

# Your Brain on Java—A Learner's Guide

2nd Edition - Covers Java 5.0

# Head First Java

Learn how threads  
can change your life



Make Java concepts  
stick to your brain



Avoid embarrassing  
OO mistakes

Fool around in  
the Java Library



Bend your mind  
around 42  
Java puzzles



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)

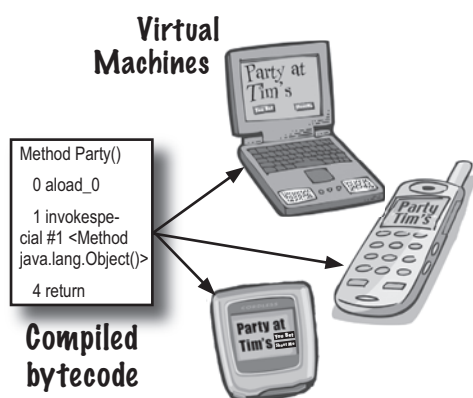
### **i** Intro

**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

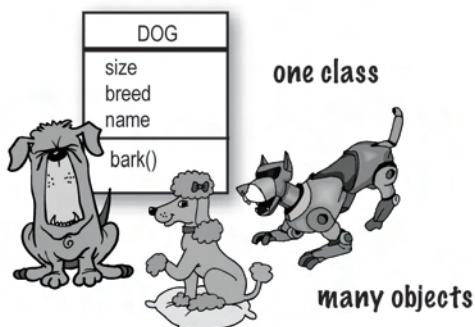
**Java takes you to new places.** From its humble release to the public as the (wimpy) version 1.02, Java seduced programmers with its friendly syntax, object-oriented features, memory management, and best of all—the promise of portability. We'll take a quick dip and write some code, compile it, and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

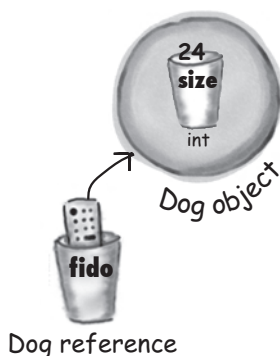


Chair Wars (Brad the OO guy vs. Larry the procedural guy)	28
Inheritance (an introduction)	31
Overriding methods (an introduction)	32
What's in a class? (methods, instance variables)	34
Making your first object	36
Using main()	38
Guessing Game code	39
Exercises and puzzles	42

## 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.

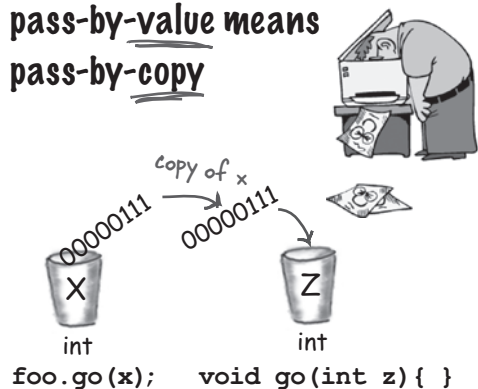


Declaring a variable (Java cares about <i>type</i> )	50
Primitive types (“I’d like a double with extra foam, please”)	51
Java keywords	53
Reference variables (remote control to an object)	54
Object declaration and assignment	55
Objects on the garbage-collectible heap	57
Arrays (a first look)	59
Exercises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

## 5 Extra-Strength Methods

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

We're gonna build the Sink a Dot Com game

A							
B	Go2.com						
C							
D			Pets.com				
E							
F							
G			AskMe.com				
	0	1	2	3	4	5	6

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?"*

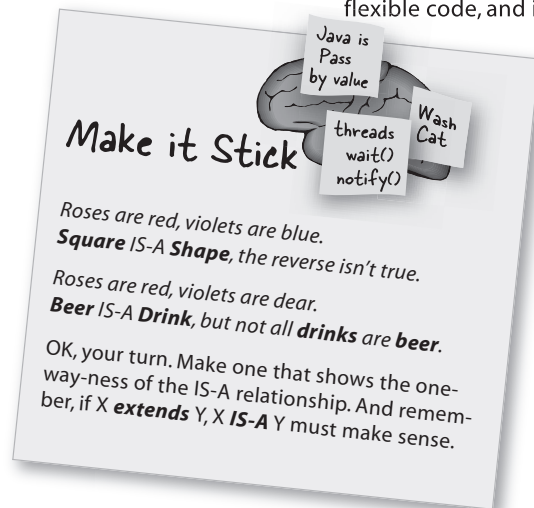
- Julia, 31, hand model



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.

