can be used.

If you don't give a value to a final variable in one of those two places:

```
public class Bar {  
    public static final double BAR_SIGN;  
}  
no initialization!
```

The compiler will catch it:

```
File Edit Window Help Jack-In  
% javac Bar.java  
Bar.java:1: variable BAR_SIGN  
might not have been initialized  
1 error
```

final isn't just for static variables...

You can use the keyword `final` to modify non-static variables too, including instance variables, local variables, and even method parameters. In each case, it means the same thing: the value can't be changed. But you can also use `final` to stop someone from overriding a method or making a subclass.

non-static final variables

```
class Foof {
    final int size = 3; ← now you can't change size
    final int whuffle;

    Foof() {
        whuffle = 42; ← now you can't change whuffle
    }

    void doStuff(final int x) {
        // you can't change x
    }

    void doMore() {
        final int z = 7;
        // you can't change z
    }
}
```

final method

```
class Poof {
    final void calcWhuffle() {
        // important things
        // that must never be overridden
    }
}
```

final class

```
final class MyMostPerfectClass {
    // cannot be extended
}
```

A `final` variable means you can't change its value.

A `final` method means you can't override the method.

A `final` class means you can't extend the class (i.e. you can't make a subclass).



static and final

there are no Dumb Questions

Q: A static method can't access a non-static variable. But can a non-static method access a static variable?

A: Of course. A non-static method in a class can always call a static method in the class or access a static variable of the class.

Q: Why would I want to make a class final? Doesn't that defeat the whole purpose of OO?

A: Yes and no. A typical reason for making a class final is for security. You can't, for example, make a subclass of the String class. Imagine the havoc if someone extended the String class and substituted their own String subclass objects, polymorphically, where String objects are expected. If you need to count on a particular implementation of the methods in a class, make the class final.

Q: Isn't it redundant to have to mark the methods final if the class is final?

A: If the class is final, you don't need to mark the methods final. Think about it—if a class is final it can never be subclassed, so none of the methods can ever be overridden.

On the other hand, if you do want to allow others to extend your class, and you want them to be able to override some, but not all, of the methods, then don't mark the class final but go in and selectively mark specific methods as final. A final method means that a subclass can't override that particular method.

BULLET POINTS

- A static method should be called using the class name rather than an object reference variable: `Math.random()` vs. `myFoo.go()`
- A static method can be invoked without any instances of the method's class on the heap.
- A static method is good for a utility method that does not (and will never) depend on a particular instance variable value.
- A static method is not associated with a particular instance—only the class—so it cannot access any instance variable values of its class. It wouldn't know which instance's values to use.
- A static method cannot access a non-static method, since non-static methods are usually associated with instance variable state.
- If you have a class with only static methods, and you do not want the class to be instantiated, you can mark the constructor private.
- A static variable is a variable shared by all members of a given class. There is only one copy of a static variable in a class, rather than one copy per each individual instance for instance variables.
- A static method can access a static variable.
- To make a constant in Java, mark a variable as both static and final.
- A final static variable must be assigned a value either at the time it is declared, or in a static initializer.

```
static {  
    DOG_CODE = 420;  
}
```
- The naming convention for constants (final static variables) is to make the name all uppercase.
- A final variable value cannot be changed once it has been assigned.
- Assigning a value to a final instance variable must be either at the time it is declared, or in the constructor.
- A final method cannot be overridden.
- A final class cannot be extended (subclassed).



What's Legal?

Given everything you've just learned about static and final, which of these would compile?



```

❶ public class Foo {
    static int x;

    public void go() {
        System.out.println(x);
    }
}

❷ public class Foo2 {
    int x;

    public static void go() {
        System.out.println(x);
    }
}

❸ public class Foo3 {
    final int x;

    public void go() {
        System.out.println(x);
    }
}

❹ public class Foo4 {
    static final int x = 12;

    public void go() {
        System.out.println(x);
    }
}

❺ public class Foo5 {
    static final int x = 12;

    public void go(final int x) {
        System.out.println(x);
    }
}

❻ public class Foo6 {
    int x = 12;

    public static void go(final int x) {
        System.out.println(x);
    }
}

```

Math methods

Math methods

Now that we know how static methods work, let's look at some static methods in class Math. This isn't all of them, just the highlights. Check your API for the rest including sqrt(), tan(), ceil(), floor(), and asin().

Math.random()

Returns a double between 0.0 through (but not including) 1.0.

```
double r1 = Math.random();  
int r2 = (int) (Math.random() * 5);
```

Math.abs()

Returns a double that is the absolute value of the argument. The method is overloaded, so if you pass it an int it returns an int. Pass it a double it returns a double.

```
int x = Math.abs(-240); // returns 240  
double d = Math.abs(240.45); // returns 240.45
```

Math.round()

Returns an int or a long (depending on whether the argument is a float or a double) rounded to the nearest integer value.

```
int x = Math.round(-24.8f); // returns -25  
int y = Math.round(24.45f); // returns 24
```

Remember, floating point literals are assumed to be doubles unless you add the 'f'.

Math.min()

Returns a value that is the minimum of the two arguments. The method is overloaded to take ints, longs, floats, or doubles.

```
int x = Math.min(24, 240); // returns 24  
double y = Math.min(90876.5, 90876.49); // returns 90876.49
```

Math.max()

Returns a value that is the maximum of the two arguments. The method is overloaded to take ints, longs, floats, or doubles.

```
int x = Math.max(24, 240); // returns 240  
double y = Math.max(90876.5, 90876.49); // returns 90876.5
```

Wrapping a primitive

Sometimes you want to treat a primitive like an object. For example, in all versions of Java prior to 5.0, you cannot put a primitive directly into a collection like ArrayList or HashMap:

```
int x = 32;
ArrayList list = new ArrayList();
list.add(x);
```

This won't work unless you're using Java 5.0 or greater!! There's no add(int) method in ArrayList that takes an int! (ArrayList only has add() methods that take object references, not primitives.)

There's a wrapper class for every primitive type, and since the wrapper classes are in the java.lang package, you don't need to import them. You can recognize wrapper classes because each one is named after the primitive type it wraps, but with the first letter capitalized to follow the class naming convention.

Oh yeah, for reasons absolutely nobody on the planet is certain of, the API designers decided not to map the names *exactly* from primitive type to class type. You'll see what we mean:

Boolean

Character

Byte

Short

Integer

Long

Float

Double

Give the primitive to the wrapper constructor. That's it.

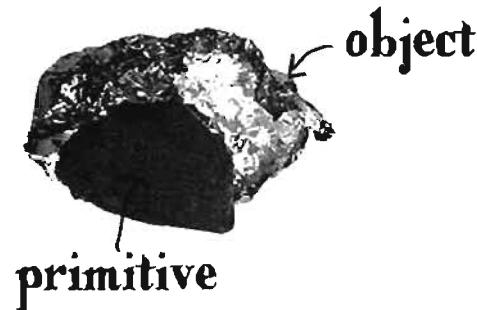
wrapping a value

```
int i = 288;
Integer iWrap = new Integer(i);
```

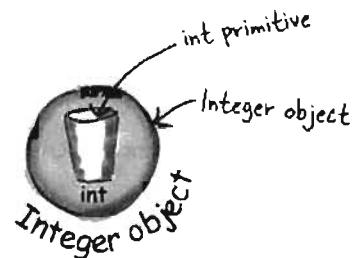
All the wrappers work like this. Boolean has a booleanValue(), Character has a charValue(), etc.

unwrapping a value

```
int unWrapped = iWrap.intValue();
```

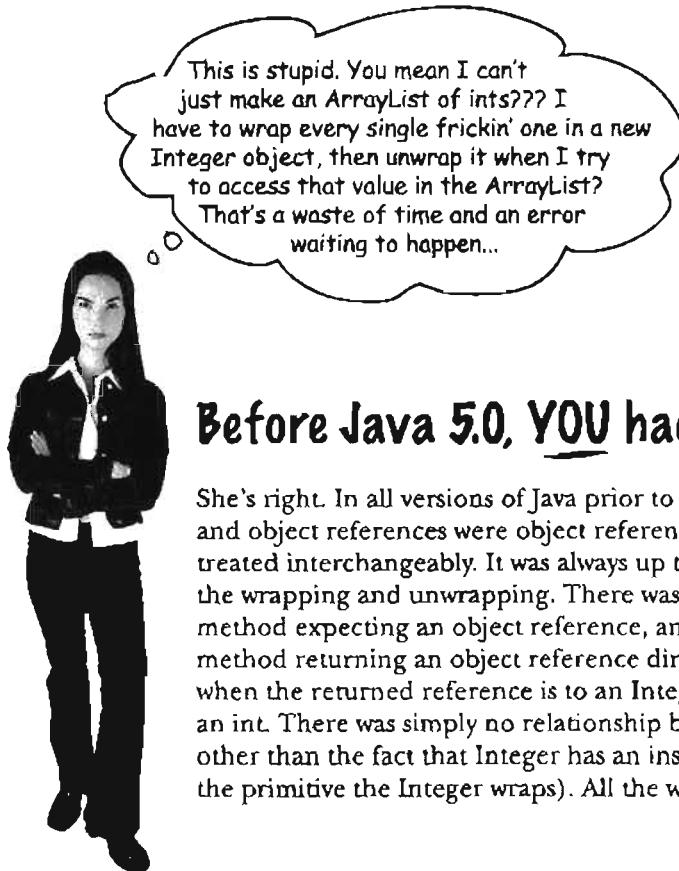


When you need to treat a primitive like an object, wrap it. If you're using any version of Java before 5.0, you'll do this when you need to store a primitive value inside a collection like ArrayList or HashMap.



