

Appendix B

The Top Ten Topics that almost made it into the Real Book...



We covered a lot of ground, and you're almost finished with this book. We'll miss you, but before we let you go, we wouldn't feel right about sending you out into JavaLand without a little more preparation. We can't possibly fit everything you'll need to know into this relatively small appendix. Actually, we *did* originally include everything you need to know about Java (not already covered by the other chapters), by reducing the type point size to .00003. It all fit, but nobody could read it. So, we threw most of it away, but kept the best bits for this Top Ten appendix.

This really *is* the end of the book. Except for the index (a must-read!).

#10 Bit Manipulation

Why do you care?

We've talked about the fact that there are 8 bits in a byte, 16 bits in a short, and so on. You might have occasion to turn individual bits on or off. For instance you might find yourself writing code for your new Java enabled toaster, and realize that due to severe memory limitations, certain toaster settings are controlled at the bit level. For easier reading, we're showing only the last 8 bits in the comments rather than the full 32 for an `int`).

Bitwise NOT Operator: `~`

This operator 'flips all the bits' of a primitive.

```
int x = 10;    // bits are 00001010
x = ~x;       // bits are now 11110101
```

The next three operators compare two primitives on a bit by bit basis, and return a result based on comparing these bits. We'll use the following example for the next three operators:

```
int x = 10;    // bits are 00001010
int y = 6;     // bits are 00000110
```

Bitwise AND Operator: `&`

This operator returns a value whose bits are turned on only if *both* original bits are turned on:

```
int a = x & y; // bits are 00000010
```

Bitwise OR Operator: `|`

This operator returns a value whose bits are turned on only if *either* of the original bits are turned on:

```
int a = x | y; // bits are 00001110
```

Bitwise XOR (exclusive OR) Operator: `^`

This operator returns a value whose bits are turned on only if *exactly one* of the original bits are turned on:

```
int a = x ^ y; // bits are 00001100
```

The Shift Operators

These operators take a single integer primitive and shift (or slide) all of its bits in one direction or another. If you want to dust off your binary math skills, you might realize that shifting bits *left* effectively *multiplies* a number by a power of two, and shifting bits *right* effectively *divides* a number by a power of two.

We'll use the following example for the next three operators:

```
int x = -11;    // bits are 11110101
```

Ok, ok, we've been putting it off, here is the world's shortest explanation of storing negative numbers, and *two's complement*. Remember, the leftmost bit of an integer number is called the **sign bit**. A negative integer number in Java *always* has its sign bit turned *on* (i.e. set to 1). A positive integer number always has its sign bit turned *off* (0). Java uses the *two's complement* formula to store negative numbers. To change a number's sign using two's complement, flip all the bits, then add 1 (with a byte, for example, that would mean adding 00000001 to the flipped value).

Right Shift Operator: `>>`

This operator shifts all of a number's bits right by a certain number, and fills all of the bits on the left side with whatever the original leftmost bit was. **The sign bit does *not* change:**

```
int y = x >> 2; // bits are 11111101
```

Unsigned Right Shift Operator: `>>>`

Just like the right shift operator BUT it ALWAYS fills the leftmost bits with zeros. **The sign bit *might* change:**

```
int y = x >>> 2; // bits are 00111101
```

Left Shift Operator: `<<`

Just like the unsigned right shift operator, but in the other direction; the rightmost bits are filled with zeros. **The sign bit *might* change.**

```
int y = x << 2; // bits are 11010100
```

#9 Immutability

Why do you care that Strings are Immutable?

When your Java programs start to get big, you'll inevitably end up with lots and lots of String objects. For security purposes, and for the sake of conserving memory (remember your Java programs can run on teeny Java-enabled cell phones), Strings in Java are immutable. What this means is that when you say:

```
String s = "0";

for (int x = 1; x < 10; x++) {
    s = s + x;
}
```

What's actually happening is that you're creating ten String objects (with values "0", "01", "012", through "0123456789"). In the end *s* is referring to the String with the value "0123456789", but at this point there are *ten* Strings in existence!

Whenever you make a new String, the JVM puts it into a special part of memory called the 'String Pool' (sounds refreshing doesn't it?). If there is already a String in the String Pool with the same value, the JVM doesn't create a duplicate, it simply refers your reference variable to the existing entry. The JVM can get away with this because Strings are immutable; one reference variable can't change a String's value out from under another reference variable referring to the same String.

The other issue with the String pool is that the Garbage Collector *doesn't go there*. So in our example, unless by coincidence you later happen to make a String called "01234", for instance, the first nine Strings created in our *for* loop will just sit around wasting memory.

How does this save memory?

Well, if you're not careful, *it doesn't!* But if you understand how String immutability works, then you can sometimes take advantage of it to save memory. If you have to do a lot of String manipulations (like concatenations, etc.), however, there is another class `StringBuilder`, better suited for that purpose. We'll talk more about `StringBuilder` in a few pages.

Why do you care that Wrappers are Immutable?

In the Math chapter we talked about the two main uses of the wrapper classes:

- Wrapping a primitive so it can pretend to be an object.
- Using the static utility methods (for example, `Integer.parseInt()`).

It's important to remember that when you create a wrapper object like:

```
Integer iWrap = new Integer(42);
```

That's it for that wrapper object. Its value will *always* be 42. **There is no setter method for a wrapper object.** You can, of course, refer *iWrap* to a *different* wrapper object, but then you'll have *two* objects. Once you create a wrapper object, there's no way to change the *value* of that object!



#8 Assertions

We haven't talked much about how to debug your Java program while you're developing it. We believe that you should learn Java at the command line, as we've been doing throughout the book. Once you're a Java pro, if you decide to use an IDE*, you might have other debugging tools to use. In the old days, when a Java programmer wanted to debug her code, she'd stick a bunch of `System.out.println()` statements throughout the program, printing current variable values, and "I got here" messages, to see if the flow control was working properly. (The ready-bake code in chapter 6 left some debugging 'print' statements in the code.) Then, once the program was working correctly, she'd go through and take all those `System.out.println()` statements back out again. It was tedious and error prone. But as of Java 1.4 (and 5.0), debugging got a whole lot easier. The answer?

Assertions

Assertions are like `System.out.println()` statements on steroids. Add them to your code as you would add `println` statements. The Java 5.0 compiler assumes you'll be compiling source files that are 5.0 compatible, so as of Java 5.0, compiling with assertions is enabled by default.

At runtime, if you do nothing, the `assert` statements you added to your code will be ignored by the JVM, and won't slow down your program. But if you tell the JVM to *enable* your assertions, they will help you do your debugging, without changing a line of code!