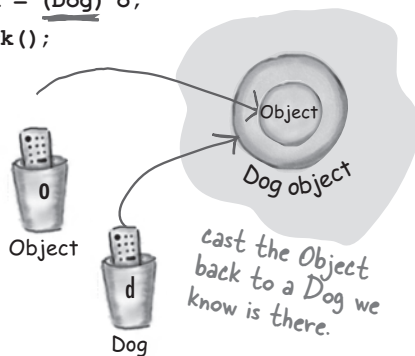


Understanding inheritance (superclass and subclass relationships)	168
Designing an inheritance tree (the Animal simulation)	170
Avoiding duplicate code (using inheritance)	171
Overriding methods	172
IS-A and HAS-A (bathtub girl)	177
What do you inherit from your superclass?	180
What does inheritance 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();
```

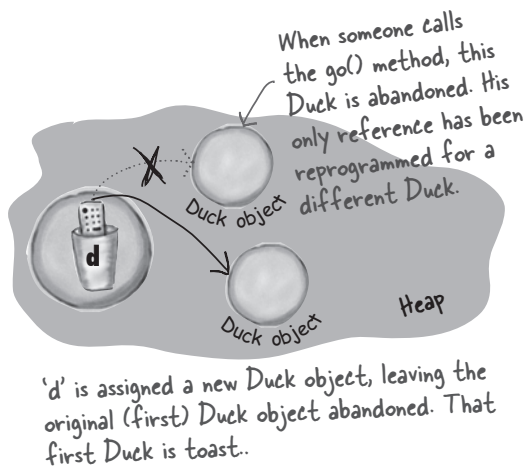


Some classes just should <i>not</i> be instantiated	200
Abstract classes ( <i>can't</i> be instantiated)	201
Abstract methods (must be implemented)	203
Polymorphism in action	206
Class Object (the ultimate superclass of <i>everything</i> )	208
Taking objects out of an ArrayList (they come out as type 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.



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

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

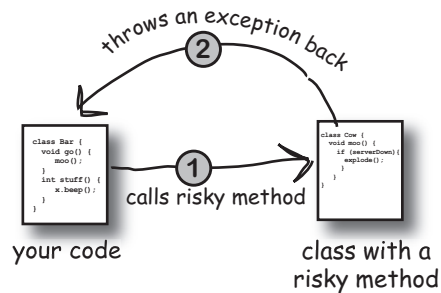
**Static variables are shared by all instances of a class.**



Math class (do you really need an instance of it?)	274
static methods	275
static variables	277
Constants (static final variables)	282
Math methods ( <code>random()</code> , <code>round()</code> , <code>abs()</code> , 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

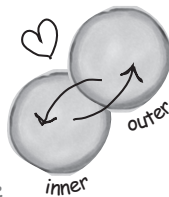
# 12 A Very Graphic Story

**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.

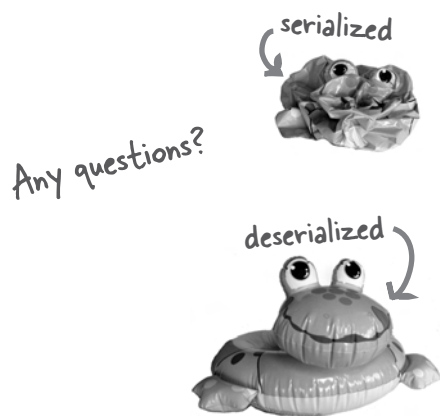
Things in the north and south get their preferred height.



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

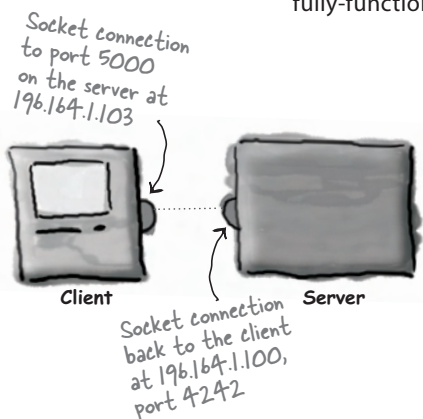
**Objects can be flattened and inflated.** Objects have state and behavior. Behavior lives in the class, but *state* lives within each individual *object*. If your program needs to save state, *you can do it the hard way*, interrogating each object, painstakingly writing the value of each instance variable. 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.



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

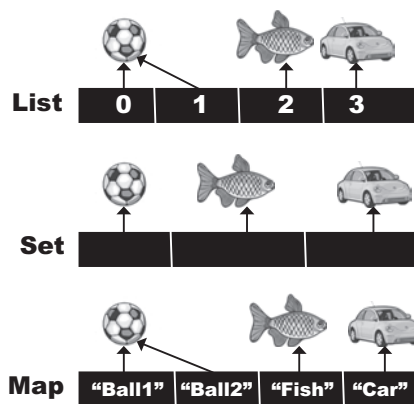
**Connect with the outside world.** It's easy. All the low-level networking details are taken care of by classes in the `java.net` library. One of Java's best features is that sending and receiving data over a network is really just I/O with a slightly different connection stream at the end of the chain. 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 <code>ChatClient</code> code	518
Ready-bake <code>SimpleChatServer</code>	520
Exercises and puzzles	524

# 16 Data Structures

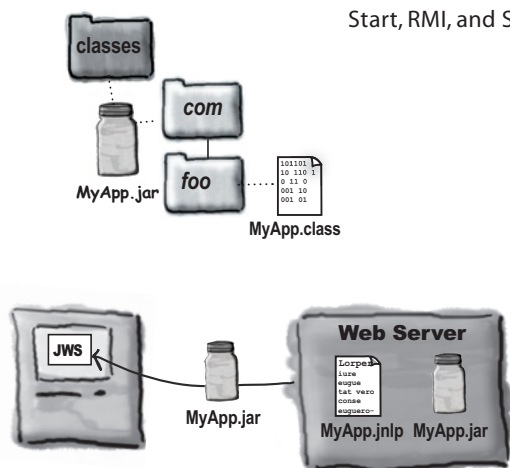
**Sorting is a snap in Java.** You have all the tools for collecting and manipulating your data without having to write your own sort algorithms. The Java Collections Framework has a data structure that should work for virtually anything you'll ever need to do. Want to keep a list that you can easily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

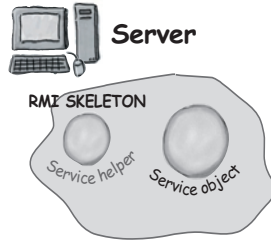
