O'REILLY®

Third Edition
Covers Java 17

Head First Java

A Learner's Guide to Real-World Programming

Kathy Sierra
& Bert Bates

Early Release
RAW & UNEDITED

A Brain-Friendly Guide

Head First Java

A Brain-Friendly Guide

Kathy Sierra and Bert Bates

Head First Java

by Kathy Sierra and Bert Bates

Printed in the United States of America.

Published by O’Reilly Media, Inc. , 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O’Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (http://oreilly.com). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com .

Editors: Nicole Tache and Suzanne McQuade

Production Editor: Kristen Brown

Copyeditor: FILL IN COPYEDITOR

Proofreader: FILL IN PROOFREADER

Indexer: FILL IN INDEXER

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Kate Dullea

December 2021: Third Edition

Revision History for the Third Edition

- 2021-03-17: First Early Release

See http://oreilly.com/catalog/errata.csp?isbn=9781491910771 for release details.

978-1-491-91077-1

[FILL IN]

Chapter 1. Dive in A Quick Dip: Breaking the Surface

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

The Way Java Works

The goal is to write one application (in this example, an interactive party invitation) and have it work on whatever device your friends have.

What you'll do in Java

You'll type a source code file, compile it using the javac compiler, then run the compiled bytecode on a Java virtual machine.

> **Note**
>
> this is **NOT** meant to be a tutorial... you'll be writing real code in a moment, but for now, we just want you to get a feel for how it all fits together.
>
> In other words, the code on this page isn't quite real, don't try to compile it..)

A Very Brief History of Java

Java was initially released (some would say "escaped"), on January 23, 1996. It's over 25 years old! In the first 25 years, Java as a language evolved, and the Java API grew enormously. The best estimate we have is that over 17 gazillion lines of Java code have been written in the last 25 years. As you spend time being a Java programmer, you will most certainly come across Java code that's quite old, and some that's much newer.

In this book we'll generally start off by using older coding styles (remember, you're likely to encounter such code in the "real world"), and then we'll introduce newer style code.

In a similar fashion, we will sometimes show you older classes in the Java API, and then show you newer alternatives.

Speed and Memory Usage

When Java was first released, it was slow. But soon after, the HotSpot VM was created, as were other performance enhancers, and while it's true that Java isn't the fastest language out there, it's considered to be a very fast language. Almost as fast as languages like C and Rust, and MUCH, MUCH faster than languages like Python or C# or even the popular new JVM language Kotlin.

But - full disclosure - compared to C and Rust, Java uses a lot of memory.

INLINE SHARPEN YOUR PENCIL

Look how easy it is to write Java.

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

```
int size = 27;
String name = “Fido”;
Dog myDog = new Dog(name, size);
x = size - 5;
if (x < 15) myDog.bark(8);

while (x > 3) {
   myDog.play();
}

int[] numList = {2,4,6,8};
System.out.print(“Hello”);
System.out.print(“Dog: “ + name);
String num = “8”;
int z = Integer.parseInt(num);

try {
  readTheFile(“myFile.txt”);
}

catch(FileNotFoundException ex) {
   System.out.print(“File not found.”);
}
```

Q: The naming conventions for Java’s various versions are confusing. There was JDK 1.0, and j2SE 1.2, 1.3, 1.4 then a jump to J2SE 5.0, then it changed to Java SE6, Java SE7, and last time I checked, Java was up to Java SE 15. Can you explain what’s going on?

A: It’s true that for the first several years, Java’s version numbers were hard to predict. But since the release of Java SE 9 in 2017, Java has been on a

“rapid release” model, and it looks like from 2017 through 2020, the version numbers have become more stable and predictable.

Inline Sharpen your pencil Answers

LOOK HOW EASY IT IS TO WRITE JAVA.

```
int size = 27;
String name = “Fido”;
Dog myDog = new Dog(name, size);
x = size - 5;
if (x < 15) myDog.bark(8);

while (x > 3){
   myDog.play();
}

int[] numList = {2,4,6,8};
System.out.print(“Hello”);
System.out.print(“Dog: “ + name);
String num = “8”;
int z = Integer.parseInt(num);

try {
  readTheFile(“myFile.txt”);
}

catch(FileNotFoundException ex) {
   System.out.print(“File not found.”);
}
```

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

Code structure in Java

Put a class in a source file.

Put methods in a class.

Put statements in a method.

What goes in a source file?

A source code file (with the *.java* extension) holds one ***class*** definition. The class represents a *piece* of your program, although a very tiny application might need just a single class. The class must go within a pair of curly braces.

What goes in a class?

A class has one or more ***methods***. In the Dog class, the ***bark*** method will hold instructions for how the Dog should bark. Your methods must be declared *inside* a class (in other words, within the curly braces of the class).

What goes in a method?

Within the curly braces of a method, write your instructions for how that method should be performed. Method *code* is basically a set of statements, and for now you can think of a method kind of like a function or procedure.

Anatomy of a class

When the JVM starts running, it looks for the class you give it at the command line. Then it starts looking for a specially-written method that looks exactly like:

```
public static void main (String[] args) {
   // your code goes here
}
```

Next, the JVM runs everything between the curly braces { } of your main method. Every Java application has to have at least one **class**, and at least one **main** method (not one main per *class*; just one main per *application*).

> **NOTE**
>
> Don’t worry about memorizing anything right now... this chapter is just to get you started.

Writing a class with a main

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

Running a program means telling the Java Virtual Machine (JVM) to “Load the **`MyFirstApp`** class, then start executing its **`main()`** method. Keep running ‘til all the code in main is finished.”

In Chapter 2, we go deeper into the whole *class* thing, but for now, all you need to think is, ***how do I write Java code so that it will run?*** And it all begins with **`main()`**.

The **`main()`** method is where your program starts running.

No matter how big your program is (in other words, no matter how many *classes* your program uses), there's got to be a **main()** method to get the ball rolling.

What can you say in the main method?

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

Your code can tell the JVM to:

 **Inline do something**

Statements: declarations, assignments, method calls, etc.

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

 Inline **do something again and again**

Loops: *for* and *while*

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

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

 Inline **do something under this condition**

Branching: *if/else* tests

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

SYNTAX FUN

Inline Each statement must end in a semicolon.

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

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

```
x = 22;
// this line disturbs me
```

Inline Most white space doesn't matter.

```
x = 3 ;
```

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

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

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

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

Looping and looping and...

Java has a lot of looping constructs: *while*, *do-while*, and *for,* being the oldest. You'll get the full loop scoop later in the book, but not for awhile, so let's do *while* for now.

The syntax (not to mention logic) is so simple you're probably asleep already. As long as some condition is true, you do everything inside the loop *block*. The loop block is bounded by a pair of curly braces, so whatever you want to repeat needs to be inside that block.

The key to a loop is the *conditional test*. In Java, a conditional test is an expression that results in a *boolean* value—in other words, something that is either ***true*** or ***false***.

If you say something like, "While *iceCreamInTheTub is true*, keep scooping", you have a clear boolean test. There either *is* ice cream in the tub or there *isn't*. But if you were to say, "While *Bob* keep scooping", you don't have a real test. To make that work, you'd have to change it to something like, "While Bob is snoring..." or "While Bob is *not* wearing plaid..."

Simple boolean tests

You can do a simple boolean test by checking the value of a variable, using a comparison operator including:

< (less than)

> (greater than)

== (equality) (yes, that's *two* equals signs)

Notice the difference between the *assignment* operator (a *single* equals sign) and the *equals* operator (*two* equals signs). Lots of programmers accidentally type = when they *want* ==. (But not you.)

```
int x = 4; // assign 4 to x
while (x > 3) {
  // loop code will run because
  // x is greater than 3
  x = x - 1; // or we'd loop forever
}
int z = 27; //
  while (z == 17) { // loop code will not run because
  // z is not equal to 17
}
```

There are no dumb Questions

Q: Why does everything have to be in a class?

A: Java is an object-oriented (OO) language. It's not like the old days when you had steam-driven compilers and wrote one monolithic source file with a pile of procedures. In Chapter 2 you'll learn that a class is a blueprint for an object, and that nearly everything in Java is an object.

Q: Do I have to put a main in every class I write?