Some folks have complained about having to leave `assert` statements in their production code, but leaving them in can be really valuable when your code is already deployed in the field. If your client is having trouble, you can instruct the client to run the program with assertions enabled, and have the client send you the output. If the assertions were stripped out of your deployed code, you'd never have that option. And there is almost no downside; when assertions are not enabled, they are completely ignored by the JVM, so there's no performance hit to worry about.

How to make Assertions work

Add assertion statements to your code wherever you believe that something *must be true*. For instance:

```
assert (height > 0);

// if true, program continues normally
// if false, throw an AssertionError
```

You can add a little more information to the stack trace by saying:

```
assert (height > 0) : "height = " +
height + " weight = " + weight;
```

The expression after the colon can be any legal Java expression *that resolves to a non-null value*. But whatever you do, **don't create assertions that change an object's state!** If you do, enabling assertions at runtime might change how your program performs.

Compiling and running with Assertions

To *compile* with assertions:

```
javac TestDriveGame.java
```

(Notice that no command line options were necessary.)

To *run* with assertions:

```
java -ea TestDriveGame
```

* IDE stands for Integrated Development Environment and includes tools such as Eclipse, Borland's JBuilder, or the open source NetBeans (netbeans.org).

#7 Block Scope

In chapter 9, we talked about how local variables live only as long as the method in which they're declared stays on the stack. But some variables can have even *shorter* lifespans. Inside of methods, we often create *blocks* of code. We've been doing this all along, but we haven't explicitly *talked* in terms of *blocks*. Typically, blocks of code occur within methods, and are bounded by curly braces { }. Some common examples of code blocks that you'll recognize include loops (*for*, *while*) and conditional expressions (like *if* statements).

Let's look at an example:

```
void doStuff() {
    int x = 0;
    for(int y = 0; y < 5; y++) {
        x = x + y;
    }
    x = x * y;
}
```

Annotations for the code above:

- `{` ← start of the method block
- `int x = 0;` ← local variable scoped to the entire method
- `for(int y = 0; y < 5; y++) {` ← beginning of a for loop block, and y is scoped to only the for loop!
- `x = x + y;` ← No problem, x and y are both in scope
- `}` ← end of the for loop block
- `x = x * y;` ← Aaack! Won't compile! y is out of scope here! (this is not the way it works in some other languages, so beware!)
- `}` ← end of the method block, now x is also out of scope

In the previous example, *y* was a block variable, declared inside a block, and *y* went out of scope as soon as the for loop ended. Your Java programs will be more debuggable and expandable if you use local variables instead of instance variables, and block variables instead of local variables, whenever possible. The compiler will make sure that you don't try to use a variable that's gone out of scope, so you don't have to worry about runtime meltdowns.

#6 Linked Invocations

While you did see a little of this in this book, we tried to keep our syntax as clean and readable as possible. There are, however, many legal shortcuts in Java, that you'll no doubt be exposed to, especially if you have to read a lot of code you didn't write. One of the more common constructs you will encounter is known as *linked invocations*. For example:

```
StringBuffer sb = new StringBuffer("spring");
sb = sb.delete(3,6).insert(2,"umme").deleteCharAt(1);
System.out.println("sb = " + sb);
// result is sb = summer
```

What in the world is happening in the second line of code? Admittedly, this is a contrived example, but you need to learn how to decipher these.

1 - Work from left to right.

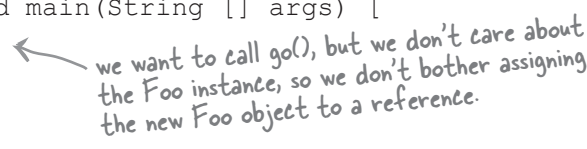
2 - Find the result of the leftmost method call, in this case `sb.delete(3,6)`. If you look up `StringBuffer` in the API docs, you'll see that the `delete()` method returns a `StringBuffer` object. The result of running the `delete()` method is a `StringBuffer` object with the value "spr".

3 - The next leftmost method (`insert()`) is called on the newly created `StringBuffer` object "spr". The result of that method call (the `insert()` method), is *also* a `StringBuffer` object (although it doesn't have to be the same type as the previous method return), and so it goes, the returned object is used to call the next method to the right. In theory, you can link as many methods as you want in a single statement (although it's rare to see more than three linked methods in a single statement). Without linking, the second line of code from above would be more readable, and look something like this:

```
sb = sb.delete(3,6);
sb = sb.insert(2,"umme");
sb = sb.deleteCharAt(1);
```

But here's a more common, and useful example, that you saw us using, but we thought we'd point it out again here. This is for when your `main()` method needs to invoke an instance method of the main class, but you don't need to keep a *reference* to the instance of the class. In other words, the `main()` needs to create the instance *only* so that `main()` can invoke one of the instance's *methods*.

```
class Foo {
    public static void main(String [] args) {
        new Foo().go();
    }
    void go() {
        // here's what we REALLY want...
    }
}
```



#5 Anonymous and Static Nested Classes

Nested classes come in many flavors

In the GUI event-handling section of the book, we started using inner (nested) classes as a solution for implementing listener interfaces. That's the most common, practical, and readable form of an inner class—where the class is simply nested within the curly braces of another enclosing class. And remember, it means you need an instance of the outer class in order to get an instance of the inner class, because the inner class is a *member* of the outer/enclosing class.

But there are other kinds of inner classes including *static* and *anonymous*. We're not going into the details here, but we don't want you to be thrown by strange syntax when you see it in someone's code. Because out of virtually anything you can do with the Java language, perhaps nothing produces more bizarre-looking code than anonymous inner classes. But we'll start with something simpler—static nested classes.

Static nested classes

You already know what static means—something tied to the class, not a particular instance. A static nested class looks just like the non-static classes we used for event listeners, except they're marked with the keyword `static`.

```
public class FooOuter {
    static class BarInner {
        void sayIt() {
            System.out.println("method of a static inner class");
        }
    }
}
```

A static nested class is just that—a class enclosed within another, and marked with the static modifier.

```
class Test {
    public static void main (String[] args) {
        FooOuter.BarInner foo = new FooOuter.BarInner();
        foo.sayIt();
    }
}
```

Because a static nested class is...static, you don't use an instance of the outer class. You just use the name of the class, the same way you invoke static methods or access static variables.

Static nested classes are more like regular non-nested classes in that they don't enjoy a special relationship with an enclosing outer object. But because static nested classes are still considered a *member* of the enclosing/outer class, they still get access to any private members of the outer class... but *only the ones that are also static*. Since the static nested class isn't connected to an instance of the outer class, it doesn't have any special way to access the non-static (instance) variables and methods.

when arrays aren't enough

#5 Anonymous and Static Nested Classes, continued

The difference between *nested* and *inner*

Any Java class that's defined within the scope of another class is known as a *nested* class. It doesn't matter if it's anonymous, static, normal, whatever. If it's inside another class, it's technically considered a *nested* class. But *non-static* nested classes are often referred to as *inner* classes, which is what we called them earlier in the book. The bottom line: all inner classes are nested classes, but not all nested classes are inner classes.