Note: the picture at the top is a chocolate in a foil wrapper. Get it? Wrapper? Some people think it looks like a baked potato, but that works too.

static methods



Before Java 5.0, YOU had to do the work...

She's right. In all versions of Java prior to 5.0, primitives were primitives and object references were object references, and they were NEVER treated interchangeably. It was always up to you, the programmer, to do the wrapping and unwrapping. There was no way to pass a primitive to a method expecting an object reference, and no way to assign the result of a method returning an object reference directly to a primitive variable—even when the returned reference is to an Integer and the primitive variable is an int. There was simply no relationship between an Integer and an int, other than the fact that Integer has an instance variable of type int (to hold the primitive the Integer wraps). All the work was up to you.

An ArrayList of primitive ints

Without autoboxing (Java versions before 5.0)

```
public void doNumsOldWay() {  
    ArrayList listOfNumbers = new ArrayList();  
    listOfNumbers.add(new Integer(3)); ← You can't add the primitive '3' to the list,  
    // so you have to wrap it in an Integer first.  
    Integer one = (Integer) listOfNumbers.get(0); ← It comes out as type  
    int intOne = one.intValue(); ← Object, but you can cast  
    // the Object to an Integer.  
}
```

Make an ArrayList (Remember, before 5.0 you could not specify the TYPE, so all ArrayLists were lists of Objects.)

Finally you can get the primitive out of the Integer.

Autoboxing: blurring the line between primitive and object

The autoboxing feature added to Java 5.0 does the conversion from primitive to wrapper object *automatically!*

Let's see what happens when we want to make an ArrayList to hold ints.

An ArrayList of primitive ints

With autoboxing (Java versions 5.0 or greater)

```
public void doNumsNewWay() {
```

```
    ArrayList<Integer> listOfNumbers = new ArrayList<Integer>();
    listOfNumbers.add(3); Just add it!
    int num = listOfNumbers.get(0);
}
```

And the compiler automatically unwraps (unboxes) the Integer object so you can assign the int value directly to a primitive without having to call the intValue() method on the Integer object.

Make an ArrayList of type Integer.
↓

Although there is NOT a method in ArrayList for add(int), the compiler does all the wrapping (boxing) for you. In other words, there really IS an Integer object stored in the ArrayList, but you get to "pretend" that the ArrayList takes ints. (You can add both ints and Integers to an ArrayList<Integer>.)

Q: Why not declare an ArrayList<int> if you want to hold ints?

A: Because... you can't. Remember, the rule for generic types is that you can specify only class or interface types, *not primitives*. So ArrayList<int> will not compile. But as you can see from the code above, it doesn't really matter, since the compiler lets you put ints into the ArrayList<Integer>. In fact, there's really no way to prevent you from putting primitives into an ArrayList where the type of the list is the type of that primitive's wrapper, if you're using a Java 5.0-compliant compiler, since autoboxing will happen automatically. So, you can put boolean primitives in an ArrayList<Boolean> and chars into an ArrayList<Character>.

static methods

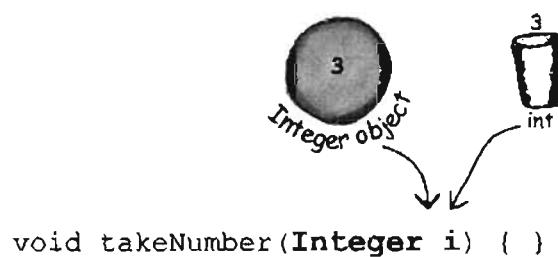
Autoboxing works almost everywhere

Autoboxing lets you do more than just the obvious wrapping and unwrapping to use primitives in a collection... it also lets you use either a primitive or its wrapper type virtually anywhere one or the other is expected. Think about that!

Fun with autoboxing

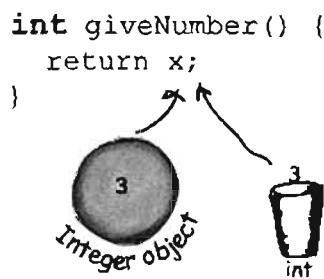
Method arguments

If a method takes a wrapper type, you can pass a reference to a wrapper or a primitive of the matching type. And of course the reverse is true—if a method takes a primitive, you can pass in either a compatible primitive or a reference to a wrapper of that primitive type.



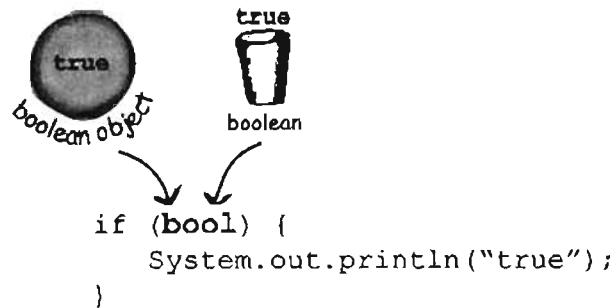
Return values

If a method declares a primitive return type, you can return either a compatible primitive or a reference to the wrapper of that primitive type. And if a method declares a wrapper return type, you can return either a reference to the wrapper type or a primitive of the matching type.



Boolean expressions

Any place a boolean value is expected, you can use either an expression that evaluates to a boolean ($4 > 2$), or a primitive boolean, or a reference to a Boolean wrapper.



Operations on numbers

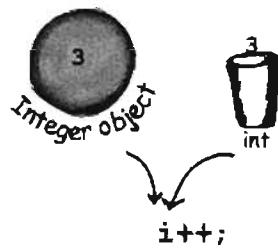
This is probably the strangest one—yes, you can now use a wrapper type as an operand in operations where the primitive type is expected. That means you can apply, say, the increment operator against a reference to an Integer object!

But don't worry—this is just a compiler trick. The language wasn't modified to make the operators work on objects; the compiler simply converts the object to its primitive type before the operation. It sure looks weird, though.

```
Integer i = new Integer(42);
i++;
```

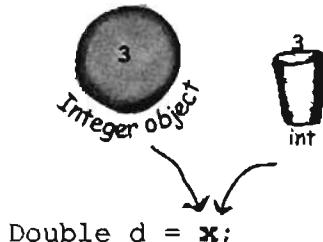
And that means you can also do things like:

```
Integer j = new Integer(5);
Integer k = j + 3;
```



Assignments

You can assign either a wrapper or primitive to a variable declared as a matching wrapper or primitive. For example, a primitive int variable can be assigned to an Integer reference variable, and vice-versa—a reference to an Integer object can be assigned to a variable declared as an int primitive.



Sharpen your pencil

```
public class TestBox {

    Integer i;
    int j;

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

    public void go() {
        j=i;
        System.out.println(j);
        System.out.println(i);
    }
}
```

Will this code compile? Will it run? If it runs, what will it do?

Take your time and think about this one; it brings up an implication of autoboxing that we didn't talk about.

You'll have to go to your compiler to find the answers. (Yes, we're forcing you to experiment, for your own good of course.)

wrapper methods

But wait! There's more! Wrappers have static utility methods too!

Besides acting like a normal class, the wrappers have a bunch of really useful static methods. We've used one in this book before—`Integer.parseInt()`.

The parse methods take a String and give you back a primitive value.

Converting a String to a primitive value is easy:

```
String s = "2";
int x = Integer.parseInt(s);
double d = Double.parseDouble("420.24");
```

No problem to parse
"2" into 2

```
boolean b = new Boolean("true").booleanValue();
```

You'd think there would be a
`Boolean.parseBoolean()` wouldn't you? But there
isn't. Fortunately there's a Boolean constructor
that takes (and parses) a String, and then you
just get the primitive value by unwrapping it.