A: Nope. A Java program might use dozens of classes (even hundreds), but you might only have *one* with a main method — the one that starts the program running. You might write test classes, though, that have main methods for testing your *other* classes.

Q: In my other language I can do a boolean test on an integer. In Java, can I say something like:

```
int x = 1;
while (x){ }
```

A: No. A *boolean* and an *integer* are not compatible types in Java. Since the result of a conditional test *must* be a boolean, the only variable you can directly test (without using a comparison operator) is a ***boolean.*** For example, you can say:

```
boolean isHot = true;
while(isHot) { }
```

Example of a while loop

```
public class Loopy {
   public static void main (String[] args) {
     int x = 1;
     System.out.println("Before the Loop");
     while (x < 4) {
       System.out.println("In the loop");
```

```
        System.out.println("Value of x is " + x);
          x = x + 1;
    }
    System.out.println("This is after the loop");
  }
}
```

BULLET POINTS 

- Statements end in a semicolon **;**
- Code blocks are defined by a pair of curly braces **{ }**
- Declare an *int* variable with a name and a type: **int x;**
- The **assignment** operator is *one* equals sign =
- The **equals** operator uses *two* equals signs ==
- A *while* loop runs everything within its block (defined by curly braces) as long as the *conditional test* is ***true***.
- If the conditional test is ***false***, the *while* loop code block won't run, and execution will move down to the code immediately *after* the loop block.
- Put a boolean test inside parentheses: `while` **(x == 4)** { }

Conditional branching

In Java, an *if* test is basically the same as the boolean test in a *while* loop – except instead of saying, "***while*** there's still beer...", you'll say, "***if*** there's still beer..."

```
class IfTest {
```

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

The code above executes the line that prints “x must be 3” only if the condition (*x* is equal to 3) is true. Regardless of whether it’s true, though, the line that prints, “This runs no matter what” will run. So depending on the value of *x*, either one statement or two will print out.

But we can add an *else* to the condition, so that we can say something like, “*If* there’s still beer, keep coding, *else* (otherwise) get more beer, and then continue on...”

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

SYSTEM.OUT.PRINT VS.

System.out.println

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

Did you spot the difference?

System.out.***println*** inserts a newline (think of print***ln*** as **print*new*line** while System.out.***print*** keeps printing to the *same* line. If you want each thing you print out to be on its own line, use print**ln**. If you want everything to stick together on one line, use print.

 INLINE SHARPEN YOUR PENCIL

Given the output:

```
% java DooBee
DooBeeDooBeeDo
```

Fill in the missing code:

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

Coding a Serious Business Application

Let’s put all your new Java skills to good use with something practical. We need a class with a *main()*, an *int* and a *String* variable, a *while* loop, and an *if* test. A little more polish, and you’ll be building that business back-end in no time. But *before* you look at the code on this page, think for a moment about how *you* would code that classic children’s favorite, “99 bottles of beer.”


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

     while (beerNum > 0) {

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

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

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

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

Monday Morning at Bob’s Java-Enabled House

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

First, the alarm clock sends a message to the coffee maker “Hey, the geek’s sleeping in again, delay the coffee 12 minutes.”

The coffee maker sends a message to the Motorola™ toaster, “Hold the toast, Bob’s snoozing.”

The alarm clock then sends a message to Bob’s Android, “Call Bob’s 9 o’clock and tell him we’re running a little late.”

Finally, the alarm clock sends a message to Sam’s (Sam is the dog) wireless collar, with the too-familiar signal that means, “Get the paper, but don’t expect a walk.”

A few minutes later, the alarm goes off again. And *again* Bob hits SNOOZE and the appliances start chattering. Finally, the alarm rings a third time. But just as Bob reaches for the snooze button, the clock sends the “jump and bark” signal to Sam’s collar. Shocked to full consciousness, Bob rises, grateful that his Java skills and a little trip to Radio Shack™ have enhanced the daily routines of his life.

His toast is toasted.

His coffee steams.

His paper awaits.

Just another wonderful morning in ***The Java-Enabled House.***

> Could this story be true? Mostly, yes!. There *are* versions of Java running in devices including cell phones (*especially* cell phones), ATMs, credit cards, home security systems, parking meters, game consoles and more –but you might not find a Java toaster or dog collar... yet.
>
> Java Platform, Micro Edition (Java ME), allows Java applications to run on embedded and mobile devices. Java ME is very popular in the Internet of Things (IoT), world, providing security, network protocols, and all the other IoT goodies.

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

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

 **// make three sets of words to choose from. Add your own!**

```
     String[] wordListOne = {“agnostic”, “opinionated”,
“voice activated”, “haptically driven”, “extensible”,
“reactive”, “agent based”, “functional”, “AI enabled”,
“strongly typed”};

     String[] wordListTwo = {“loosely coupled”,
“six sigma”, “asynchronous”, “event driven”, “pub
sub”, “IoT”, “cloud native”, “service oriented”,
```

```
“containerized”, “serverless”, “n-tier”, “distributed
ledger”};

     String[] wordListThree = {“framework”, “library”,
“DSL”, “REST API”, “repository”, “pipeline”, “service
mesh”, “architecture”, “perspective”, “design”,
“orientation”};
```

```
// find out how many words are in each list

int oneLength = wordListOne.length;
int twoLength = wordListTwo.length;
int threeLength = wordListThree.length;
```

```
// generate three random numbers

int rand1 = (int) (Math.random() * oneLength);
int rand2 = (int) (Math.random() * twoLength);
int rand3 = (int) (Math.random() * threeLength);
```

```
// now build a phrase


    String phrase = wordListOne[rand1] + “ “ +
wordListTwo[rand2] + “ “ + wordListThree[rand3];
```

```
// print out the phrase


  System.out.println(“What we need is a “ + phrase);
  }
}
```

NOTE

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

Phrase-O-Matic

How it works.

In a nutshell, the program makes three lists of words, then randomly picks one word from each of the three lists, and prints out the result. Don't worry if you don't understand *exactly* what's happening in each line. For gosh sakes, you've got the whole book ahead of you, so relax. This is just a quick look from a 30,000 foot outside-the-box targeted leveraged paradigm.

1. The first step is to create three String arrays – the containers that will hold all the words. Declaring and creating an array is easy; here's a small one:

   String[] pets = {"Fido", "Zeus", "Bin"};

   Each word is in quotes (as all good Strings must be) and separated by commas.

2. For each of the three lists (arrays), the goal is to pick a random word, so we have to know how many words are in each list. If there are 14 words in a list, then we need a random number between 0 and 13 (Java arrays are zero-based, so the first word is at position 0, the second word position 1, and the last word is position 13 in a 14-element array). Quite handily, a Java array is more than happy to tell you its length. You just have to ask. In the pets array, we'd say:

   ```
   int x = pets.length;
   ```

   and **x** would now hold the value 3.

> **NOTE**
>
> **what we need here is a...**
>
> **pervasive targeted process**

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

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

> **NOTE**
>
> **dynamic outside-the-box tipping-point**
>
> **smart distributed core competency**

4. Now we get to build the phrase, by picking a word from each of the three lists, and smooshing them together (also inserting spaces between words). We use the "+" operator, which *concatenates* (we prefer the more technical '*smooshes*') the String objects together. To get an element from an array, you give the array the index number (position) of the thing you want using:

```
String s = pets[0]; // s is now the String "Fido"
s = s + " " + "is a dog"; // s is now "Fido is a dog"
```