Anonymous inner classes

Imagine you're writing some GUI code, and suddenly realize that you need an instance of a class that implements `ActionListener`. But you realize you don't *have* an instance of an `ActionListener`. Then you realize that you also never wrote a *class* for that listener. You have two choices at that point:

1) Write an inner class in your code, the way we did in our GUI code, and then instantiate it and pass that instance into the button's event registration (`addActionListener()`) method.

OR

2) Create an *anonymous* inner class and instantiate it, right there, just-in-time. **Literally right where you are at the point you need the listener object.** That's right, you create the class and the instance in the place where you'd normally be supplying just the instance. Think about that for a moment—it means you pass the entire *class* where you'd normally pass only an *instance* into a method argument!

```
import java.awt.event.*;
import javax.swing.*;
public class TestAnon {
    public static void main (String[] args) {

        JFrame frame = new JFrame();
        JButton button = new JButton("click");
        frame.getContentPane().add(button);
        // button.addActionListener(quitListener);
```

We made a frame and added a button, and now we need to register an action listener with the button. Except we never made a class that implements the `ActionListener` interface...

Normally we'd do something like this—passing in a reference to an instance of an inner class... an inner class that implements `ActionListener` (and the `actionPerformed()` method).

This statement:

```
button.addActionListener (new ActionListener() {
    public void actionPerformed(ActionEvent ev) {
        System.exit(0);
    }
});
```

ends down here!

Notice that we say "`new ActionListener()`" even though `ActionListener` is an interface and so you can't **MAKE** an instance of it! But this syntax really means, "create a new class (with no name) that implements the `ActionListener` interface, and by the way, here's the implementation of the interface methods `actionPerformed()`."

But now instead of passing in an object reference, we pass in... the whole new class definition!! In other words, we write the class that implements `ActionListener` **RIGHT HERE WHERE WE NEED IT**. The syntax also creates an instance of the class automatically.

#4 Access Levels and Access Modifiers (Who Sees What)

Java has *four* access *levels* and *three* access *modifiers*. There are only *three* modifiers because the *default* (what you get when you don't use any access modifier) is one of the four access levels.

Access Levels (in order of how restrictive they are, from least to most restrictive)

- public* ← public means any code anywhere can access the public thing (by 'thing' we mean class, variable, method, constructor, etc.).
- protected* ← protected works just like default (code in the same package has access), EXCEPT it also allows subclasses outside the package to inherit the protected thing.
- default* ← default access means that only code within the same package as the class with the default thing can access the default thing.
- private* ← private means that only code within the same class can access the private thing. Keep in mind it means private to the class, not private to the object. One Dog can see another Dog object's private stuff, but a Cat can't see a Dog's privates.

Access modifiers

```
public
protected
private
```

Most of the time you'll use only public and private access levels.

public

Use public for classes, constants (static final variables), and methods that you're exposing to other code (for example getters and setters) and most constructors.

private

Use private for virtually all instance variables, and for methods that you don't want outside code to call (in other words, methods *used* by the public methods of your class).

But although you might not use the other two (protected and default), you still need to know what they do because you'll see them in other code.

when arrays aren't enough

#4 Access Levels and Access Modifiers, cont.

default and protected

default

Both protected and default access levels are tied to packages. Default access is simple—it means that only code *within the same package* can access code with default access. So a default class, for example (which means a class that isn't explicitly declared as *public*) can be accessed by only classes within the same package as the default class.

But what does it really mean to *access* a class? Code that does not have access to a class is not allowed to even *think* about the class. And by think, we mean *use* the class in code. For example, if you don't have access to a class, because of access restriction, you aren't allowed to instantiate the class or even declare it as a type for a variable, argument, or return value. You simply can't type it into your code at all! If you do, the compiler will complain.

Think about the implications—a default class with public methods means the public methods aren't really public at all. You can't access a method if you can't *see* the class.

Why would anyone want to restrict access to code within the same package? Typically, packages are designed as a group of classes that work together as a related set. So it might make sense that classes within the same package need to access one another's code, while as a package, only a small number of classes and methods are exposed to the outside world (i.e. code outside that package).

OK, that's default. It's simple—if something has default access (which, remember, means no explicit access modifier!), only code within the same package as the default *thing* (class, variable, method, inner class) can access that *thing*.

Then what's *protected* for?

protected

Protected access is almost identical to default access, with one exception: it allows subclasses to *inherit* the protected thing, *even if those subclasses are outside the package of the superclass they extend*. That's it. That's *all* protected buys you—the ability to let your subclasses be outside your superclass package, yet still *inherit* pieces of the class, including methods and constructors.

Many developers find very little reason to use protected, but it is used in some designs, and some day you might find it to be exactly what you need. One of the interesting things about protected is that—unlike the other access levels—protected access applies only to *inheritance*. If a subclass-outside-the-package has a *reference* to an instance of the superclass (the superclass that has, say, a protected method), the subclass can't access the protected method using that superclass reference! The only way the subclass can access that method is by *inheriting* it. In other words, the subclass-outside-the-package doesn't have *access* to the protected method, it just *has* the method, through inheritance.

#3 String and StringBuffer/StringBuilder Methods

Two of the most commonly used classes in the Java API are String and StringBuffer (remember from #9 a few pages back, Strings are immutable, so a StringBuffer/StringBuilder can be a lot more efficient if you're manipulating a String). As of Java 5.0 you should use the String**Builder** class instead of String**Buffer**, unless your String manipulations need to be thread-safe, which is not common. Here's a brief overview of the **key** methods in these classes:

Both String and StringBuffer/StringBuilder classes have:

```
char charAt(int index);           // what char is at a certain position
int length();                     // how long is this
String substring(int start, int end); // get a part of this
String toString();                // what's the String value of this
```

To concatenate Strings:

```
String concat(String);            // for the String class
String append(String);            // for StringBuffer & StringBuilder
```

The String class has:

```
String replace(char old, char new); // replace all occurrences of a char
String substring(int begin, int end); // get a portion of a String
char [] toCharArray();               // convert to an array of chars
String toLowerCase();                // convert all characters to lower case
String toUpperCase();                // convert all characters to upper case
String trim();                       // remove whitespace from the ends
String valueOf(char []);              // make a String out of a char array
String valueOf(int i);                // make a String out of a primitive
// other primitives are supported as well
```

The StringBuffer & StringBuilder classes have:

```
StringBxxxx delete(int start, int end); // delete a portion
StringBxxxx insert(int offset, any primitive or a char []); // insert something
StringBxxxx replace(int start, int end, String s); // replace this part with this String
StringBxxxx reverse(); // reverse the SB from front to back
void setCharAt(int index, char ch); // replace a given character
```

Note: StringBxxxx refers to either StringBuffer or StringBuilder, as appropriate.