But if you try to do this:

```
String t = "two";
int y = Integer.parseInt(t);
```

Uh-oh. This compiles just fine, but
at runtime it blows up. Anything
that can't be parsed as a number
will cause a `NumberFormatException`

You'll get a runtime exception:

```
File Edit Window Help Close
% java Wrappers
Exception in thread "main"
java.lang.NumberFormatException: two
at java.lang.Integer.parseInt(Integer.java:409)
at java.lang.Integer.parseInt(Integer.java:458)
at Wrappers.main(Wrappers.java:9)
```

Every method or constructor that parses a String can throw a `NumberFormatException`. It's a runtime exception, so you don't have to handle or declare it. But you might want to.

(We'll talk about Exceptions in the next chapter.)

And now in reverse... turning a primitive number into a String

There are several ways to turn a number into a String.
The easiest is to simply concatenate the number to an existing String.

```
double d = 42.5;
String doubleString = "" + d;
```

Remember the '+' operator is overloaded in Java (the only overloaded operator) as a String concatenator. Anything added to a String becomes Stringified.

```
double d = 42.5;
String doubleString = Double.toString(d);
```

Another way to do it using a static method in class Double.

Yeah,
but how do I make it
look like money? With a dollar
sign and two decimal places
like \$56.87 or what if I want
commas like 45,687,890 or
what if I want it in...

Where's my printf
like I have in C? Is
number formatting part of
the I/O classes?



Number formatting

In Java, formatting numbers and dates doesn't have to be coupled with I/O. Think about it. One of the most typical ways to display numbers to a user is through a GUI. You put Strings into a scrolling text area, or maybe a table. If formatting was built only into print statements, you'd never be able to format a number into a nice String to display in a GUI. Before Java 5.0, most formatting was handled through classes in the `java.text` package that we won't even look at in this version of the book, now that things have changed.

In Java 5.0, the Java team added more powerful and flexible formatting through a `Formatter` class in `java.util`. But you don't need to create and call methods on the `Formatter` class yourself, because Java 5.0 added convenience methods to some of the I/O classes (including `printf()`) and the `String` class. So it's a simple matter of calling a static `String.format()` method and passing it the thing you want formatted along with formatting instructions.

Of course, you do have to know how to supply the formatting instructions, and that takes a little effort unless you're familiar with the `printf()` function in C/C++. Fortunately, even if you *don't* know `printf()` you can simply follow recipes for the most basic things (that we're showing in this chapter). But you *will* want to learn how to format if you want to mix and match to get *anything* you want.

We'll start here with a basic example, then look at how it works. (Note: we'll revisit formatting again in the I/O chapter.)

Formatting a number to use commas

```
public class TestFormats {
    public static void main (String[] args) {
        String s = String.format("%, d", 1000000000);
        System.out.println(s);
    }
}
```

1,000,000,000

The annotations explain the arguments to the `String.format()` method:

- An annotation above the first argument ("%, d") says: "The number to format (we want it to have commas)."
- An annotation below the second argument ("1000000000") says: "The formatting instructions for how to format the second argument (which in this case is an int value). Remember, there are only two arguments to this method here—the first comma is INSIDE the String literal, so it isn't separating arguments to the format method."
- A handwritten note below the output says: "Now we get commas inserted into the number." with an arrow pointing to the comma in the output.

Formatting deconstructed...

At the most basic level, formatting consists of two main parts (there is more, but we'll start with this to keep it cleaner):



Formatting Instructions

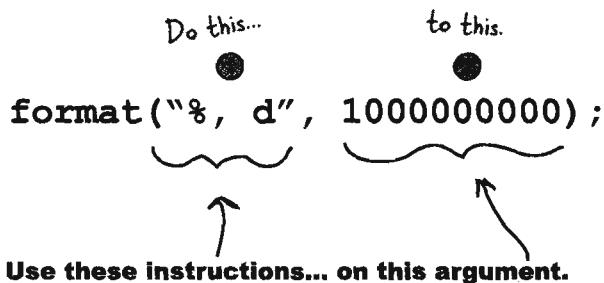
You use special format specifiers that describe how the argument should be formatted.



The argument to be formatted.

Although there can be more than one argument, we'll start with just one. The argument type can't be just *anything*... it has to be something that can be formatted using the format specifiers in the formatting instructions. For example, if your formatting instructions specify a *floating point number*, you can't pass in a Dog or even a String that looks like a floating point number.

Note: if you already know printf() from C/C++, you can probably just skim the next few pages. Otherwise, read carefully!



What do these instructions actually say?

"Take the second argument to this method, and format it as a decimal integer and insert commas."

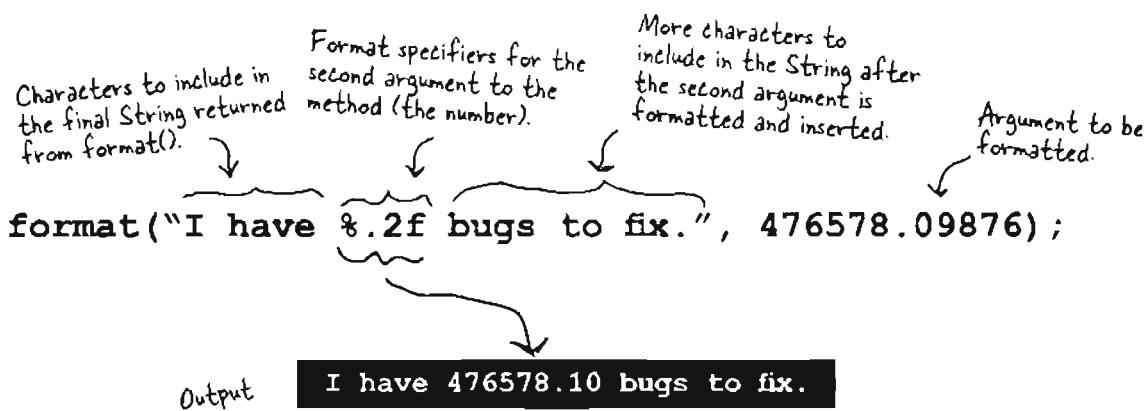
How do they say that?

On the next page we'll look in more detail at what the syntax "%, d" actually means, but for starters, any time you see the percent sign (%) in a format String (which is always the first argument to a format() method), think of it as representing a variable, and the variable is the other argument to the method. The rest of the characters after the percent sign describe the formatting instructions for the argument.

the format() method

The percent (%) says, "insert argument here" (and format it using these instructions)

The first argument to a format() method is called the format String, and it can actually include characters that you just want printed as-is, without extra formatting. When you see the % sign, though, think of the percent sign as a variable that represents the other argument to the method.



Notice we lost some of the numbers after the decimal point. Can you guess what the ".2f" means?

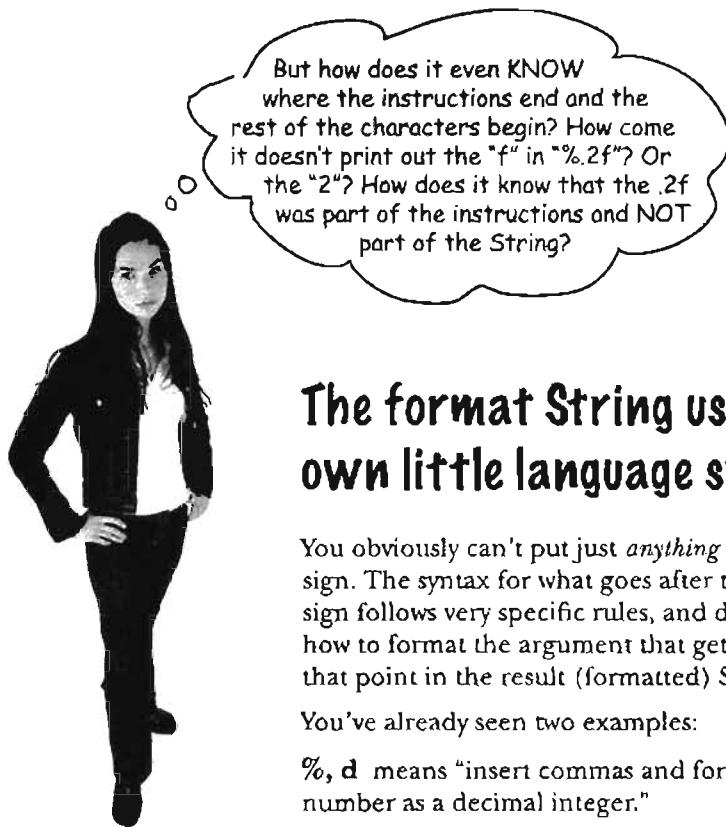
The "%" sign tells the formatter to insert the other method argument (the second argument to format(), the number) here, AND format it using the ".2f" characters after the percent sign. Then the rest of the format String, "bugs to fix", is added to the final output.