**It's time to let go.** You wrote your code. You tested your code. You refined your code. You told everyone you know that if you never saw a line of code again, that'd be fine. But in the end, you've created a work of art. The thing actually runs! But now what? In these final two chapters, we'll explore 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

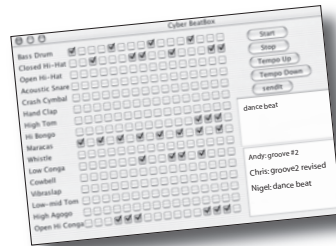
**Being remote doesn't have to be a bad thing.** Sure, things *are* easier when all the parts of your application are in one place, in one heap, with one JVM to rule them all. But that's not always possible. Or desirable. What if your application handles powerful computations? What if your app needs 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
--------------	-----

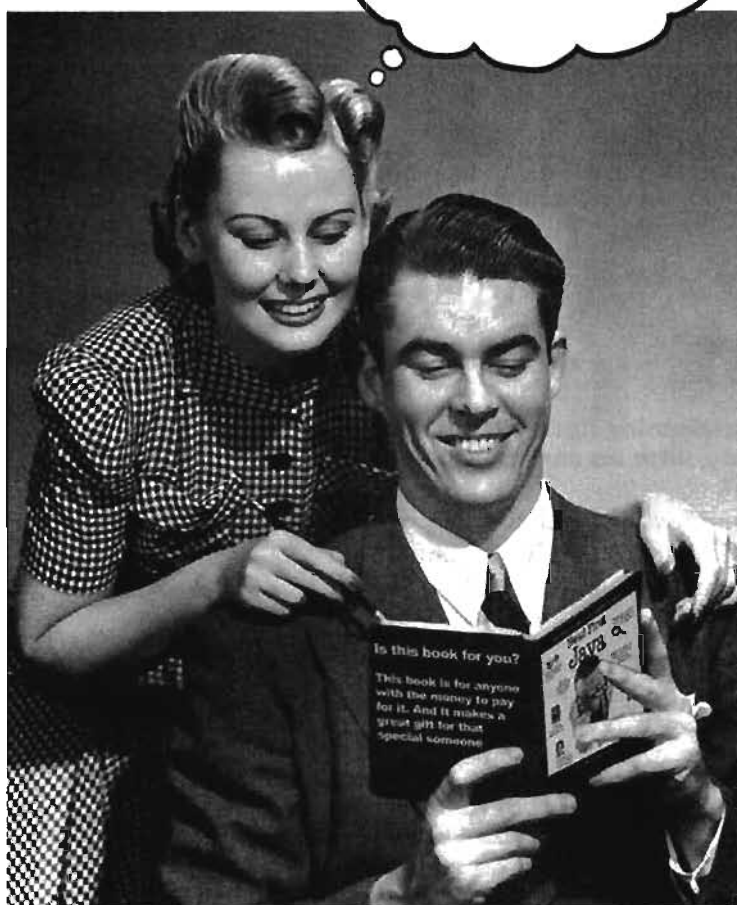
## i Index

	677
--	-----

how to use this book

# Intro

I can't believe they  
put *that* in a Java  
programming book!



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

how to use this 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... -FredJ*

## We know what you're thinking.

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

"What's with all the graphics?"

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

"Do I smell pizza?"



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

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

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

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

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

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

And that's how your brain knows...

### **This must be important! Don't forget it!**

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

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

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



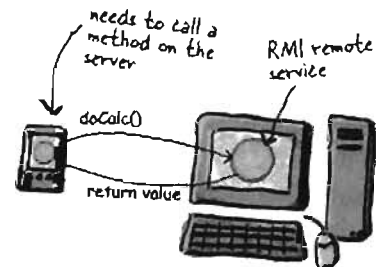
## We think of a “Head First Java” reader as a learner.

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

### Some of the Head First learning principles:



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



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

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



No method body!  
End it with a semicolon

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

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

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



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

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





## Metacognition: thinking about thinking.

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

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

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

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

The trick is to get your brain to see the new material you're learning as Really Important. Crucial to your well-being. As important as a tiger. Otherwise, you're in for a constant battle, with your brain doing its best to keep the new content from sticking.

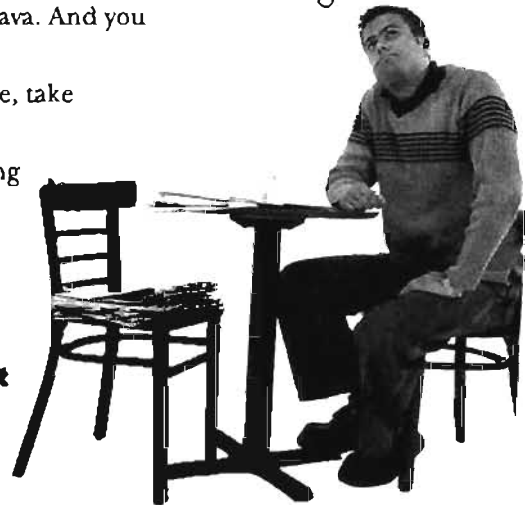
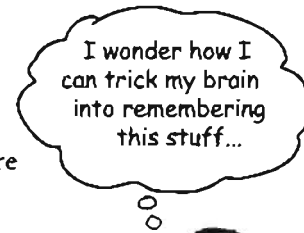