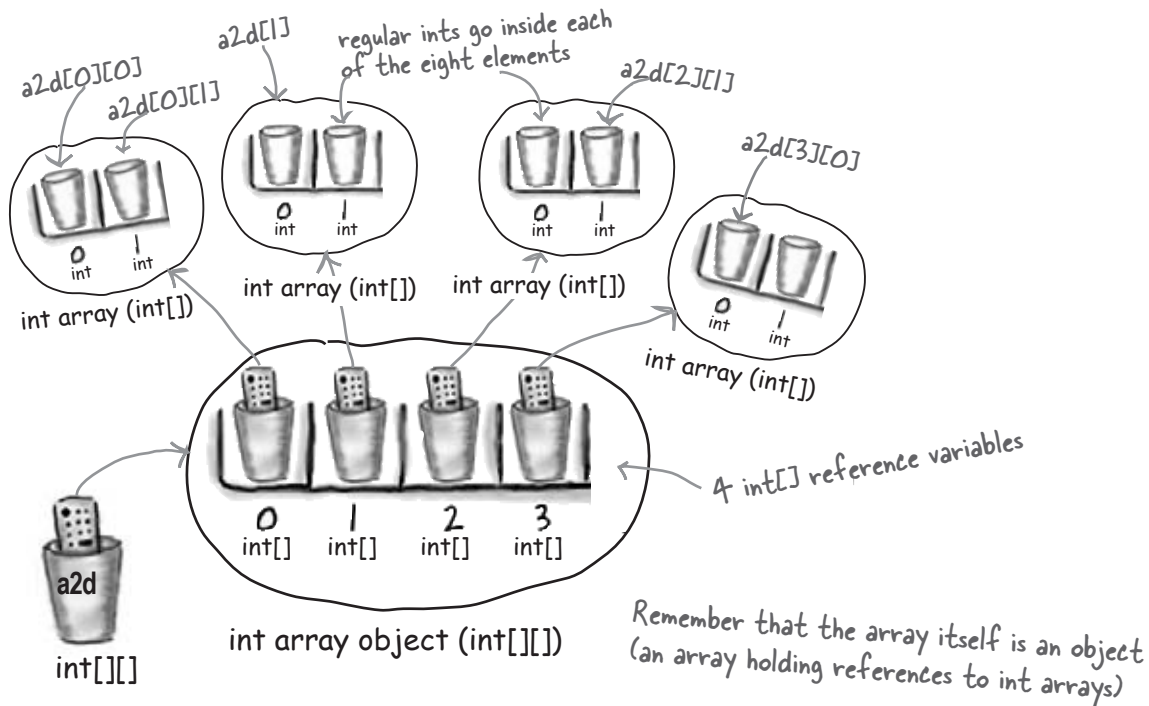
when arrays aren't enough

#2 Multidimensional Arrays

In most languages, if you create, say, a 4 x 2 two-dimensional array, you would visualize a rectangle, 4 elements by 2 elements, with a total of 8 elements. But in Java, such an array would actually be 5 arrays linked together! In Java, a two dimensional array is simply *an array of arrays*. (A three dimensional array is an array of arrays of arrays, but we'll leave that for you to play with.) Here's how it works

```
int[][] a2d = new int [4][2];
```

The JVM creates an array with 4 elements. *Each* of these four elements is actually a reference variable referring to a (newly created), int array with 2 elements.



Working with multidimensional arrays

- To access the second element in the third array: `int x = a2d[2][1];` // remember, 0 based!
- To make a one-dimensional reference to one of the sub-arrays: `int[] copy = a2d[1];`
- Short-cut initialization of a 2 x 3 array: `int[][] x = { { 2,3,4 }, { 7,8,9 } };`
- To make a 2d array with irregular dimensions:

```
int[][] y = new int [2][]; // makes only the first array, with a length of 2
y[0] = new int [3]; // makes the first sub-array 3 elements in length
y[1] = new int [5]; // makes the second sub-array 5 elements in length
```

And the number one topic that didn't quite make it in...

#1 Enumerations (also called Enumerated Types or Enums)

We've talked about constants that are defined in the API, for instance, `JFrame.EXIT_ON_CLOSE`. You can also create your own constants by marking a variable `static final`. But sometimes you'll want to create a set of constant values to represent the *only* valid values for a variable. This set of valid values is commonly referred to as an *enumeration*. Before Java 5.0 you could only do a half-baked job of creating an enumeration in Java. As of Java 5.0 you can create full fledged enumerations that will be the envy of all your pre-Java 5.0-using friends.

Who's in the band?

Let's say that you're creating a website for your favorite band, and you want to make sure that all of the comments are directed to a particular band member.

The old way to fake an "enum":

```
public static final int JERRY = 1;
public static final int BOBBY = 2;
public static final int PHIL = 3;
```

```
// later in the code
```

```
if (selectedBandMember == JERRY) {
    // do JERRY related stuff
}
```

← We're hoping that by the time we got here
"selectedBandMember" has a valid value!

The good news about this technique is that it DOES make the code easier to read. The other good news is that you can't ever change the value of the fake enums you've created; JERRY will always be 1. The bad news is that there's no easy or good way to make sure that the value of `selectedBandMember` will always be 1, 2, or 3. If some hard to find piece of code sets `selectedBandMember` equal to 812, it's pretty likely your code will break...

when arrays aren't enough

#1 Enumerations, cont.

The same situation using a genuine Java 5.0 enum. While this is a very basic enumeration, most enumerations usually *are* this simple.

A new, official “enum”:

```
public enum Members { JERRY, BOBBY, PHIL };  
public Members selectedBandMember;
```

// later in the code

```
if (selectedBandMember == Members.JERRY) {  
    // do JERRY-related stuff  
}
```

No need to worry about this variable's value!

The syntax to refer to an enum “instance”.

This kind of looks like a simple class definition doesn't it? It turns out that enums ARE a special kind of class. Here we've created a new enumerated type called “Members”.

The “selectedBandMember” variable is of type “Members”, and can ONLY have a value of “JERRY”, “BOBBY”, or “PHIL”.