Adding a comma

```
format("I have %,.2f bugs to fix.", 476578.09876);
```

`I have 476,578.10 bugs to fix.`

By changing the format instructions from "%,.2f" to "%,2f", we got a comma in the formatted number.



The format String uses its own little language syntax

You obviously can't put just *anything* after the "%" sign. The syntax for what goes after the percent sign follows very specific rules, and describes how to format the argument that gets inserted at that point in the result (formatted) String.

You've already seen two examples:

%, d means "insert commas and format the number as a decimal integer."

and

%.2f**** means "format the number as a floating point with a precision of two decimal places."

and

%,.2f**** means "insert commas and format the number as a floating point with a precision of two decimal places."

The real question is really, "How do I know what to put after the percent sign to get it to do what I want?" And that includes knowing the symbols (like "d" for decimal and "f" for floating point) as well as the order in which the instructions must be placed following the percent sign. For example, if you put the comma after the "d" like this: "%d," instead of "%,d" it won't work!

Or will it? What do you think this will do:

```
String.format("I have %.2f, bugs to fix.", 476578.09876);
```

(We'll answer that on the next page.)

format specifier

The format specifier

Everything after the percent sign up to and including the type indicator (like "d" or "f") are part of the formatting instructions. After the type indicator, the formatter assumes the next set of characters are meant to be part of the output String, until or unless it hits another percent (%) sign. Hmmmm... is that even possible? Can you have more than one formatted argument variable? Put that thought on hold for right now; we'll come back to it in a few minutes. For now, let's look at the syntax for the format specifiers—the things that go after the percent (%) sign and describe how the argument should be formatted.

A format specifier can have up to five different parts (not including the "%"). Everything in brackets [] below is optional, so only the percent (%) and the type are required. But the order is also mandatory, so any parts you DO use must go in this order.

%[argument number] [flags] [width] [.precision] type

We'll get to this later... it lets you say WHICH argument if there's more than one. (Don't worry about it just yet.)

These are for special formatting options like inserting commas, or putting negative numbers in parentheses, or to make the numbers left justified.

This defines the MINIMUM number of characters that will be used. That's *minimum* not TOTAL. If the number is longer than the width, it'll still be used in full, but if it's less than the width, it'll be padded with zeroes.

You already know this one...it defines the precision. In other words, it sets the number of decimal places. Don't forget to include the ":" in there.

Type is mandatory (see the next page) and will usually be "d" for a decimal integer or "f" for a floating point number.

%[argument number] [flags] [width] [.precision] type

```
format ("%,.1f", 42.000);
```

There's no "argument number" specified in this format String, but all the other pieces are there.

The only required specifier is for TYPE

Although type is the only required specifier, remember that if you *do* put in anything else, type must always come last! There are more than a dozen different type modifiers (not including dates and times; they have their own set), but most of the time you'll probably use %d (decimal) or %f (floating point). And typically you'll combine %f with a precision indicator to set the number of decimal places you want in your output.

The TYPE is mandatory, everything else is optional.

%d **decimal**

```
format("%d", 42);
```

42

A 42.25 would not work! It would be the same as trying to directly assign a double to an int variable.

The argument must be compatible with an int, so that means only byte, short, int, and char (or their wrapper types).

%f **floating point**

```
format("%.3f", 42.000000);
```

42.000

Here we combined the "f" with a precision indicator ".3" so we ended up with three zeroes.

The argument must be of a floating point type, so that means only a float or double (primitive or wrapper) as well as something called BigDecimal (which we don't look at in this book).

%x **hexadecimal**

```
format("%x", 42);
```

2a

The argument must be a byte, short, int, long (including both primitive and wrapper types), and BigInteger.

%c **character**

```
format("%c", 42);
```

*

The number 42 represents the char "*".

The argument must be a byte, short, char, or int (including both primitive and wrapper types).

You must include a type in your format instructions, and if you specify things besides type, the type must always come last.

Most of the time, you'll probably format numbers using either "d" for decimal or "f" for floating point.

format arguments

What happens if I have more than one argument?

Imagine you want a String that looks like this:

"The rank is 20,456,654 out of 100,567,890.24."

But the numbers are coming from variables. What do you do? You simply add *two* arguments after the format String (first argument), so that means your call to format() will have three arguments instead of two. And inside that first argument (the format String), you'll have two different format specifiers (two things that start with "%"). The first format specifier will insert the second argument to the method, and the second format specifier will insert the third argument to the method. In other words, the variable insertions in the format String use the order in which the other arguments are passed into the format() method.

```
int one = 20456654;  
double two = 100567890.248907;  
String s = String.format("The rank is %,d out of %,.2f", one, two);
```

The rank is 20,456,654 out of 100,567,890.25

We added commas to both variables, and restricted the floating point number (the second variable) to two decimal places.

When you have more than one argument, they're inserted using the order in which you pass them to the format() method.

As you'll see when we get to date formatting, you might actually want to apply different formatting specifiers to the same argument. That's probably hard to imagine until you see how *date* formatting (as opposed to the *number* formatting we've been doing) works. Just know that in a minute, you'll see how to be more specific about which format specifiers are applied to which arguments.

Q: Um, there's something REALLY strange going on here. Just how many arguments can I pass? I mean, how many overloaded format() methods are IN the String class? So, what happens if I want to pass, say, ten different arguments to be formatted for a single output String?

A: Good catch. Yes, there *is* something strange (or at least new and different) going on, and no there are *not* a bunch of overloaded format() methods to take a different number of possible arguments. In order to support this new formatting (printf-like) API in Java, the language needed another new feature—*variable argument lists* (called *varargs* for short). We'll talk about varargs only in the appendix because outside of formatting, you probably won't use them much in a well-designed system.

So much for numbers, what about dates?

Imagine you want a String that looks like this: "Sunday, Nov 28 2004"

Nothing special there, you say? Well, imagine that all you have to start with is a variable of type Date—A Java class that can represent a timestamp, and now you want to take that object (as opposed to a number) and send it through the formatter.

The main difference between number and date formatting is that date formats use a two-character type that starts with "t" (as opposed to the single character "f" or "d", for example). The examples below should give you a good idea of how it works:

The complete date and time %tc

```
String.format("%tc", new Date());
```

Sun Nov 28 14:52:41 MST 2004

Just the time %tr

```
String.format("%tr", new Date());
```

03:01:47 PM

Day of the week, month and day %tA %tB %td

There isn't a single format specifier that will do exactly what we want, so we have to combine three of them for day of the week (%tA), month (%tB), and day of the month (%td).

```
Date today = new Date();
String.format("%tA, %tB %td", today, today, today)
```

↑
The comma is not part of the formatting... it's just the character we want printed after the first inserted formatted argument.

But that means we have to pass the Date object in three times, one for each part of the format that we want. In other words, the %tA will give us just the day of the week, but then we have to do it again to get just the month and again for the day of the month.

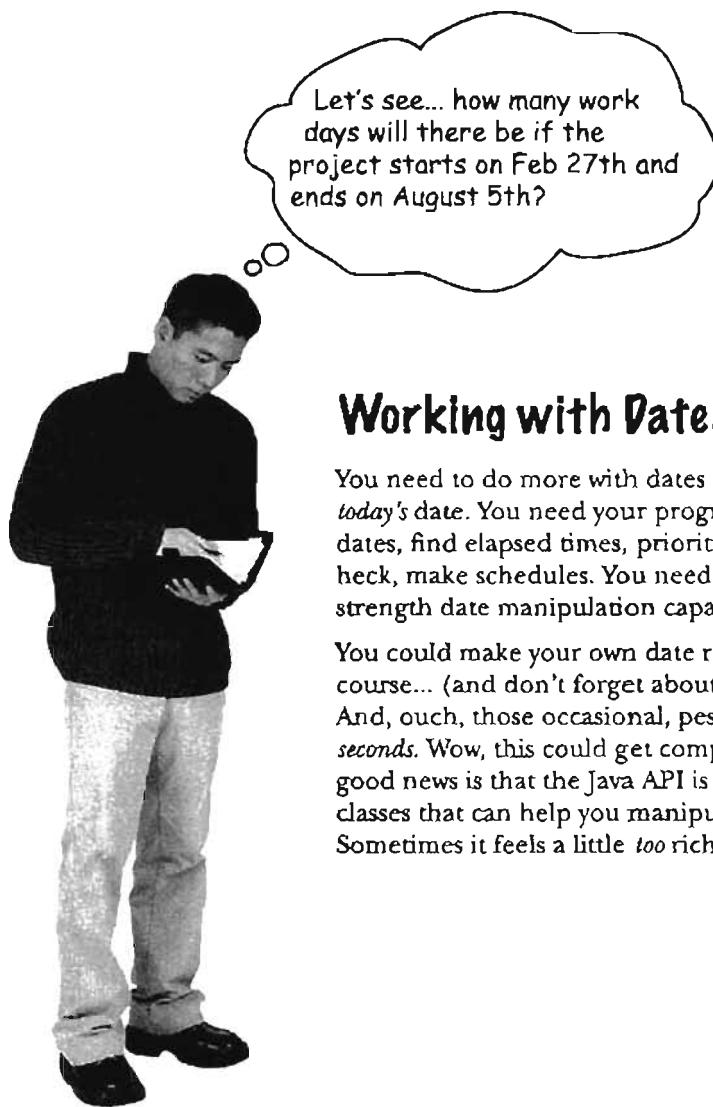
Sunday, November 28

Same as above, but without duplicating the arguments %tA %tB %td

```
Date today = new Date();
String.format("%tA, %<tB %<td", today);
```