> **NOTE**
>
> **24/7 empowered mindshare**
>
> **30,000 foot win-win vision**

5. Finally, we print the phrase to the command-line and... voila! *We're in marketing*.

> **NOTE**
>
> **six-sigma networked portal**

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

The Java Virtual Machine	The Compiler
What, are you kidding? ***HELLO***. I am Java. I'm the guy who actually makes a program run. The compiler just gives you a file. That's it. Just a file. You can print it out and use it for wall paper, kindling, lining the bird cage whatever, but the file doesn't do anything unless I'm there to run it.	
And that's another thing, the compiler has no sense of humor. Then again, if you had to spend all day checking nit-picky little syntax violations...	I don't appreciate that tone.
I'm not saying you're, like, *completely* useless. But really, what is it that you do? Seriously. I have no idea. A programmer could just write bytecode by hand, and I'd take it. You might be out of a job soon, buddy.	Excuse me, but without *me*, what exactly would you run? There's a *reason* Java was designed to use a bytecode compiler, for your information. If Java were a purely interpreted language, where—at runtime—the virtual machine had to translate straight-from-a-text-editor source code, a Java program would run at a ludicrously glacial pace. Java's had a challenging enough time convincing people that it's finally fast and powerful enough for most jobs.
(I rest my case on the humor thing.) But you still didn't answer my question, what *do* you actually do?	Excuse me, but that's quite an ignorant (not to mention *arrogant*) perspective. While it *is* true that —*theoretically*—you can run any properly formatted bytecode even if it didn't come out of a Java compiler, in practice that's absurd. A programmer writing bytecode by hand is like doing your word processing by writing raw postscript. And I would appreciate it if you would *not* refer to me as "buddy." Remember that Java is a strongly-typed language, and that means I can't allow variables to hold data of the wrong type. This is a crucial safety feature, and I'm able to stop the vast majority of violations before they ever get to you. And I also—
But some still get through! I can throw ClassCast-Exceptions and sometimes I get people trying to put the wrong type of thing in an	Excuse me, but I wasn't done. And yes, there *are* some datatype exceptions that can emerge at runtime, but some of those have to be allowed to support one of Java's other important features— dynamic binding. At runtime, a Java program can include new objects that weren't even *known* to the original programmer, so I have to allow a certain amount of flexibility. But my job is to stop anything that

array that was declared to hold something else, and—	would never—*could* never—succeed at runtime. Usually I can tell when something won't work, for example, if a programmer accidentally tried to use a Button object as a Socket connection, I would detect that and thus protect him from causing harm at runtime.
OK. Sure. But what about *security*? Look at all the security stuff I do, and you're like, what, checking for *semicolons*? Oooohhh big security risk! Thank goodness for you!	Excuse me, but I am the first line of defense, as they say. The datatype violations I previously described could wreak havoc in a program if they were allowed to manifest. I am also the one who prevents access violations, such as code trying to invoke a private method, or change a method that – for security reasons – must never be changed. I stop people from touching code they're not meant to see, including code trying to access another class' critical data. It would take hours, perhaps days even, to describe the significance of my work.
Whatever. I have to do that same stuff *too*, though, just to make sure nobody snuck in after you and changed the bytecode before running it.	Of course, but as I indicated previously, if I didn't prevent what amounts to perhaps 99% of the potential problems, you would grind to a halt. And it looks like we're out of time, so we'll have to revisit this in a later chat.
Oh, you can count on it. *Buddy*.	

Code Magnets

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

BE the compiler

Each of the Java files on this page represents a complete source file. Your job is to play compiler and determine whether each of these files will compile. If they won't compile, how would you fix them?

A

```
class Exercise1b {
  public static void main(String [] args) {
    int x = 1;
    while ( x < 10 ) {
      if ( x > 3) {
        System.out.println("big x");
    }
  }
 }
}
```

B

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

C

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

```
        }
    }
```

JavaCross 7.0

Let’s give your right brain something to do.

It’s your standard crossword, but almost all of the solution words are from Chapter 1. Just to keep you awake, we also threw in a few (non-Java) words from the high-tech world.

Across

4. Command-line invoker

6. Back again?

8. Can’t go both ways

9. Acronym for your laptop’s power

12. number variable type

13. Acronym for a chip

14. Say something

18. Quite a crew of characters

19. Announce a new class or method

21. What’s a prompt good for?

Down

1. Not an integer (or _____ your boat)

2. Come back empty-handed

3. Open house

5. ‘Things’ holders

7. Until attitudes improve

10. Source code consumer

11. Can’t pin it down

13. Dept. of LAN jockeys

15. Shocking modifier

16. Just gotta have one

17. How to get things done

20. Bytecode consumer

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

Pool Puzzle

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

Output

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

    while ( ___________ ) {
      _____________________________
      if ( x < 1 ) {
       _____________________________
      }
     _____________________________
     if ( ___________ ) {
        ___________________________________
        ___________
     }
     if ( x == 1 ) {
       _____________________________
     }
     if ( ____________ ) {
       _____________________________
     }
     System.out.println(“”);
     _____________
    }
  }
}
```

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

 Inline Exercise Solutions

Code Magnets:

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

    int x = 3;
    while (x > 0) {

      if (x > 2) {
        System.out.print(“a”);
      }

      x = x - 1;
      System.out.print(“-”);

      if (x == 2) {
        System.out.print(“b c”);
      }

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

```
class Exercise1b {
  public static void main(String [] args) {
    int x = 1;
    while ( x < 10 ) {
      x = x + 1;
A     if ( x > 3) {
        System.out.println(“big x”);
       }
    }
  }       This will compile and run (no output), but
}         without a line added to the program, it
          would run forever in an infinite ‘while’ loop!
```

```
class Foo {

public static void main(String [] args) {
  int x = 5;
  while ( x > 1 ) {
    x = x - 1;
B   if ( x < 3) {
      System.out.println(“small x”);
    }
  }   This file won’t compile without a
 }    class declaration, and don’t forget
}     the matching curly brace !
```

```
class Exercise1b {
  public static void main(String [] args) {
    int x = 5;
    while ( x > 1 ) {
C     x = x - 1;
      if ( x < 3) {
        System.out.println(“small x”);
    }
  }     The ‘while’ loop code must be inside
 }      a method. It can’t just be
}       hanging out inside the class.
```

Inline puzzle answers

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

    while ( X < 4 ) {

      System.out.print(“a”);
      if ( x < 1 ) {
         System.out.print(“ “);
      }
      System.out.print(“n”);

      if ( X > 1 ) {
```

```
            System.out.print(“ oyster”);
            x = x + 2;
        }
        if ( x == 1 ) {

            System.out.print(“noys”);
        }
        if ( X < 1 ) {

            System.out.print(“oise”);
        }
        System.out.println(“”);

        X = X + 1;
        }
    }
}
```

Free! Bonus Puzzle!

There is another way to solve the pool puzzle, that might be easier to read, can you find it?

Chapter 2. Classes and Objects: A Trip to Objectville

I was told there would be objects. In Chapter 1, we put all of our code in the main() method. That's not exactly object-oriented. In fact, that's not object-oriented *at all*. Well, we did *use* a few objects, like the String arrays for the Phrase-O-Matic, but we didn't actually develop any of our own object *types*. So now we've got to leave that procedural world behind, get

the heck out of main(), and start making some objects of our own. We'll look at what makes object-oriented (OO) development in Java so much fun. We'll look at the difference between a *class* and an *object*. We'll look at how objects can give you a better life (at least the programming part of your life. Not much we can do about your fashion sense). Warning: once you get to Objectville, you might never go back. Send us a postcard.