Your enum extends java.lang.Enum

When you create an enum, you're creating a new class, and *you're implicitly extending java.lang.Enum*. You can declare an enum as its own standalone class, in its own source file, or as a member of another class.

Using “if” and “switch” with Enums

Using the enum we just created, we can perform branches in our code using either the if or switch statement. Also notice that we can compare enum instances using either == or the .equals() method. Usually == is considered better style.

```
Members n = Members.BOBMY;   
if (n.equals(Members.JERRY)) System.out.println("Jerrrry!");  
if (n == Members.BOBMY) System.out.println("Rat Dog");
```

Assigning an enum value to a variable.

Both of these work fine!
“Rat Dog” is printed.

```
Members ifName = Members.PHIL;  
switch (ifName) {  
    case JERRY: System.out.print("make it sing ");  
    case PHIL: System.out.print("go deep ");  
    case BOBBY: System.out.println("Cassidy! ");  
}
```

Pop Quiz! What's the output?

Answer: go deep Cassidy!

#1 Enumerations, completed

A really tricked-out version of a similar enum

You can add a bunch of things to your enum like a constructor, methods, variables, and something called a constant-specific class body. They're not common, but you might run into them:

```
public class HfjEnum {
    enum Names {
        JERRY("lead guitar") { public String sings() {
                                return "plaintively"; }
                                },
        BOBBY("rhythm guitar") { public String sings() {
                                return "hoarsely"; }
                                },
        PHIL("bass");

        private String instrument;

        Names(String instrument) {
            this.instrument = instrument;
        }
        public String getInstrument() {
            return this.instrument;
        }
        public String sings() {
            return "occasionally";
        }
    }

    public static void main(String [] args) {
        for (Names n : Names.values()) {
            System.out.print(n);
            System.out.print(", instrument: " + n.getInstrument());
            System.out.println(", sings: " + n.sings());
        }
    }
}
```

This is an argument passed in to the constructor declared below.

These are the so-called "constant-specific class bodies". Think of them as overriding the basic enum method (in this case the "sing()" method), if sing() is called on a variable with an enum value of JERRY or BOBBY.

This is the enum's constructor. It runs once for each declared enum value (in this case it runs three times).

You'll see these methods being called from "main()".

Every enum comes with a built-in "values()" method which is typically used in a "for" loop as shown.

```
File Edit Window Help Bootleg
%java HfjEnum

JERRY, instrument: lead guitar, sings: plaintively
BOBBY, instrument: rhythm guitar, sings: hoarsely
PHIL, instrument: bass, sings: occasionally
%
```

Notice that the basic "sing()" method is only called when the enum value has no constant-specific class body.

when arrays aren't enough



Five-Minute Mystery

A Long Trip Home



Captain Byte of the Flatland starship “Traverser” had received an urgent, Top Secret transmission from headquarters. The message contained 30 heavily encrypted navigational codes that the Traverser would need to successfully plot a course home through enemy sectors. The enemy Hackarians, from a neighboring galaxy, had devised a devilish code-scrambling ray that was capable of creating bogus objects on the heap of the Traverser’s only navigational computer. In addition, the alien ray could alter valid reference variables so that they referred to these bogus objects. The only defense the Traverser crew had against this evil Hackarian ray was to run an inline virus checker which could be imbedded into the Traverser’s state of the art Java 1.4 code.

Captain Byte gave Ensign Smith the following programming instructions to process the critical navigational codes:

“Put the first five codes in an array of type ParsecKey. Put the last 25 codes in a five by five, two dimensional array of type QuadrantKey. Pass these two arrays into the plotCourse() method of the public final class ShipNavigation. Once the course object is returned run the inline virus checker against all the programs reference variables and then run the NavSim program and bring me the results.”

A few minutes later Ensign Smith returned with the NavSim output. “NavSim output ready for review, sir”, declared Ensign Smith. “Fine”, replied the Captain, “Please review your work”. “Yes sir!”, responded the Ensign, “First I declared and constructed an array of type ParsecKey with the following code; `ParsecKey [] p = new ParsecKey[5];`, next I declared and constructed an array of type QuadrantKey with the following code: `QuadrantKey [][] q = new QuadrantKey [5] [5];`. Next, I loaded the first 5 codes into the ParsecKey array using a ‘for’ loop, and then I loaded the last 25 codes into the QuadrantKey array using nested ‘for’ loops. Next, I ran the virus checker against all 32 reference variables, 1 for the ParsecKey array, and 5 for its elements, 1 for the QuadrantKey array, and 25 for its elements. Once the virus check returned with no viruses detected, I ran the NavSim program and re-ran the virus checker, just to be safe... Sir ! “

Captain Byte gave the Ensign a cool, long stare and said calmly, “Ensign, you are confined to quarters for endangering the safety of this ship, I don’t want to see your face on this bridge again until you have properly learned your Java! Lieutenant Boolean, take over for the Ensign and do this job correctly!”

Why did the captain confine the Ensign to his quarters?



Five-Minute Mystery Solution



A Long Trip Home

Captain Byte knew that in Java, multidimensional arrays are actually arrays of arrays. The five by five `QuadrantKey` array 'q', would actually need a total of 31 reference variables to be able to access all of its components:

1 - reference variable for 'q'

5 - reference variables for `q[0]` - `q[4]`

25 - reference variables for `q[0][0]` - `q[4][4]`

The ensign had forgotten the reference variables for the five one dimensional arrays embedded in the 'q' array. Any of those five reference variables could have been corrupted by the Hackarian ray, and the ensign's test would never reveal the problem.

Index

Symbols

&, &&, |, || (boolean operators) 151, 660
 &, <<, >>, >>>, ^, |, ~ (bitwise operators) 660
 ++ – (increment/decrement) 105, 115
 + (String concatenation operator) 17
 . (dot operator) 36
 reference 54
 <, <=, ==, !=, >, >= (comparison operators) 86, 114, 151
 <, <=, ==, >, >= (comparison operators) 11