You can think of this as kind of like calling three different getter methods on the Date object, to get three different pieces of data from it.

The angle-bracket "<" is just another flag in the specifier that tells the formatter to "use the previous argument again." So it saves you from repeating the arguments, and instead you format the same argument three different ways.



Working with Dates

You need to do more with dates than just get *today's date*. You need your programs to adjust dates, find elapsed times, prioritize schedules, heck, make schedules. You need industrial strength date manipulation capabilities.

You could make your own date routines of course... (and don't forget about leap years!) And, ouch, those occasional, pesky leap seconds. Wow, this could get complicated. The good news is that the Java API is rich with classes that can help you manipulate dates. Sometimes it feels a little *too* rich...

Moving backward and forward in time

Let's say your company's work schedule is Monday through Friday. You've been assigned the task of figuring out the last work day in each calendar month this year...

It seems that `java.util.Date` is actually... out of date

Earlier we used `java.util.Date` to find today's date, so it seems logical that this class would be a good place to start looking for some handy date manipulation capabilities, but when you check out the API you'll find that most of `Date`'s methods have been deprecated!

The `Date` class is still great for getting a "time stamp"—an object that represents the current date and time, so use it when you want to say, "give me NOW".

The good news is that the API recommends `java.util.Calendar` instead, so let's take a look:

Use `java.util.Calendar` for your date manipulation

The designers of the `Calendar` API wanted to think globally, literally. The basic idea is that when you want to work with dates, you ask for a `Calendar` (through a static method of the `Calendar` class that you'll see on the next page), and the JVM hands you back an instance of a concrete subclass of `Calendar`. (`Calendar` is actually an abstract class, so you're always working with a concrete subclass.)

More interesting, though, is that the *kind* of `calendar` you get back will be *appropriate for your locale*. Much of the world uses the Gregorian calendar, but if you're in an area that doesn't use a Gregorian calendar you can get Java libraries to handle other calendars such as Buddhist, or Islamic or Japanese.

The standard Java API ships with `java.util.GregorianCalendar`, so that's what we'll be using here. For the most part, though, you don't even have to think about the kind of `Calendar` subclass you're using, and instead focus only on the methods of the `Calendar` class.

For a time-stamp of "now", use `Date`. But for everything else, use `Calendar`.

getting a Calendar

Getting an object that extends Calendar

How in the world do you get an “instance” of an abstract class?
Well you don’t of course, this won’t work:

This WON’T work:

```
Calendar cal = new Calendar();
```

The compiler won’t allow this!

Instead, use the static “getInstance()” method:

```
Calendar cal = Calendar.getInstance();
```

This syntax should look familiar at this point – we’re invoking a static method.

Wait a minute.
If you can’t make an instance of the Calendar class, what exactly are you assigning to that Calendar reference?



You can’t get an instance of Calendar, but you can get an instance of a concrete Calendar subclass.

Obviously you can’t get an instance of Calendar, because Calendar is abstract. But you’re still free to call static methods on Calendar, since *static* methods are called on the *class*, rather than on a particular instance. So you call the static getInstance() on Calendar and it gives you back... an instance of a concrete subclass. Something that extends Calendar (which means it can be polymorphically assigned to Calendar) and which—by contract—can respond to the methods of class Calendar.

In most of the world, and by default for most versions of Java, you’ll be getting back a `java.util.GregorianCalendar` instance.

Working with Calendar objects

There are several key concepts you'll need to understand in order to work with Calendar objects:

- **Fields hold state** - A Calendar object has many fields that are used to represent aspects of its ultimate state, its date and time. For instance, you can get and set a Calendar's *year* or *month*.
- **Dates and Times can be incremented** - The Calendar class has methods that allow you to add and subtract values from various fields, for example "add one to the month", or "subtract three years".
- **Dates and Times can be represented in milliseconds** - The Calendar class lets you convert your dates into and out of a millisecond representation. (Specifically, the number of milliseconds that have occurred since January 1st, 1970.) This allows you to perform precise calculations such as "elapsed time between two times" or "add 63 hours and 23 minutes and 12 seconds to this time".

An example of working with a Calendar object:

```

Calendar c = Calendar.getInstance();           Set time to Jan. 1, 2004 at 15:40.
c.set(2004,0,7,15,40);                      (Notice the month is zero-based.)
long day1 = c.getTimeInMillis();             Convert this to a big ol'
day1 += 1000 * 60 * 60;                     Add an hour's worth of millis, then update the time.
c.setTimeInMillis(day1);                    (Notice the "+=", it's like day1 = day1 + ...).

System.out.println("new hour " + c.get(c.HOUR_OF_DAY));
c.add(c.DATE, 35);                         Add 35 days to the date, which
System.out.println("add 35 days " + c.getTime());    should move us into February.

c.roll(c.DATE, 35);                        ← "Roll" 35 days onto this date. This
System.out.println("roll 35 days " + c.getTime());   "rolls" the date ahead 35 days, but
c.set(c.DATE, 1);                          ← DOES NOT change the month!

System.out.println("set to 1 " + c.getTime());

```

We're not incrementing here, just doing a "set" of the date.

```

File Edit Window Help Time-Files
new hour 16
add 35 days Wed Feb 11 16:40:41 MST 2004
roll 35 days Tue Feb 17 16:40:41 MST 2004
set to 1 Sun Feb 01 16:40:41 MST 2004

```

This output confirms how millis, add, roll, and set work.

Highlights of the Calendar API

We just worked through using a few of the fields and methods in the `Calendar` class. This is a big API, so we're showing only a few of the most common fields and methods that you'll use. Once you get a few of these it should be pretty easy to bend the rest of the this API to your will.

Key Calendar Methods

`add(int field, int amount)`

Adds or subtracts time from the calendar's field.

`get(int field)`

Returns the value of the given calendar field.

`getInstance()`

Returns a `Calendar`, you can specify a locale.

`getTimeInMillis()`

Returns this `Calendar`'s time in millis, as a long.

`roll(int field, boolean up)`

Adds or subtracts time without changing larger fields.

`set(int field, int value)`

Sets the value of a given `Calendar` field.

`set(year, month, day, hour, minute) (all ints)`

A common variety of `set` to set a complete time.

`setTimeInMillis(long millis)`

Sets a `Calendar`'s time based on a long milli-time.

// more...

Key Calendar Fields

`DATE / DAY_OF_MONTH`

Get / set the day of month

`HOUR / HOUR_OF_DAY`

Get / set the 12 hour or 24 hour value.

`MILLISECOND`

Get / set the milliseconds.

`MINUTE`

Get / set the minute.

`MONTH`

Get / set the month.

`YEAR`

Get / set the year.

`ZONE_OFFSET`

Get / set raw offset of GMT in millis.

// more...

Even more Statics!... static imports

New to Java 5.0... a real mixed blessing. Some people love this idea, some people hate it. Static imports exist only to save you some typing. If you hate to type, you might just like this feature. The downside to static imports is that - if you're not careful - using them can make your code a lot harder to read.

The basic idea is that whenever you're using a static class, a static variable, or an enum (more on those later), you can import them, and save yourself some typing.

Some old-fashioned code:

```
import java.lang.Math;

class NoStatic {

    public static void main(String [] args) {

        System.out.println("sqrt " + Math.sqrt(2.0));
        System.out.println("tan " + Math.tan(60));
    }
}
```

The syntax to use when declaring static imports.