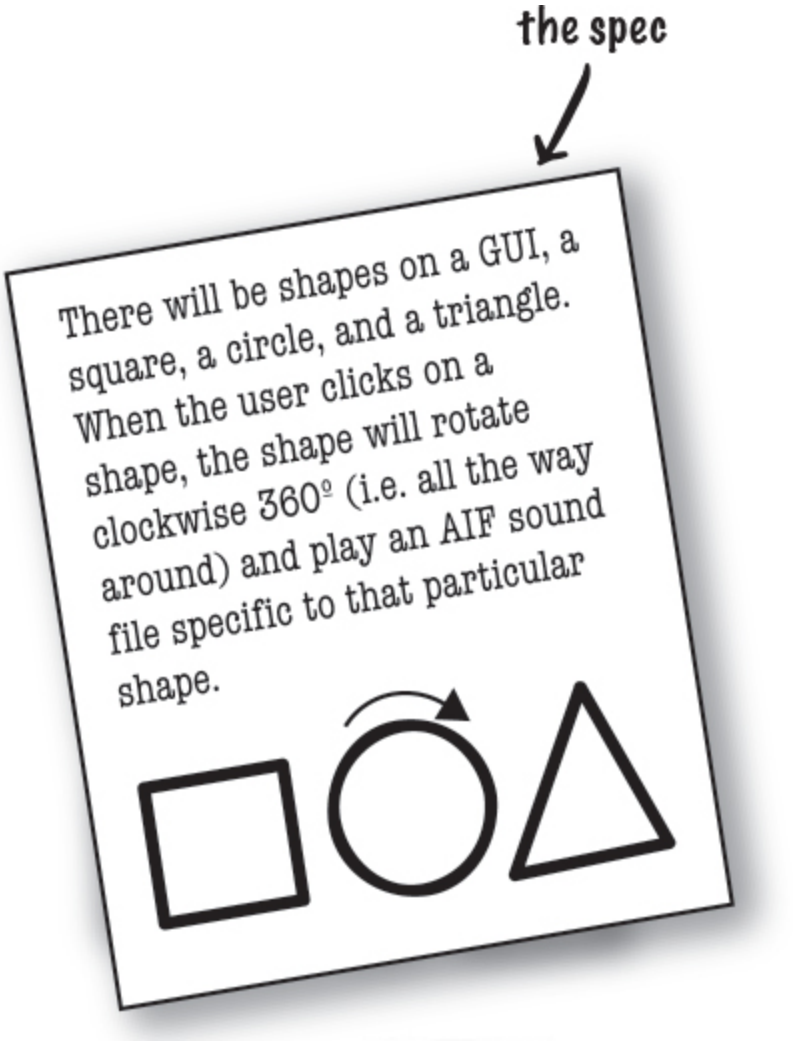
Chair Wars

(or How Objects Can Change Your Life)

Once upon a time in a software shop, two programmers were given the same spec and told to "build it". The Really Annoying Project Manager forced the two coders to compete, by promising that whoever delivers first gets a cool Aeron™ chair and adjustable height standing desk like all the Silicon Valley guys have. Larry, the procedural programmer, and Brad, the OO guy, both knew this would be a piece of cake.

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

Brad, meanwhile, kicked back at the cafe and thought to himself, "What are the ***things*** in this program... who are the key *players*?" He first thought of **The Shapes**. Of course, there were other objects he thought of like the User, the Sound, and the Clicking event. But he already had a library of code for those pieces, so he focused on building Shapes. Read on to see how Brad and Larry built their programs, and for the answer to your burning question, ***"So, who got the Aeron and the desk?"***



Image

In Larry's cube

As he had done a gazillion times before, Larry set about writing his **Important Procedures**. He wrote **rotate** and **playSound** in no time.

```
rotate(shapeNum) {
   // make the shape rotate 360º
}
playSound(shapeNum) {
   // use shapeNum to lookup which
   // AIF sound to play, and play it
}
```

At Brad’s laptop at the cafe

Brad wrote a ***class*** for each of the three shapes

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

But wait! There’s been a spec change.

“OK, *technically* you were first, Larry,” said the Manager, “but we have to add just one tiny thing to the program. It’ll be no problem for crack programmers like you two.”

“If I had a dime for every time I’ve heard that one”, thought Larry, knowing that spec-change-no-problem was a fantasy. *“And yet Brad looks strangely serene. What’s up with that?”* Still, Larry held tight to his core belief that the OO way, while cute, was just slow. And that if you wanted to change his mind, you’d have to pry it from his cold, dead, carpal-tunnelled hands.

Back in Larry’s cube

The rotate procedure would still work; the code used a lookup table to match a shapeNum to an actual shape graphic. But ***playSound would have to change.***

```
playSound(shapeNum) {
   // if the shape is not an amoeba,
      // use shapeNum to lookup which
      // AIF sound to play, and play it
   // else
      // play amoeba .mp3 sound
   }
```

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

At Brad's laptop at the beach

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

Larry snuck in just moments ahead of Brad.

(Hah! So much for that foofy OO nonsense). But the smirk on Larry's face melted when the Really Annoying Project Manager said (with that tone of disappointment), "Oh, no, *that's* not how the amoeba is supposed to rotate..."

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

1. **determine the rectangle that surrounds the shape**
2. **calculate the center of that rectangle, and rotate the shape around that point.**

But the amoeba shape was supposed to rotate around a point on one *end*, like a clock hand.

"I'm toast." thought Larry, visualizing charred Wonderbread™. "Although, hmmmm. I could just add another if/else to the rotate procedure, and then just hard-code the rotation point code for the amoeba. That probably won't break anything." But the little voice at the back of his head said, "*Big Mistake. Do you honestly think the spec won't change again?"*

Back in Larry’s cube

He figured he better add rotation point arguments to the rotate procedure. ***A lot of code was affected***. Testing, recompiling, the whole nine yards all over again. Things that used to work, didn’t.

```
rotate(shapeNum, xPt, yPt) {
// if the shape is not an amoeba,
   // calculate the center point
   // based on a rectangle,
   // then rotate
// else
   // use the xPt and yPt as
   // the rotation point offset
   // and then rotate
}
```

At Brad’s laptop on his lawn chair at the Telluride Bluegrass Festival

Without missing a beat, Brad modified the rotate **method**, but only in the Amoeba class. ***He never touched the tested, working, compiled code*** for the other parts of the program. To give the Amoeba a rotation point, he added an **attribute** that all Amoebas would have. He modified, tested, and delivered (wirelessly) the revised program during a single Bela Fleck set.

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

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

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

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

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

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

You can read this as, **"Square inherits from Shape"**, **"Circle inherits from Shape"**, and so on. I removed rotate() and playSound() from the other shapes, so now there's only one copy to maintain.

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

What about the Amoeba rotate()?

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

BRAD: Method.

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

BRAD: That's the last step. The Amoeba class **overrides** the methods of the Shape class. Then at runtime, the JVM knows exactly which rotate() method to run when someone tells the Amoeba to rotate.

LARRY: How do you “tell” an Amoeba to do something? Don’t you have to call the procedure, sorry—*method*, and then tell it *which* thing to rotate?

BRAD: That’s the really cool thing about OO. When it’s time for, say, the triangle to rotate, the program code invokes (calls) the rotate() method *on the triangle object*. The rest of the program really doesn’t know or care *how* the triangle does it. And when you need to add something new to the program, you just write a new class for the new object type, so the **new objects will have their own behavior.**

The suspense is killing me. Who got the chair and desk?

Amy from the second floor.

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

WHAT DO YOU LIKE ABOUT OO?

“It helps me design in a more natural way. Things have a way of evolving.”

-Joy, 27, software architect

“Not messing around with code I’ve already tested, just to add a new feature.”

-Brad, 32, programmer

“I like that the data and the methods that operate on that data are together in one class.”

-Josh, 22, beer drinker

“Reusing code in other applications. When I write a new class, I can make it flexible enough to be used in something new, later.”

-Chris, 39, project manager

“I can’t believe Chris just said that. He hasn’t written a line of code in 5 years.”

-Daryl, 44, works for Chris

“Besides the chair?”

-Amy, 34, programmer

 Inline Brain Power

Time to pump some neurons.

You just read a story bout a procedural programmer going head-to-head with an OO programmer. You got a quick overview of some key OO concepts including classes, methods, and attributes. We’ll spend the rest of the chapter looking at classes and objects (we’ll return to inheritance and overriding in later chapters).

Based on what you've seen so far (and what you may know from a previous OO language you've worked with), take a moment to think about these questions:

What are the fundamental things you need to think about when you design a Java class? What are the questions you need to ask yourself? If you could design a checklist to use when you're designing a class, what would be on the checklist?

METACOGNITIVE TIP

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

When you design a class, think about the objects that will be created from that class t ype. Think about:

Things an object ***knows*** about itself are called **instance variables**. They represent an object's state (the data), and can have unique values for each object of that type.

Think of instance as another way of saying object.

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

example, Alarm objects have an instance variable to hold the alarmTime, and two methods for getting and setting the alarmTime.

So objects have instance variables and methods, but those instance variables and methods are designed as part of the class.

Inline **sharpen your Pencil**

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

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

A class is not an object.

(but it's used to construct them)

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

LOOK AT IT THIS WAY...

An object is like one entry in your address book.

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

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

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

Making your first object

So what does it take to create and use an object? You need *two* classes. One class for the type of object you want to use (Dog, AlarmClock, Television, etc.) and another class to *test* your new class. The *tester* class is where you put the main method, and in that main() method you create and access objects of your new class type. The tester class has only one job: to *try out* the methods and variables of your new object class type.