A

abandoned objects. *See* garbage collection
 abstract
 class 200–210
 class modifier 200
 abstract methods
 declaring 203
 access
 and inheritance 180
 class modifiers 667
 method modifiers 81, 667
 variable modifiers 81, 667
 accessors and mutators. *See* getters and setters
 ActionListener interface 358, 358–361
 addActionListener() 359–361
 advice guy 480, 484
 Aeron™ 28
 animation 382–385
 API 154–155, 158–160
 ArrayList 532
 collections 558

appendix A 649–658
 beat box final client 650
 beat box final server 657
 appendix B
 access levels and modifiers 667
 assertions 662
 bit manipulation 660
 block scope 663
 immutability 661
 linked invocations 664
 multidimensional arrays 670
 String and StringBuffer methods 669
 apples and oranges 137
 arguments
 method 74, 76, 78
 polymorphic 187
 ArrayList 132, 133–138, 156, 208, 558
 API 532
 ArrayList<Object> 211–213
 autoboxing 288–289
 casting 229
 arrays
 about 17, 59, 135
 assigning 59
 compared to ArrayList 134–137
 creation 60
 declaring 59
 length attribute 17
 multidimensional 670
 objects, of 60, 83
 primitives, of 59
 assertions
 assertions 662
 assignments, primitive 52
 assignments, reference variables 55, 57, 83
 atomic code blocks 510–512. *See also* threads

the index

audio. *See* midi
autoboxing 288–291
 and operators 291
 assignments 291

B

bark different 73
bathtub 177
beat box 316, 347, 472. *See also* appendix A
beer 14
behavior 73
Bela Fleck 30
bitwise operators 660
bit shifting 660
block scope 663
boolean 51
boolean expressions 11, 114
 logical 151
BorderLayout manager 370–371, 401, 407
BoxLayout manager 411
brain barbell 33, 167, 188
break statement 105
BufferedReader 454, 478
BufferedWriter 453
buffers 453, 454
byte 51
bytecode 2

C

Calendar 303–305
 methods 305
casting
 explicit primitive 117
 explicit reference 216
 implicit primitive 117
catching exceptions 326

 catch 338
 catching multiple exceptions 329, 330, 332
 try 321
catch blocks 326, 338
 catching multiple exceptions 329, 330, 332
chair wars 28, 166
char 51
chat client 486
 with threads 518
chat server (simple) 520
checked exceptions
 runtime vs. 324
checking account. *See* Ryan and Monica
check box (JCheckBox) 416
class
 abstract 200–210
 concrete 200–210
 designing 34, 41, 79
 final 283
 fully qualified names 154–155, 157
client/server 473
code kitchen
 beat box save and restore 462
 final beat box. *See* appendix A
 making the GUI 418
 music with graphics 386
 playing sound 339
coffee cups 51
collections 137, 533
 API 558
 ArrayList 137
 ArrayList<Object> 211–213
 Collections.sort() 534, 539
 HashMap 533
 HashSet 533
 LinkedHashMap 533
 LinkedList 533
 List 557
 Map 557, 567
 parameterized types 137

- Set 557
- TreeSet 533
- Collections.sort() 534, 539
 - Comparator 551
 - compare() 553
- Comparable 547, 566
 - and TreeSet 566
 - compareTo() method 549
- Comparator 551, 566
 - and TreeSet 566
- compare() 553
- compareTo() 549
- comparing with == 86
- compiler 2
 - about 18
 - java -d 590
- concatenate 17
- concrete classes 200–210
- conditional expressions 10, 11, 13
- constants 282
- constructors
 - about 240
 - chaining 250–256
 - overloaded 256
 - superclass 250–256
- contracts 190–191, 218
- cups 51
- curly braces 10

D

- daily advice client 480
- daily advice server 484
- dancing girl 316
- dates
 - Calendar 303
 - methods 305
 - formatting 301
 - GregorianCalendar 303
 - java.util.Date 303
- deadlock 516
- deadly diamond of death 223
- declarations
 - about 50
 - exceptions 335–336
 - instance variables 50
- default access 668
- default value 84
- deployment options 582, 608
- deserialized objects 441. *See also* serialization
- directory structures
 - packages 589
 - servlets 626
- doctor 169
- dot operator
 - reference 54
- double 51
- duck 277
 - construct 242
 - garbage collect 261
- ducking exceptions 335

E

- EJB 631
- encapsulation
 - about 79–82
 - benefits 80
- end of book 648
- enumerations 671–672
- enums 671–672
- equality 560
 - and hashCode() 561
- equals() 561
- equals()
 - about 209
 - Object class 209

the index

- event handling 357–361
 - event object 361
 - listener interface 358–361
 - using inner classes 379
- event source 359–361
- exceptions
 - about 320, 325, 338
 - catch 321, 338
 - catching multiple exceptions 329, 332
 - checked vs. runtime 324
 - declaring 335–336
 - ducking 335–336
 - finally 327
 - flow control 326
 - handle or declare law 337
 - propagating 335–336
 - remote exceptions 616
 - throwing 323–326
 - try 321, 338
- executable JAR 585–586, 586
 - with packages 592, 592–593
- exercises
 - be the... 88, 118, 266, 310, 395
 - code magnets 20, 43, 64, 119, 312, 349, 467, 524–525
 - honeypot 267
 - true or false 311, 348, 466, 602
 - what’s the declaration 231
 - what’s the picture 230
 - which layout manager? 424
 - who am I 45, 89, 394
- Extreme Programming 101

F

- File 452
- FileInputStream 441. *See also* I/O
- FileOutputStream 432
- FileReader 454. *See also* I/O
- files
 - File class 452

- reading from 441, 454
 - source file structure 7
 - writing to 432, 447

- FileWriter 447

- File class 452

- final

- class 189, 283
 - methods 189, 283
 - static variables 282
 - variables 282, 283

- finally block 327

- fireside chats

- about 18

- five minute mystery. *See* puzzles

- float 51

- FlowLayout 403, 408–410

- flow control

- exceptions 326

- font 406

- formatting

- dates 301–302
 - format specifiers 295–296
 - argument 300
 - numbers 294–295
 - printf() 294
 - String.format() 294

- for loops 105

- fully qualified name 154, 157

- packages 587

G

- garbage collection

- about 40
 - eligible objects 260–263
 - heap 57, 58
 - nulling references 58
 - reassigning references 58

- generics 540, 542, 568–574

- methods 544

- wildcards 574
- getters and setters 79
- ghost town 109
- giraffe 50
- girl dreaming
 - inner classes 375
 - Java Web Start 596
- girl in a tub 177
- girl who isn't getting it 182–188
- graphics 364–366. *See also* GUI
 - Graphics2D class 366
 - Graphics object 364
- GregorianCalendar 303
- guessing game 38
- GUI 406
 - about 354, 400
 - animation 382–385
 - BorderLayout 370–371, 401, 407
 - box layout 403, 411
 - buttons 405
 - components 354, 363–368, 400
 - event handling 357–361, 379
 - flow layout 403, 408
 - frames 400
 - graphics 363–367
 - ImageIcon class 365
 - JButton 400
 - JLabel 400
 - JPanel 400, 401
 - JTextArea 414
 - JTextField 413
 - layout managers 401–412
 - listener interface 358–361
 - scrolling (JScrollPane) 414
 - Swing 354
- GUI Constants
 - ScrollPaneConstants.HORIZONTAL_SCROLL-
BAR_NEVER 415
 - ScrollPaneConstants.VERTICAL_SCROLLBAR_
ALWAYS 415
- GUI methods