Same code, with static imports:

```
import static java.lang.System.out;
import static java.lang.Math.*;

class WithStatic {

    public static void main(String [] args) {

        out.println("sqrt " + sqrt(2.0));
        out.println("tan " + tan(60));
    }
}
```

Static imports in action.

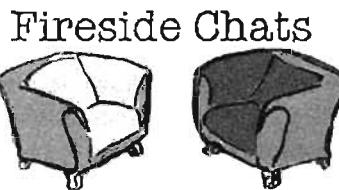
Use Carefully:

static imports can make your code confusing to read

- Caveats & Gotchas

- If you're only going to use a static member a few times, we think you should avoid static imports, to help keep the code more readable.
- If you're going to use a static member a lot, (like doing lots of Math calculations), then it's probably OK to use the static import.
- Notice that you can use wildcards (*), in your static import declaration.
- A big issue with static imports is that it's not too hard to create naming conflicts. For example, if you have two different classes with an "add()" method, how will you and the compiler know which one to use?

static vs. instance



Fireside Chats

Tonight's Talk: **An instance variable takes cheap shots at a static variable**

Instance Variable

I don't even know why we're doing this. Everyone knows static variables are just used for constants. And how many of those are there? I think the whole API must have, what, four? And it's not like anybody ever uses them.

Full of it. Yeah, you can say that again. OK, so there are a few in the Swing library, but everybody knows Swing is just a special case.

Ok, but besides a few GUI things, give me an example of just one static variable that anyone would actually use. In the real world.

Well, that's another special case. And nobody uses that except for debugging anyway.

Static Variable

You really should check your facts. When was the last time you looked at the APP? It's frickin' loaded with statics! It even has entire classes dedicated to holding constant values. There's a class called `SwingConstants`, for example, that's just full of them.

It might be a special case, but it's a really important one! And what about the `Color` class? What a pain if you had to remember the RGB values to make the standard colors? But the `color` class already has constants defined for blue, purple, white, red, etc. Very handy.

How's `System.out` for starters? The `out` in `System.out` is a static variable of the `System` class. You personally don't make a new instance of the `System`, you just ask the `System` class for its `out` variable.

Oh, like debugging isn't important?

And here's something that probably never crossed your narrow mind—let's face it, static variables are more efficient. One per class instead of one per instance. The memory savings might be huge!

Instance Variable

Um, aren't you forgetting something?

Static variables are about as un-OO as it gets!!
Gee why not just go take a giant backwards
step and do some procedural programming
while we're at it.

You're like a global variable, and any
programmer worth his PDA knows that's
usually a Bad Thing.

Yeah you live in a class, but they don't call
it *Class-Oriented* programming. That's just
stupid. You're a relic. Something to help the
old-timers make the leap to java.

Well, OK, every once in a while sure, it makes
sense to use a static, but let me tell you, abuse
of static variables (and methods) is the mark
of an immature OO programmer. A designer
should be thinking about *object* state, not *class*
state.

Static methods are the worst things of all,
because it usually means the programmer is
thinking procedurally instead of about objects
doing things based on their unique object
state.

Riiiiiight. Whatever you need to tell yourself...

Static Variable

What?

What do you mean *un-OO*?

I am NOT a global variable. There's no such
thing. I live in a class! That's pretty OO you
know, a CLASS. I'm not just sitting out there
in space somewhere; I'm a natural part of the
state of an object; the only difference is that
I'm shared by all instances of a class. Very
efficient.

Alright just stop right there. THAT is
definitely not true. Some static variables are
absolutely crucial to a system. And even the
ones that aren't crucial sure are handy.

Why do you say that? And what's wrong with
static methods?

Sure, I know that objects should be the focus
of an OO design, but just because there are
some clueless programmers out there... don't
throw the baby out with the bytecode. There's
a time and place for statics, and when you
need one, nothing else beats it.

be the compiler



BE the compiler

The Java file on this page represents a complete program. Your job is to play compiler and determine whether this file will compile. If it won't compile, how would you fix it, and if it does compile, what would be its output?



```
class StaticSuper{

    static {
        System.out.println("super static block");
    }

    StaticSuper{
        System.out.println(
            "super constructor");
    }
}

public class StaticTests extends StaticSuper {
    static int rand;

    static {
        rand = (int) (Math.random() * 6);
        System.out.println("static block " + rand);
    }

    StaticTests() {
        System.out.println("constructor");
    }

    public static void main(String [] args) {
        System.out.println("in main");
        StaticTests st = new StaticTests();
    }
}
```

If it compiles, which of these is the output?

Possible Output

```
File Edit Window Help Cling
%java StaticTests
static block 4
in main
super static block
super constructor
constructor
```

Possible Output

```
File Edit Window Help Electricity
%java StaticTests
super static block
static block 3
in main
super constructor
constructor
```



This chapter explored the wonderful, static, world of Java. Your job is to decide whether each of the following statements is true or false.

TRUE OR FALSE

1. To use the Math class, the first step is to make an instance of it.
2. You can mark a constructor with the `static` keyword.
3. Static methods don't have access to instance variable state of the 'this' object.
4. It is good practice to call a static method using a reference variable.
5. Static variables could be used to count the instances of a class.
6. Constructors are called before static variables are initialized.
7. MAX_SIZE would be a good name for a static final variable.
8. A static initializer block runs before a class's constructor runs.
9. If a class is marked final, all of its methods must be marked final.
10. A final method can only be overridden if its class is extended.
11. There is no wrapper class for boolean primitives.
12. A wrapper is used when you want to treat a primitive like an object.
13. The `parseXxx` methods always return a `String`.
14. Formatting classes (which are decoupled from I/O), are in the `java.format` package.

code magnets



Lunar Code Magnets

This one might actually be useful! In addition to what you've learned in the last few pages about manipulating dates, you'll need a little more information... First, full moons happen every 29.52 days or so. Second, there was a full moon on Jan. 7th, 2004. Your job is to reconstruct the code snippets to make a working Java program that produces the output listed below (plus more full moon dates). (You might not need all of the magnets, and add all the curly braces you need.) Oh, by the way, your output will be different if you don't live in the mountain time zone.

```
long day1 = c.getTimeInMillis();
c.set(2004,1,7,15,40);

import static java.lang.System.out;
static int DAY_IM = 60 * 60 * 24;
("full moon on %tc", c));
(c.format
Calendar c = new Calendar();
class FullMoons {

public static void main(String [] args) {
day1 += (DAY_IM * 29.52);
for (int x = 0; x < 60; x++) {
static int DAY_IM = 1000 * 60 * 60 * 24;
println
import java.io.*;
("full moon on %t", c));
import java.util.*;
static import java.lang.System.out;
c.setTimeInMillis(day1);
out.println
(String.format
Calendar c = Calendar.getInstance();
```

File Edit Window Help How
? java FullMoons
full moon on Fri Feb 06 04:09:35 MST 2004
full moon on Sat Mar 06 16:38:23 MST 2004
full moon on Mon Apr 05 06:07:11 MDT 2004

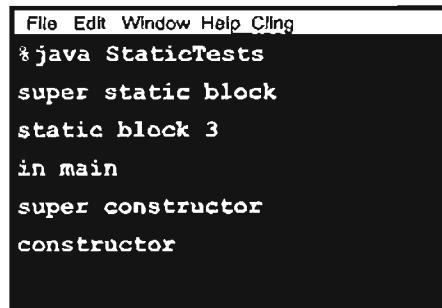
Exercise Solutions

BE the compiler

```
StaticSuper() {
    System.out.println(
        "super constructor");
}
```

StaticSuper is a constructor, and must have () in its signature. Notice that as the output below demonstrates, the static blocks for both classes run before either of the constructors run.