### So just how **DO** you get your brain to treat Java like it was a hungry tiger?

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

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

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

But pictures and conversational style are just the beginning.



## how to use this book

### 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 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.

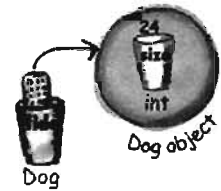
We used *multiple learning styles*, because *you* might prefer step-by-step procedures, while someone else wants to understand the big picture first, while someone else just wants to see a code example. But regardless of your own learning preference, *everyone* benefits from seeing the same content represented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

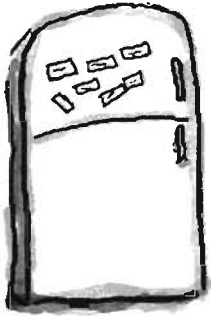
We include content for *both sides of your brain*, because the more of your brain you engage, the more likely you are to learn and remember, and the longer you can stay focused. Since working one side of the brain often means giving the other side a chance to rest, you can be more productive at learning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



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

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

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

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

### 3 Read the "There are No Dumb Questions"

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

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

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

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

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

### 6 Drink water. Lots of it.

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

### 7 Talk about it. Out loud.

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

### 8 Listen to your brain.

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

### 9 Feel something!

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

### 10 Type and run the code.

Type and run the code examples. Then you can experiment with changing and improving the code (or breaking it, which is sometimes the best way to figure out what's really happening). For long examples or Ready-bake code, you can download the source files from [headfirstjava.com](http://headfirstjava.com)

## how to use this book

# 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.

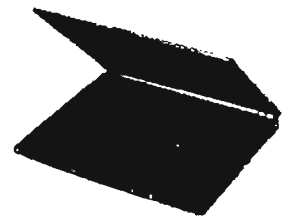
### SETTING UP JAVA

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

Note: much of the code from this book is available at [wickedlysmart.com](http://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.

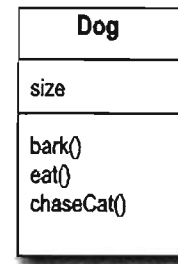
### The 'Sharpen Your Pencil' exercises don't have answers.

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](http://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 crossword puzzles, you won't like these either.