From this point forward in the book, you'll see two classes in many of our examples. One will be the *real* class – the class whose objects we really want to use, and the other class will be the *tester* class, which we call <*whateverYourClassNameIs*> **`TestDrive`**. For example, if we make a **`Bungee`** class, we'll need a **`BungeeTestDrive`** class as well. Only the <*someClassName*>**`TestDrive`** class will have a main() method, and its sole purpose is to create objects of your new type (the not-the-tester class), and then use the dot operator (.) to access the methods and variables of the

new objects. This will all be made stunningly clear by the following examples. No, *really.*

THE DOT OPERATOR (.)

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

```
// make a new object
Dog d = new Dog();

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

d.bark();

// set its size using the
// dot operator

d.size = 40;
```

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

Making and testing Movie objects

```
class Movie {
  String title;
  String genre;
  int rating;

  void playIt() {
    System.out.println("Playing the movie");
```

```
  }
}

public class MovieTestDrive {
  public static void main(String[] args) {
    Movie one = new Movie();
    one.title = "Gone with the Stock";
    one.genre = "Tragic";
    one.rating = -2;
    Movie two = new Movie();
    two.title = "Lost in Cubicle Space";
    two.genre = "Comedy";
    two.rating = 5;
    two.playIt();
    Movie three = new Movie();
    three.title = "Byte Club";
    three.genre = "Tragic but ultimately uplifting";
    three.rating = 127;
  }
}
```

The MovieTestDrive class creates objects (instances) of the Movie class and uses the dot operator (.) to set the instance variables to a specific value. The MovieTest-Drive class also invokes (calls) a method on one of the objects. Fill in the chart to the right with the values the three objects have at the end of main().

Quick! Get out of main!

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

The two uses of main:

- to **test** your real class

- to **launch/start** your Java **application**

A real Java application is nothing but objects talking to other objects. In this case, *talking* means objects calling methods on one another. On the previous page, and in Chapter 4 , we look at using a main() method from a separate TestDrive class to create and test the methods and variables of another class. In chapter 6 we look at using a class with a main() method to start the ball rolling on a *real* Java application (by making objects and then turning those objects loose to interact with other objects, etc.)

As a ‘sneak preview’, though, of how a real Java application might behave, here’s a little example. Because we’re still at the earliest stages of learning Java, we’re working with a small toolkit, so you’ll find this program a little clunky and inefficient. You might want to think about what you could do to improve it, and in later chapters that’s exactly what we’ll do. Don’t worry if some of the code is confusing; the key point of this example is that objects talk to objects.

The Guessing Game

Summary:

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

Classes:

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

The Logic:

1. The GameLauncher class is where the application starts; it has the main() method.

2. In the main() method, a GuessGame object is created, and its startGame() method is called.

3. The GuessGame object's startGame() method is where the entire game plays out. It creates three players, then "thinks" of a random number (the target for the players to guess). It then asks each player to guess, checks the result, and either prints out information about the winning player(s) or asks them to guess again.

Running the Guessing Game

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

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

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

JAVA TAKES OUT THE GARBAGE

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

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

There are no Dumb Questions

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

A: There isn't a concept of 'global' variables and methods in a Java OO program. In practical use, however, there are times when you want a method (or a constant) to be available to any code running in any part of your program. Think of the `random()` method in the Phrase-O-Matic app; it's a method that should be callable from anywhere. Or what about a

constant like *pi*? You'll learn in chapter 10 that marking a method as `public` and `static` makes it behave much like a 'global'. Any code, in any class of your application, can access a public static method. And if you mark a variable as `public, static`, and `final` – you have essentially made a globally-available *constant*.

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

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

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

A: A Java program is a pile of classes (or at least *one* class). In a Java application, *one* of the classes must have a main method, used to start-up the program. So as a programmer, you write one or more classes. And those classes are what you deliver. If the end-user doesn't have a JVM, then you'll also need to include that with your application's classes, so that they can run your program. There are a number of installer programs that let you bundle your classes with a variety of JVM's (say, for different platforms), and put it all on a CD-ROM. Then the end-user can install the correct version of the JVM (assuming they don't already have it on their machine.)

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

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

BULLET POINTS INLINE

- Object-oriented programming lets you extend a program without having to touch previously-tested, working code.
- All Java code is defined in a **class**.
- A class describes how to make an object of that class type. **A class is like a blueprint.**
- An object can take care of itself; you don’t have to know or care *how* the object does it.
- An object **knows** things and **does** things.
- Things an object knows about itself are called **instance variables**. They represent the *state* of an object.
- Things an object does are called **methods**. They represent the *behavior* of an object.
- When you create a class, you may also want to create a separate test class which you’ll use to create objects of your new class type.
- A class can **inherit** instance variables and methods from a more abstract **superclass**.
- At runtime, a Java program is nothing more than objects ‘talking’ to other objects.

BE the compiler

Each of the Java files on this page represents a complete source file. Your job is to play compiler and determine whether each of these files will compile. If they won't compile, how would you fix them, and if they do compile, what would be their output?

A

```
class TapeDeck {

  boolean canRecord = false;

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

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

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

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

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

B

```
class DVDPlayer {

  boolean canRecord = false;

  void recordDVD() {
    System.out.println(“DVD recording”);
  }
}
```

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

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

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

Code Magnets

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

Pool Puzzle

Your ***job*** is to take code snippets from the pool and place them into the blank lines in the code. You **may** use the same snippet more than once, and you won't need to use all the snippets. Your ***goal*** is to make classes that will compile and run and produce the output listed. Some of the exercises and puzzles in this book might have more than one correct answer. If you find another correct answer, give yourself bonus points!

Output

Bonus Question !

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

```
public class EchoTestDrive {
  public static void main(String [] args) {
    Echo e1 = new Echo();
    ____________________________
    int x = 0;
    while ( ____________ ) {
      e1.hello();
      ____________________________
      if ( _____________ ) {
        e2.count = e2.count + 1;
    }
    if ( _____________ ) {
      e2.count = e2.count + e1.count;
    }
    x = x + 1;
  }
  System.out.println(e2.count);
 }
}
```

```
class _____________ {
  int __________ = 0;
  void ____________ {
    System.out.println(“helloooo... “);
  }
}
```

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

A bunch of Java components, in full costume, are playing a party game, “Who am I?” They give you a clue, and you try to guess who they are, based on what they say. Assume they always tell the truth about themselves. If they happen to say something that could be true for more than one of them, choose all for whom that sentence can apply. Fill in the blanks next to the sentence with the names of one or more attendees. The first one’s on us.

Exercise Solutions

Code Magnets:

```
class DrumKit {

  boolean topHat = true;
  boolean snare = true;

  void playTopHat() {
    System.out.println("ding ding da-ding");
  }

  void playSnare() {
    System.out.println("bang bang ba-bang");
  }
}

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

    DrumKit d = new DrumKit();
    d.playSnare();
    d.snare = false;
    d.playTopHat();
```

```
    if (d.snare == true) {
      d.playSnare();
   }
  }
}
```

Inline Puzzle Solutions

Pool Puzzle

```
public class EchoTestDrive {
  public static void main(String [] args) {
    Echo e1 = new Echo();
    Echo e2 = new Echo( ); // correct answer
    - or -
    Echo e2 = e1; // bonus “24” answer
    int x = 0;
    while ( x < 4 ) {
      e1.hello();
      e1.count = e1.count + 1;
      if ( x == 3 ) {
        e2.count = e2.count + 1;
      }
      if ( x > 0 ) {
        e2.count = e2.count + e1.count;
      }
      x = x + 1;
  }
  System.out.println(e2.count);
}
```