Possible Output



```
File Edit Window Help C:\>
*java StaticTests
super static block
static block 3
in main
super constructor
constructor
```

True or False

1. To use the Math class, the first step is to make an instance of it. **False**
2. You can mark a constructor with the keyword 'static'. **False**
3. Static methods don't have access to an object's instance variables. **True**
4. It is good practice to call a static method using a reference variable. **False**
5. Static variables could be used to count the instances of a class. **True**
6. Constructors are called before static variables are initialized. **False**
7. MAX_SIZE would be a good name for a static final variable. **True**
8. A static initializer block runs before a class's constructor runs. **True**
9. If a class is marked final, all of its methods must be marked final. **False**
10. A final method can only be overridden if its class is extended. **False**
11. There is no wrapper class for boolean primitives. **False**
12. A wrapper is used when you want to treat a primitive like an object. **True**
13. The parseXxx methods always return a String. **False**
14. Formatting classes (which are decoupled from I/O), are in the java.format package. **False**



Exercise Solutions

```
import java.util.*;
import static java.lang.System.out;
class FullMoons {
    static int DAY_IM = 1000 * 60 * 60 * 24;
    public static void main(String [] args) {
        Calendar c = Calendar.getInstance();
        c.set(2004,0,7,15,40);
        long day1 = c.getTimeInMillis();
        for (int x = 0; x < 60; x++) {
            day1 += (DAY_IM * 29.52)
            c.setTimeInMillis(day1);
            out.println(String.format("full moon on %tc", c));
        }
    }
}
```

```
File Edit Window Help How
$ java FullMoons
full moon on Fri Feb 06 04:09:35 MST 2004
full moon on Sat Mar 06 16:38:23 MST 2004
full moon on Mon Apr 05 06:07:11 MDT 2004
```

Notes on the Lunar Code Magnet:

You might discover that a few of the dates produced by this program are off by a day. This astronomical stuff is a little tricky, and if we made it perfect, it would be too complex to make an exercise here.

Hint: one problem you might try to solve is based on differences in time zones. Can you spot the issue?

Risky Behavior



Stuff happens. The file isn't there. The server is down. No matter how good a programmer you are, you can't control everything. Things can go wrong. *Very wrong.* When you write a *risky* method, you need code to handle the bad things that might happen. But how do you know when a method is *risky*? And where do you put the code to *handle* the *exceptional* situation? So far in this book, we haven't *really* taken any risks. We've certainly had things go wrong at runtime, but the problems were mostly flaws in our own code. Bugs. And those we should fix at development time. No, the problem-handling code we're talking about here is for code that you *can't* guarantee will work at runtime. Code that expects the file to be in the right directory, the server to be running, or the Thread to stay asleep. And we have to do this now. Because in *this* chapter, we're going to build something that uses the *risky* JavaSound API. We're going to build a MIDI Music Player.

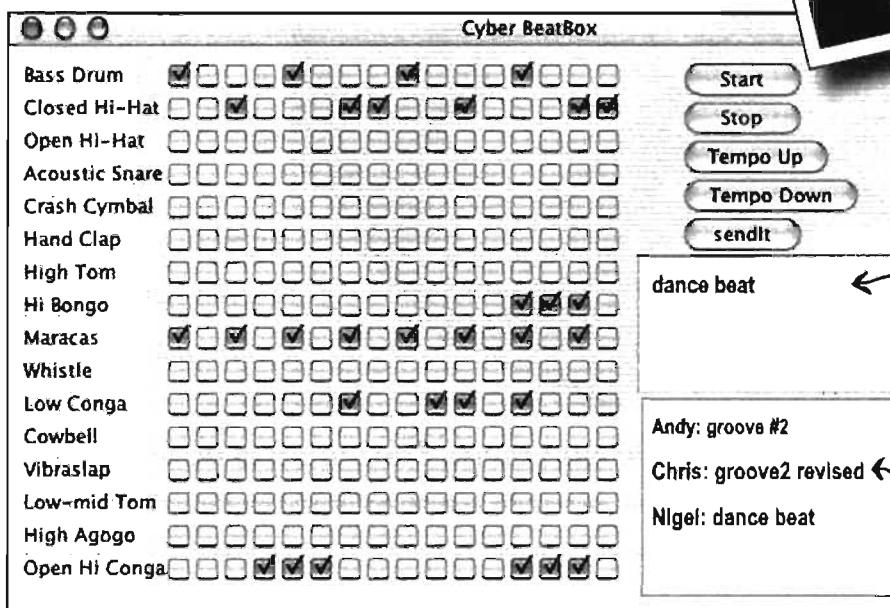
Let's make a Music Machine

Over the next three chapters, we'll build a few different sound applications, including a BeatBox Drum Machine. In fact, before the book is done, we'll have a multi-player version so you can send your drum loops to another player, kind of like a chat room. You're going to write the whole thing, although you can choose to use Ready-bake code for the GUI parts.

OK, so not every IT department is looking for a new BeatBox server, but we're doing this to *learn* more about Java. Building a BeatBox is just a way to have fun *while* we're learning Java.

The finished BeatBox looks something like this:

You make a beatbox loop (a 16-beat drum pattern) by putting checkmarks in the boxes.



your message, that gets sent to the other players, along with your current beat pattern, when you hit "Sendit"

incoming messages from other players. Click one to load the pattern that goes with it, and then click 'Start' to play it

Put checkmarks in the boxes for each of the 16 'beats'. For example, on beat 1 (of 16) the Bass drum and the Maracas will play, on beat 2 nothing, and on beat 3 the Maracas and Closed Hi-Hat... you get the idea. When you hit 'Start', it plays your pattern in a loop until you hit 'Stop'. At any time, you can "capture" one of your own patterns by sending it to the BeatBox server (which means any other players can listen to it). You can also load any of the incoming patterns by clicking on the message that goes with it.



We'll start with the basics

Obviously we've got a few things to learn before the whole program is finished, including how to build a Swing GUI, how to *connect* to another machine via networking, and a little I/O so we can *send* something to the other machine.

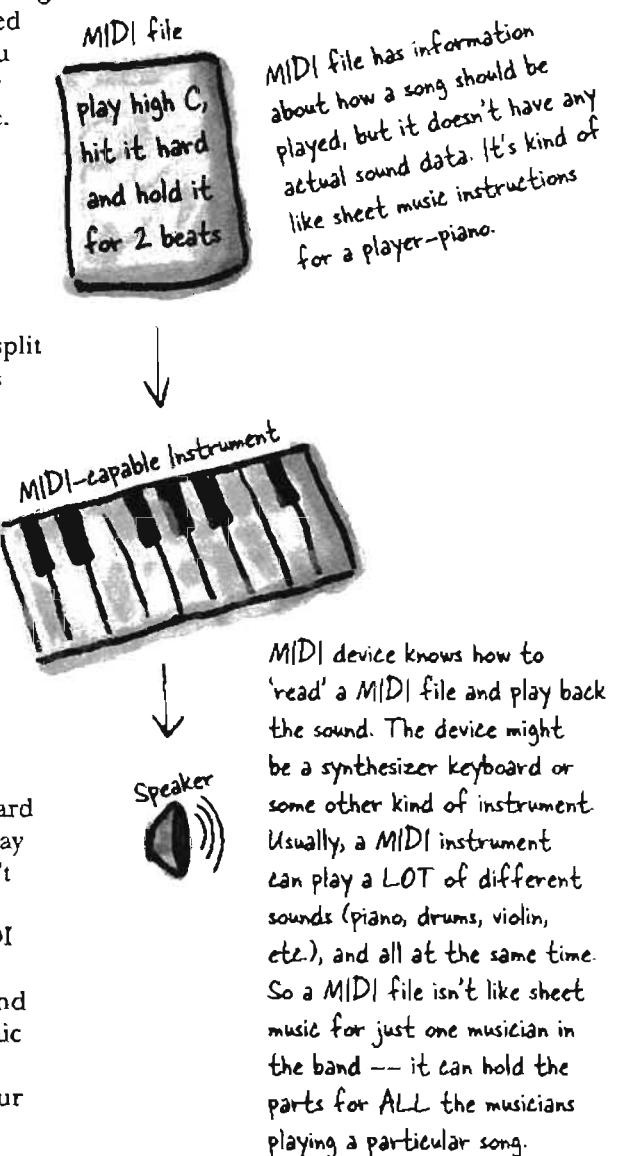
Oh yeah, and the JavaSound API. That's where we'll start in this chapter. For now, you can forget the GUI, forget the networking and the I/O, and focus only on getting some MIDI-generated sound to come out of your computer. And don't worry if you don't know a thing about MIDI, or a thing about reading or making music. Everything you need to learn is covered here. You can almost smell the record deal.

The JavaSound API

JavaSound is a collection of classes and interfaces added to Java starting with version 1.3. These aren't special add-ons; they're part of the standard J2SE class library. JavaSound is split into two parts: MIDI and Sampled. We use only MIDI in this book. MIDI stands for Musical Instrument Digital Interface, and is a standard protocol for getting different kinds of electronic sound equipment to communicate. But for our BeatBox app, you can think of MIDI as *a kind of sheet music* that you feed into some device you can think of like a high-tech 'player piano'. In other words, MIDI data doesn't actually include any *sound*, but it does include the *instructions* that a MIDI-reading instrument can play back. Or for another analogy, you can think of a MIDI file like an HTML document, and the instrument that renders the MIDI file (i.e. *plays it*) is like the Web browser.