- drawImage() 365
- fillOval() 365
- fillRect() 364
- gradientPaint(). *See also* GUI
- paintComponent() 364
- setColor() 364
- setFont() 406
- GUI Widgets 354
 - JButton 354, 405
 - JCheckBox 416
 - JFrame 354, 400, 401
 - JList 417
 - JPanel 400, 401
 - JScrollPane 414, 417
 - JTextArea 414
 - JTextField 413

H

- HAS-A 177–181
- hashCode() 561
- HashMap 533, 558
- HashSet 533, 558
- Hashtable 558
- heap
 - about 40, 57, 236–238
 - garbage collection 40, 57, 58

I

- I/O
 - BufferedReader 454, 478
 - BufferedWriter 453
 - buffers 453
 - deserialization 441
 - FileInputStream 441
 - FileOutputStream 432
 - FileWriter 447
 - InputStreamReader 478
 - ObjectInputStream 441
 - ObjectOutputStream 432, 437
 - serialization 432, 434–439, 437, 446, 460

the index

- streams 433, 437
 - with sockets 478
- if-else 13
- if statement 13
- immutability, Strings
 - immutability 661
- implements 224
- imports
 - static imports 307
- import statement 155, 157
- increment 105
- inheritance
 - about 31, 166–192
 - and abstract classes 201
 - animals 170–175
 - IS-A 214, 251
 - super 228
- initializing
 - instance variables 84
 - primitives 84
 - static variables 281
- inner classes
 - about 376–386
 - events 379
- inner class threesome 381
- InputStreamReader 478
- instance variables
 - about 34, 73
 - declaring 84
 - default values 84
 - initializing 84
 - life and scope 258–263
 - local variables vs. 236–238, 239
 - static vs. 277
- instantiation. *See* objects
- int 50
 - primitive 51
- Integer. *See* wrapper
- interfaces

- about 219–227
- for serialization 437
- implementing 224, 437
- implementing multiple 226
- java.io.Serializable 437
- IP address. *See* networking
- IS-A 177–181, 251

J

- J2EE 631
- JAR files
 - basic commands 593
 - executable 585–586, 592
 - manifest 585
 - running executable 586, 592
 - tool 593
 - with Java Web Start 598
- Java, about 5, 6
- javac. *See* compiler
- Java in a Nutshell 158–159
- java sound 317, 340
- Java Web Start 597–601
 - jnlp file 598, 599
- Jini 632–635
- JNLP 598
 - jnlp file 599
- JPEG 365
- JVM
 - about 2, 18
- JWS. *See* Java Web Start

K

- keywords 53

L

- l 264
- layout managers 401–412
 - BorderLayout 370–371, 403, 407

- BoxLayout 403, 411
- FlowLayout 403, 408–410
- lingerie, exceptions 329
- LinkedHashMap 533, 558
- LinkedHashSet 558
- LinkedList 533, 558
- linked invocations 664
- List 557
- listeners
 - listener interface 358–361
- literals, assigning values
 - primitive 52
- local
 - variables 85, 236, 236–238, 258–263
- locks
 - object 509
 - threads 509
- long 51
- loops
 - about 10
 - break 105
 - for 105
 - while 115
- lost update problem. *See* threads

M

- main() 9, 38
- make it stick 53, 87, 157, 179, 227, 278
- manifest file 585
- Map 557, 567
- Math class
 - methods 274–278, 286
 - random() 111
- memory
 - garbage collection 260–263
- metacognitive tip 33, 108, 325
- methods
 - about 34, 78

- abstract 203
- arguments 74, 76, 78
- final 283
- generic arguments 544
- on the stack 237
- overloading 191
- overriding 32, 167–192
- return 75, 78
- static 274–278
- midi 317, 340–346, 387–390
- midi sequencer 340–346
- MINI Cooper 504
- modifiers
 - class 200
 - method 203
- multidimensional arrays 670
- multiple inheritance 223
- multiple threads. *See* threads
- music. *See* midi
- mystery. *See* puzzles

N

- naming 53. *See also* RMI
 - classes and interfaces 154–155, 157
 - collisions 587
 - packages 587
- networking
 - about 473
 - ports 475
 - sockets 475
- new 55
- null
 - reference 262
- numbers
 - formatting 294–295

O

- ObjectOutputStream 432, 437

the index

objects

- about 55
- arrays 59, 60, 83
- comparing 209
- creation 55, 240–256
- eligible for garbage collection 260–263
- equality 560
- equals() 209, 561
- life 258–263
- locks 509

Object class

- about 208–216
- equals() 561
- hashCode() 561
- overriding methods 563

object graph 436, 438

object references 54, 56

- assignment 55, 262
- casting 216
- comparing 86
- equality 560
- nulling 262
- polymorphism 185–186

OO

- contracts 190–191, 218
- deadly diamond of death 223
- design 34, 41, 79, 166–191
- HAS-A 177–181
- inheritance 166–192
- interfaces 219–227
- IS-A 177–181, 251
- overload 191
- override 167–192
- polymorphism 183, 183–191, 206–217
- superclass 251–256

operators

- and autoboxing 291
- bitwise 660
- comparison 151
- conditional 11
- decrement 115

increment 105, 115

logical 151

shift 660

overload 191

constructors 256

override

about 32, 167–192

polymorphism. *See* polymorphism

P

packages 154–155, 157, 587–593

directory structure 589

organizing code 589

paintComponent() 364–368

parameter. *See* arguments

parameterized types 137

parsing an int. *See* wrapper

parsing text with String.split() 458

pass-by-copy. *See* pass-by-value

pass-by-value 77

phrase-o-matic 16

polymorphism 183–191

abstract classes 206–217

and exceptions 330

arguments and return types 187

references of type Object 211–213

pool puzzle. *See* puzzles

ports 475

prep code 99–102

primitives 53

== operator 86

autoboxing 288–289

boolean 51

byte 51

char 51

double 51

float 51

int 51

- ranges 51
 - short 51
 - type 51
- primitive casting
 - explicit primitive 117
- printf() 294
- PrintWriter 479
- private
 - access modifier 81
- protected 668
- public
 - access modifier 81, 668
- puzzles
 - five minute mystery 92, 527, 674
 - Java cross 22, 120, 162, 350, 426, 603
 - pool puzzle 24, 44, 65, 91, 194, 232, 396