```
class Echo {
  int count = 0;
  void hello( ) {
    System.out.println(“hellooooo... “);
  }
}
```

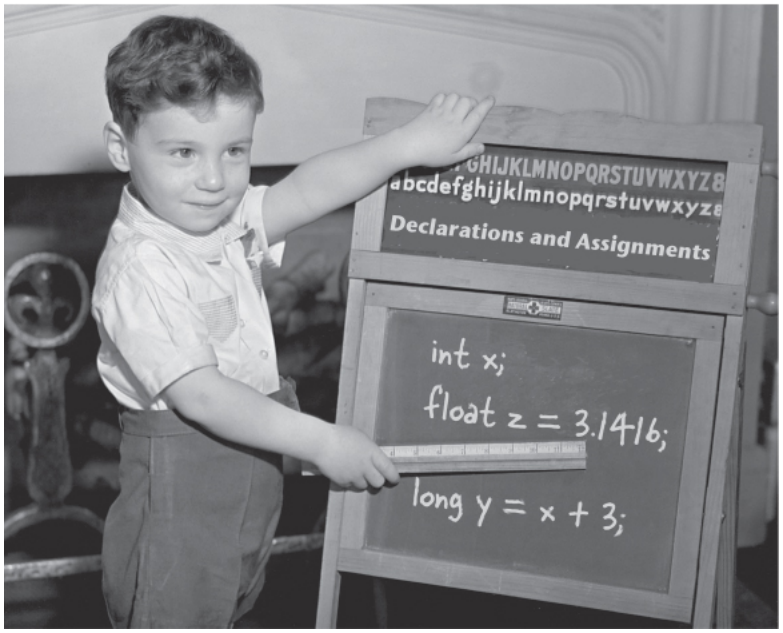
Who am I?

I am compiled from a .java file.	**class**
My instance variable values can be different from my buddy’s values.	**object**
I behave like a template.	**class**
I like to do stuff.	**object, method**
I can have many methods.	**class, object**
I represent ‘state’.	**instance variable**
I have behaviors.	**object, class**
I am located in objects.	**method, instance variable**
I live on the heap.	**object**
I am used to create object instances.	**class**
My state can change.	**object, instance variable**
I declare methods.	**class**
I can change at runtime.	**object, instance variable**

Note

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

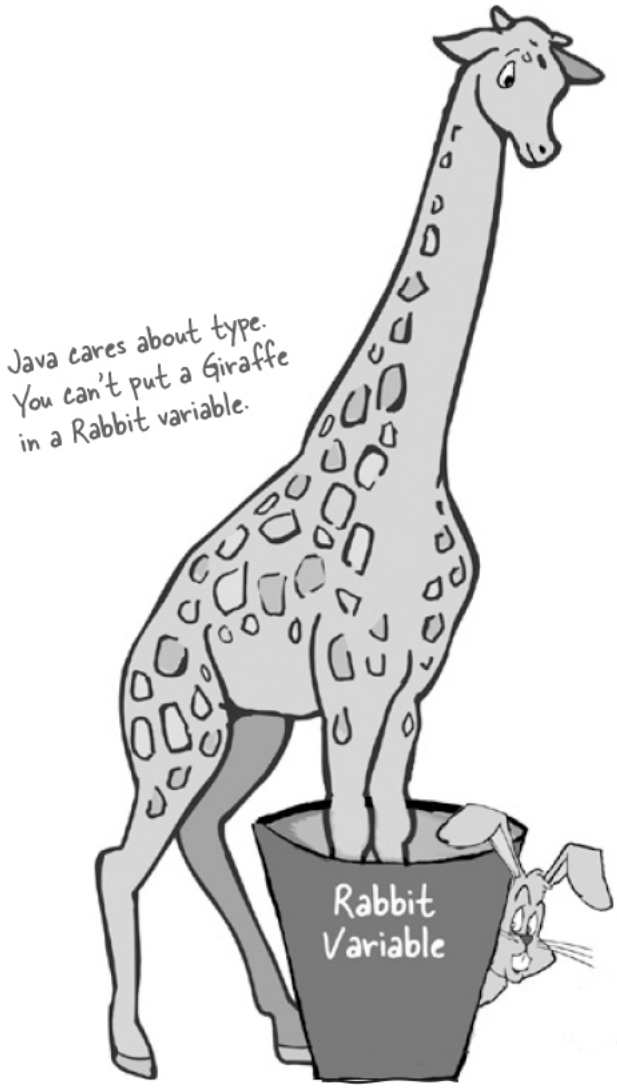
Chapter 3. Primitives And References: Know Your Variables



Variables come in two flavors: primitive and reference. So far you've used variables in two places—as object **state** (instance variables), and as **local** variables (variables declared within a *method*). Later, we'll use variables as **arguments** (values sent to a method by the calling code), and as **return types** (values sent back to the caller of the method). You've seen variables declared as simple **primitive** integer values (type `int`). You've seen variables declared as something more **complex** like a String or an array. But **there's gotta be more to life** than integers, Strings, and arrays. What if you have a PetOwner object with a Dog instance variable? Or a Car with an Engine? In this chapter we'll unwrap the mysteries of Java types and look at what you can *declare* as a variable, what you can *put* in a variable, and what you can *do* with a variable. And we'll finally see what life is *truly* like on the garbage-collectible heap.

Declaring a variable

Java cares about type. It won't let you do something bizarre and dangerous like stuff a Giraffe reference into a Rabbit variable—what happens when someone tries to ask the so-called *Rabbit* to hop()? And it won't let you put a floating point number into an integer variable, unless you *acknowledge to the compiler* that you know you might lose precision (like, everything after the decimal point).



The compiler can spot most problems:

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

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

For all this type-safety to work, you must declare the type of your variable. Is it an integer? a Dog?

A single character? Variables come in two flavors: ***primitive*** and ***object reference***. Primitives hold fundamental values (think: simple bit patterns) including integers, booleans, and floating point numbers. Object references hold, well, *references* to *objects* (gee, didn't *that* clear it up.)

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

variables must have a type

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

variables must have a name

Note

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

“I’d like a double mocha, no, make it an int.”

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

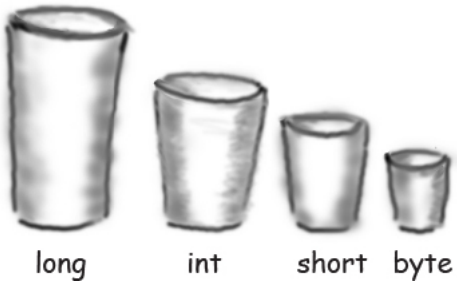
A variable is just a cup. A container. It *holds* something.

It has a size, and a type. In this chapter, we’re going to look first at the variables (cups) that hold **primitives**, then a little later we’ll look at cups that hold *references to objects*. Stay with us here on the whole cup analogy—as simple as it is right now, it’ll give us a common way to look at things when the discussion gets more complex. And that’ll happen soon.

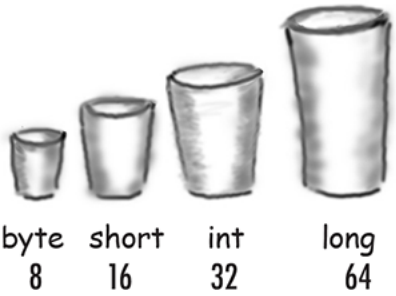
Primitives are like the cups they have at the coffeehouse. If you’ve been to a Starbucks, you know what we’re talking about here. They come in different sizes, and each has a name like ‘short’, ‘tall’, and, “I’d like a ‘grande’ mocha half-caff with extra whipped cream”.

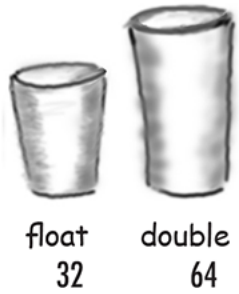
You might see the cups displayed on the counter, so you can order appropriately:

And in Java, primitives come in different sizes, and those sizes have names. When you declare a variable in Java, you must declare it with a specific type. The four containers here are for the four integer primitives in Java.



Each cup holds a value, so for Java primitives, rather than saying, “I’d like a tall french roast”, you say to the compiler, “I’d like an int variable with the number 90 please.” Except for one tiny difference... in Java you also have to give your cup a *name*. So it’s actually, “I’d like an int please, with the value of 2486, and name the variable ***height***.” Each primitive variable has a fixed number of bits (cup size). The sizes for the six numeric primitives in Java are shown below:





Primitive Types

Type	Bit Depth	Value Range

boolean and char

boolean	(JVM-specific)	***true*** or ***false***
char	16 bits	0 to 65535

numeric (all are signed)

integer

byte	8 bits	-128 to 127
short	16 bits	-32768 to 32767
int	32 bits	-2147483648 to 2147483647
long	64 bits	-huge to huge

floating point

float	32 bits	varies
double	64 bits	varies

Primitive declarations with assignments:

int x;

x = 234;

byte b = 89;

boolean isFun = true;

double d = 3456.98;

char c = ‘f’;

int z = x;

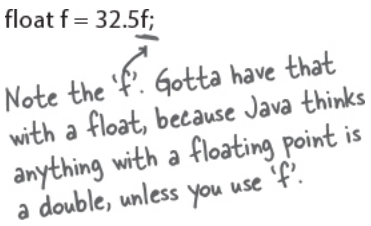
boolean isPunkRock;

isPunkRock = false;

boolean powerOn;

powerOn = isFun;

long big = 3456789;



You really don't want to spill that...

Be sure the value can fit into the variable.

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

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

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

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

Why doesn't this work, you ask? After all, the value of *x* is 24, and 24 is definitely small enough to fit into a byte. *You* know that, and *we* know that, but all the compiler cares about is that you're trying to put a big thing into a small thing, and there's the *possibility* of spilling. Don't expect the compiler to know what the value of *x* is, even if you happen to be able to see it literally in your code.

You can assign a value to a variable in one of several ways including:

- type a *literal* value after the equals sign (x=***12***, isGood = ***true***, etc.)
- assign the value of one variable to another (x = y)
- use an expression combining the two (x = y + ***43***)

In the examples below, the literal values are in bold italics:

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

SHARPEN YOUR PENCIL

The compiler won't let you put a value from a large cup into a small one. But what about the other way—pouring a small cup into a big one? ***No problem.***

Based on what you know about the size and type of the primitive variables, see if you can figure out which of these are legal and which aren't. We haven't covered all the rules yet, so on some of these you'll have to use your best judgment. ***Tip:*** The compiler always errs on the side of safety.

_____

From the following list, ***Circle*** the statements that would be legal if these lines were in a single method:

```
1. int x = 34.5;
2. boolean boo = x;
3. int g = 17;
4. int y = g;
5. y = y + 10;
6. short s;
7. s = y;
8. byte b = 3;
9. byte v = b;
10. short n = 12;
11. v = n;
12. byte k = 128;
```

Back away from that keyword!

You know you need a name and a type for your variables.

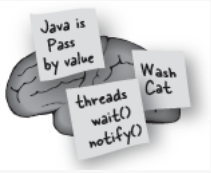
You already know the primitive types.

But what can you use as names? The rules are simple. You can name a class, method, or variable according to the following rules (the real rules are slightly more flexible, but these will keep you safe):

- **It must start with a letter, underscore (_), or dollar sign ($). You can't start a name with a number.**
- **After the first character, you can use numbers as well. Just don't start it with a number.**
- **It can be anything you like, subject to those two rules, just so long as it isn't one of Java's reserved words.**

Reserved words are keywords (and other things) that the compiler recognizes. And if you really want to play confuse-a-compiler, then just *try* using a reserved word as a name.

MAKE IT STICK



The eight primitive types are:

boolean **c**har **b**yte **s**hort **i**nt **l**ong **f**loat **d**ouble

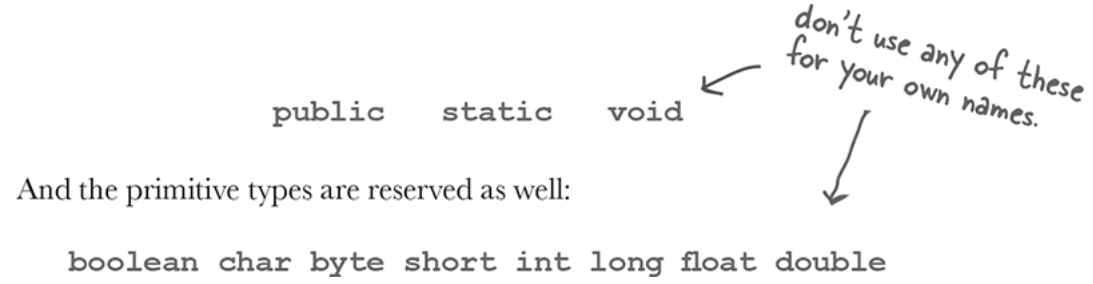
And here's a mnemonic for remembering them:

Be **C**areful! **B**ears **S**houldn't **I**ngest **L**arge **F**urry **D**ogs

If you make up your own, it'll stick even better.

*B_ C_ B_ S_ I_ L_ F_ D_*

You've already seen some reserved words:



But there are a lot more we haven't discussed yet. Even if you don't need to know what they mean, you still need to know you can't use 'em yourself. ***Do not****–under any circumstances–****try to memorize these now.*** To make room for these in your head, you'd probably have to lose something else. Like where your car is parked. Don't worry, by the end of the book you'll have most of them down cold.

This table reserved.

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

Java's keywords, reserved words, and special identifiers. If you use these for names, the compiler will *probably* be very, *very* upset.

Controlling your Dog object

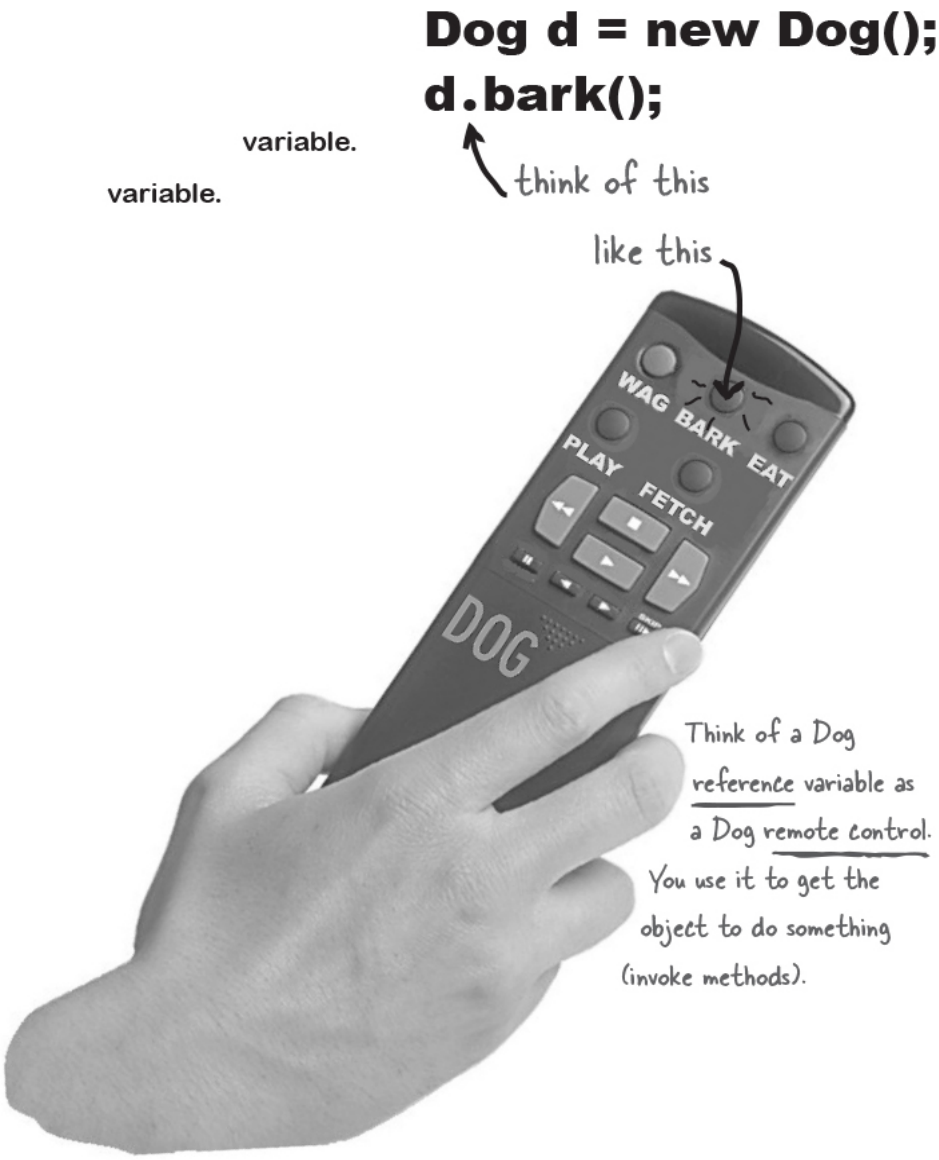
You know how to declare a primitive variable and assign it a value. But now what about non-primitive variables? In other words, *what about objects?*

- **There is actually no such thing as an object variable.**
- **There's only an object reference variable.**
- **An object reference variable holds bits that represent a way to access an object.**
- **It doesn't hold the object itself, but it holds something like a pointer. Or an address. Except, in Java we don't really know what is inside a reference variable. We do know that whatever it is, it represents one and only one object. And the JVM knows how to use the reference to get to the object.**

You can't stuff an object into a variable. We often think of it that way... we say things like, "I passed the String to the System.out. println() method." Or, "The method returns a Dog", or, "I put a new Foo object into the variable named myFoo."

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

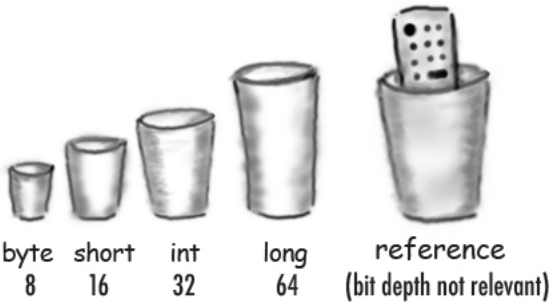
Although a primitive variable is full of bits representing the actual ***value*** of the variable, an object reference variable is full of bits representing ***a way to get to the object.***



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

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

means, "use the object referenced by the variable myDog to invoke the bark() method." When you use the dot operator on an object reference variable, think of it like pressing a button on the remote control for that object.



An object reference is just another variable value.

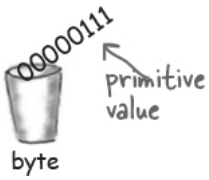
Something that goes in a cup.

Only this time, the value is a remote control.

Primitive Variable