MIDI data says *what* to do (play middle C, and here's how hard to hit it, and here's how long to hold it, etc.) but it doesn't say anything at all about the actual *sound* you hear. MIDI doesn't know how to make a flute, piano, or Jimmy Hendrix guitar sound. For the actual sound, we need an instrument (a MIDI device) that can read and play a MIDI file. But the device is usually more like an *entire band or orchestra* of instruments. And that instrument might be a physical device, like the electronic keyboard synthesizers the rock musicians play, or it could even be an instrument built entirely in software, living in your computer.

For our BeatBox, we use only the built-in, software-only instrument that you get with Java. It's called a *synthesizer* (some folks refer to it as a *software synth*) because it *creates* sound. Sound that you *hear*.



but it looked so simple

First we need a Sequencer

Before we can get any sound to play, we need a Sequencer object. The sequencer is the object that takes all the MIDI data and sends it to the right instruments. It's the thing that *plays* the music. A sequencer can do a lot of different things, but in this book, we're using it strictly as a playback device. Like a CD-player on your stereo, but with a few added features. The Sequencer class is in the javax.sound.midi package (part of the standard Java library as of version 1.3). So let's start by making sure we can make (or get) a Sequencer object.

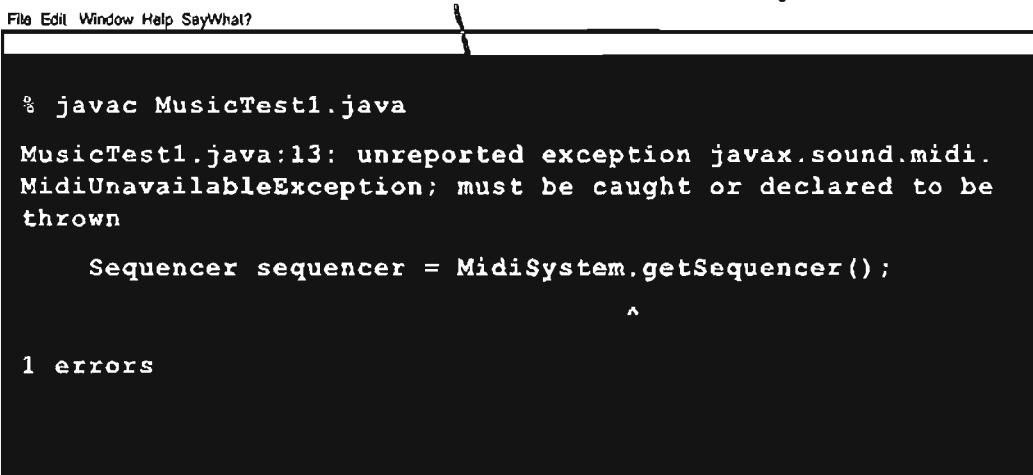
```
import javax.sound.midi.*;           ← import the javax.sound.midi package
public class MusicTest1 {
    public void play() {
        Sequencer sequencer = MidiSystem.getSequencer();
        System.out.println("We got a sequencer");
    } // close play

    public static void main(String[] args) {
        MusicTest1 mt = new MusicTest1();
        mt.play();
    } // close main
} // close class
```

We need a Sequencer object. It's the main part of the MIDI device/instrument we're using. It's the thing that, well, sequences all the MIDI information into a 'song'. But we don't make a brand new one ourselves -- we have to ask the MidiSystem to give us one.

Something's wrong!

This code won't compile! The compiler says there's an 'unreported exception' that must be caught or declared.

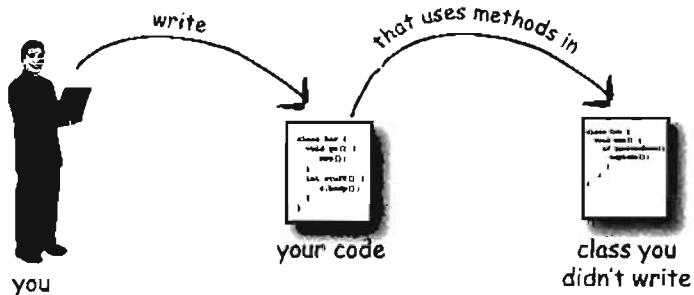


```
File Edit Window Help SayWhat? %

% javac MusicTest1.java
MusicTest1.java:13: unreported exception javax.sound.midi.MidiUnavailableException; must be caught or declared to be thrown
        Sequencer sequencer = MidiSystem.getSequencer();
                                         ^
1 errors
```

What happens when a method you want to call (probably in a class you didn't write) is risky?

- ➊ Let's say you want to call a method in a class that you didn't write.

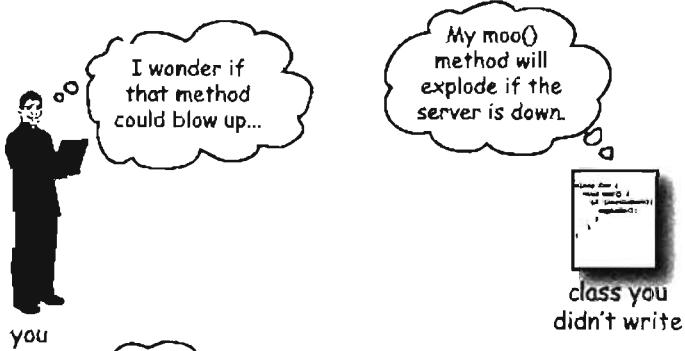


- ➋ That method does something risky, something that might not work at runtime.

 void moo() {
 if (serverDown) {
 explode();
 }
}

class you
didn't write

- ➌ You need to know that the method you're calling is risky.



- ➍ You then write code that can handle the failure if it does happen. You need to be prepared, just in case.



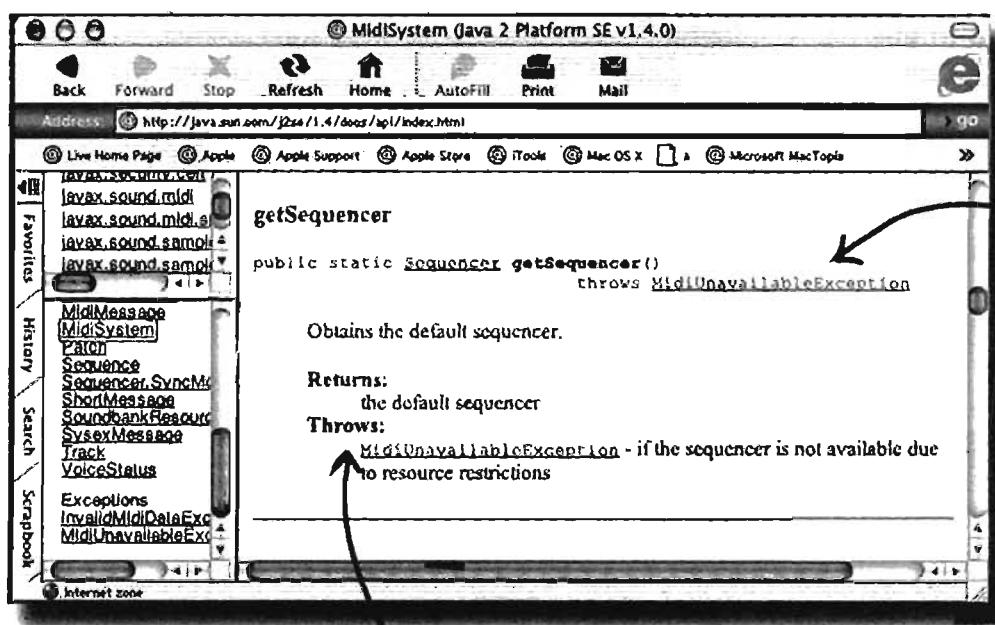
when things might go wrong

Methods in Java use exceptions to tell the calling code, "Something Bad Happened. I failed."

Java's exception-handling mechanism is a clean, well-lighted way to handle "exceptional situations" that pop up at runtime; it lets you put all your error-handling code in one easy-to-read place. It's based on you *knowing* that the method you're calling is risky (i.e. that the method *might* generate an exception), so that you can write code to deal with that possibility. If you *know* you might get an exception when you call a particular method, you can be *prepared* for—possibly even *recover* from—the problem that caused the exception.

So, how do you know if a method throws an exception? You find a `throws` clause in the risky method's declaration.

**The `getSequencer()` method takes a risk. It can fail at runtime.
So it must 'declare' the risk you take when you call it.**



The API does tell you that `getSequencer()` can throw an exception: `MidUnavailableException`. A method has to declare the exceptions it might throw.

This part tells you WHEN you might get that exception — in this case, because of resource restrictions (which could just mean the sequencer is already being used).