Q

- quiz card builder 448, 448–451

R

- rabbit 50
- random() 111
- ready-bake code 112, 152–153, 520
- reference variables. *See* object references
 - casting 216
- registry, RMI 615, 617, 620
- remote control 54, 57
- remote interface. *See* RMI
- reserved words 53
- return types
 - about 75
 - polymorphic 187
 - values 78
- risky code 319–336
- RMI
 - about 614–622
 - client 620, 622

- compiler 618
 - Jini. *See also* Jini
 - Naming.lookup() 620
 - Naming.rebind(). *See also* RMI
 - registry 615, 617, 620
 - remote exceptions 616
 - remote implementation 615, 617
 - remote interface 615, 616
 - rmic 618
 - skeleton 618
 - stub 618
 - UnicastRemoteObject 617
 - universal service browser 636–648
- rmic. *See* RMI
- run()
 - overriding in Runnable interface 494
- Runnable interface 492
 - about 493
 - run() 493, 494
 - threads 493
- runnable thread state 495
- Ryan and Monica 505–506
 - introduction 505–506

S

- scary objects 200
- scheduling threads
 - scheduling 496–498
- scope
 - variables 236–238, 258–263
- scrolling (JScrollPane) 414
- serialization 434–439, 446
 - deserialization 460
 - interface 437
 - ObjectInputStream. *See* I/O
 - objectOutputStream 432
 - objects 460
 - object graph 436
 - reading. *See* I/O
 - restoring 460. *See also* I/O

the index

- saving 432
- serialVersionUID 461
- transient 439
- versioning 460, 461
- writing 432
- server
 - socket 483. *See also* socket
- servlet 625–627
- Set 557
 - importance of equals() 561
 - importance of hashCode() 561
- short 51
- short circuit logical operators 151
- sink a dot com 96–112, 139–150
- skeleton. *See* RMI
- sleep() 501–503
- sleeping threads 501–503
- snowboard 214
- socket
 - about 475
 - addresses 475
 - creating 478
 - I/O 478
 - ports 475
 - reading from 478
 - server 483
 - TCP/IP 475
 - writing to 479
- sorting
 - Collections.sort() 534, 539, 547
 - Comparable interface 547, 549
 - Comparator 551, 553
 - TreeSet 564–566
- source files
 - structure of 7
- specifiers
 - format specifiers 295, 298
 - argument specifier 300
- stack
 - heap vs. 236
 - methods on 237
 - scope 236
 - threads 490
 - trace 323
- static
 - enumerated types 671
 - initializer 282
 - Math class methods 274–278
 - methods 274–278
 - static imports 307
 - variables 282
- streams 433. *See also* I/O
- String
 - arrays 17
 - concatenating 17
 - methods 669
 - parsing 458
 - String.format() 294–297
 - String.split() 458
- StringBuffer/StringBuilder
 - methods 669
- stub. *See* RMI
- subclass
 - about 31, 166–192
- super 228
 - about 31
- superclass
 - about 166–192, 214–217, 228
- super constructor 250–256
- Swing. *See* GUI
- synchronized
 - methods 510. *See also* threads
- syntax
 - about 10, 12
- System.out.print() 13
- System.out.println() 13

T

talking head 203
 TCP ports 475
 Telluride 30
 testing
 extreme programming 101
 text
 parsing with `String.split()` 458 458
 read from a file. *See also* I/O
 write to a file 447
 text area (`JTextArea`) 414
 text field (`TextField`) 413
`Thread.sleep()` 501–503
 threads
 about 489–515
 deadlock 516
 locks 509
 lost update problem 512–514
 `run()` 493, 494
 `Runnable` 492, 493, 494
 Ryan and Monica problem 505–507
 scheduling 496, 496–498
 `sleep()` 501–503
 stack 490–491
 `start()` 492
 starting 492
 states 495, 496
 summary 500, 517
 synchronized 510–512
 unpredictability 498–499
 throw
 exceptions 323–326
 throws 323–326
 transient 439
`TreeMap` 558
`TreeSet` 533, 558, 564–566, 566
 try
 blocks 321, 326

type 50
 parameter 137, 542, 544
 type-safety 540
 and generics 540

U

universal service browser 636–648

V

variables
 assigning 52, 262
 declaring 50, 54, 84, 236–238
 local 85, 236–238
 nulling 262
 primitive 51, 52
 references 54, 55, 56, 185–186
 scope 236–238
 static. *See* static
 variable declarations 50
 instance 84
 primitive 51
 reference 54
 virtual method invocation 175

W

web start. *See* Java Web Start
 while loops 11, 115
 wildcard 574
 wine 202
 wrapper 287
 autoboxing 288–289
 conversion utilities 292
 `Integer.parseInt()` 104, 106, 117
 writing. *See* I/O

Don't you know about the web site?
We've got answers to some of the
Sharpens, examples, the Code Kitchens,
Ready-bake Code, and daily updates
from the Head First author blogs!

This isn't goodbye

**Bring your brain over to
wickedlysmart.com**



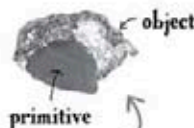
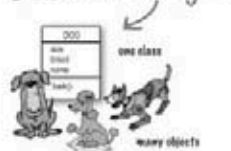
Head First Java™

Java

What will you learn from this book?

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

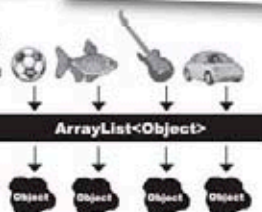
Learn the difference between a class and an object.



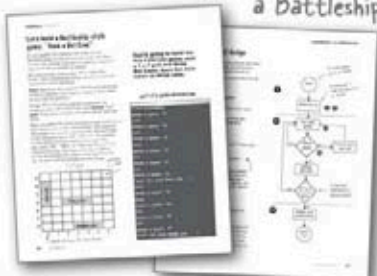
Discover how to treat a primitive like an object.

The objects go IN as SoccerBall, Fish, Guitar, and Car.

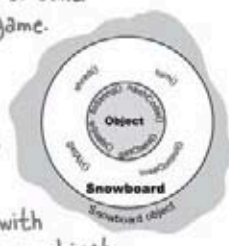
But they come OUT as though they were of type **Object**.



Use your Java skills to build a Battleship-style game.



Get in touch with your inner object.



Understand why using polymorphic references of type **Object** has a price...

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**.

What's so special about this book?

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

US \$44.95

CAN \$62.95

ISBN: 978-0-596-00920-5



O'REILLY®

www.oreilly.com

www.headfirstlabs.com

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

—*slashdot.org*

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

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

"*Head First Java* transforms the printed page into the closest thing to a GUI you've ever seen. In a wry, hip manner, the authors make learning Java an engaging, 'what're they gonna do next?' experience."

—Warren Keuffel, Software Development Magazine

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

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