```
byte x = 7;
```

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

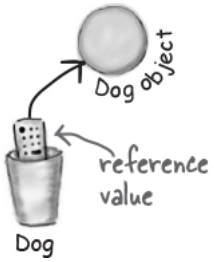


Reference Variable

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

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

The Dog object itself does not go into the variable!



NOTE

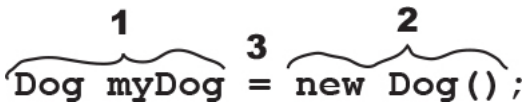
With primitive variables, the value of the variable is... the *value* (5, -26.7, 'a').

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

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

We don’t care how many 1’s and 0’s there are in a reference variable.It’s up to each JVM and the phase of the moon.

The 3 steps of object declaration, creation and assignment



❶ Declare a reference variable

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

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

❷ Create an object

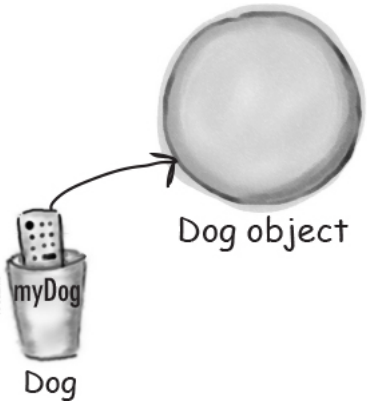
```
Dog myDog = new Dog();
```

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

❸ Link the object and the reference

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

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



There are no Dumb Questions

Q: How big is a reference variable?

A: You don't know. Unless you're cozy with someone on the JVM's development team, you don't know how a reference is represented. There are pointers in there somewhere, but you can't access them. You won't need to. (OK, if you insist, you might as well just imagine it to be a 64-bit value.) But when you're talking about memory allocation issues, your Big Concern should be about how many *objects* (as opposed to object *references*) you're creating, and how big *they* (the *objects*) really are.

Q: So, does that mean that all object references are the same size, regardless of the size of the actual objects to which they refer?

A: Yep. All references for a given JVM will be the same size regardless of the objects they reference, but each JVM might have a different way of representing references, so references on one JVM may be smaller or larger than references on another JVM.

Q: Can I do arithmetic on a reference variable, increment it, you know – C stuff?

A: Nope. Say it with me again, "Java is not C."

Java Exposed

This week's interview: Object Reference

HeadFirst: So, tell us, what's life like for an object reference?

Reference: Pretty simple, really. I'm a remote control and I can be programmed to control different objects.

HeadFirst: Do you mean different objects even while you're running? Like, can you refer to a Dog and then five minutes later refer to a Car?

Reference: Of course not. Once I'm declared, that's it. If I'm a Dog remote control then I'll never be able to point (oops – my bad, we're not supposed to say *point*) I mean *refer* to anything but a Dog.

HeadFirst: Does that mean you can refer to only one Dog?

Reference: No. I can be referring to one Dog, and then five minutes later I can refer to some *other* Dog. As long as it's a Dog, I can be redirected (like reprogramming your remote to a different TV) to it. Unless... no never mind.

HeadFirst: No, tell me. What were you gonna say?

Reference: I don't think you want to get into this now, but I'll just give you the short version – if I'm marked as final, then once I am assigned a Dog, I can never be reprogrammed to anything else but *that* one and only Dog. In other words, no other object can be assigned to me.

HeadFirst: You're right, we don't want to talk about that now. OK, so unless you're `final`, then you can refer to one Dog and then refer to a different Dog later. Can you ever refer to *nothing at all*? Is it possible to not be programmed to anything?

Reference: Yes, but it disturbs me to talk about it.

HeadFirst: Why is that?

Reference: Because it means I'm `null`, and that's upsetting to me.

HeadFirst: You mean, because then you have no value?

Reference: Oh, `null` *is* a value. I'm still a remote control, but it's like you brought home a new universal remote control and you don't have a TV. I'm not programmed to control anything. They can press my buttons all day long, but nothing good happens. I just feel so... useless. A waste of bits. Granted, not that many bits, but still. And that's not the worst part. If I am the only reference to a particular object, and then I'm set to `null` (deprogrammed), it means that now *nobody* can get to that object I had been referring to.

HeadFirst: And that's bad because...

Reference: You have to *ask*? Here I've developed a relationship with this object, an intimate connection, and then the tie is suddenly, cruelly, severed. And I will never see that object again, because now it's eligible for [producer, cue tragic music] *garbage collection*. Sniff. But do you think programmers ever consider *that*? Snif. Why, *why* can't I be a primitive? *I hate being a reference.* The responsibility, all the broken attachments...

Life on the garbage-collectible heap

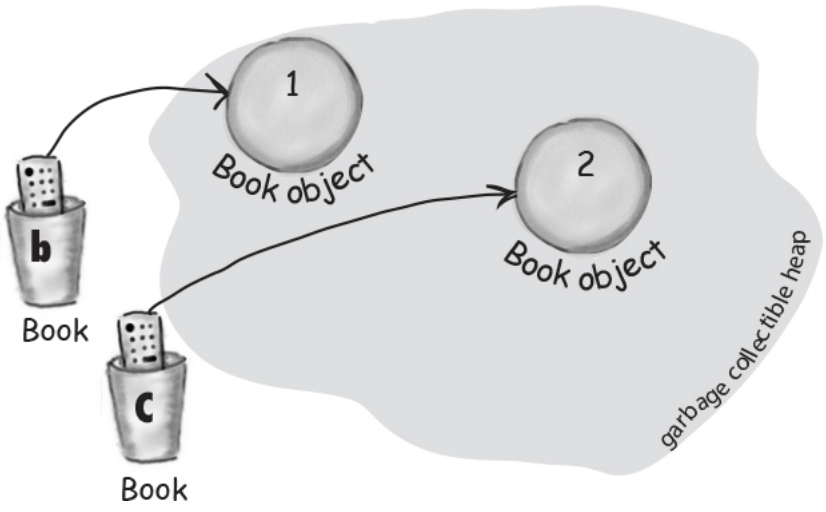
```
Book b = new Book();
Book c = new Book();
```

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

The two Book objects are now living on the heap.

References: 2

Objects: 2



```
Book d = c;
```

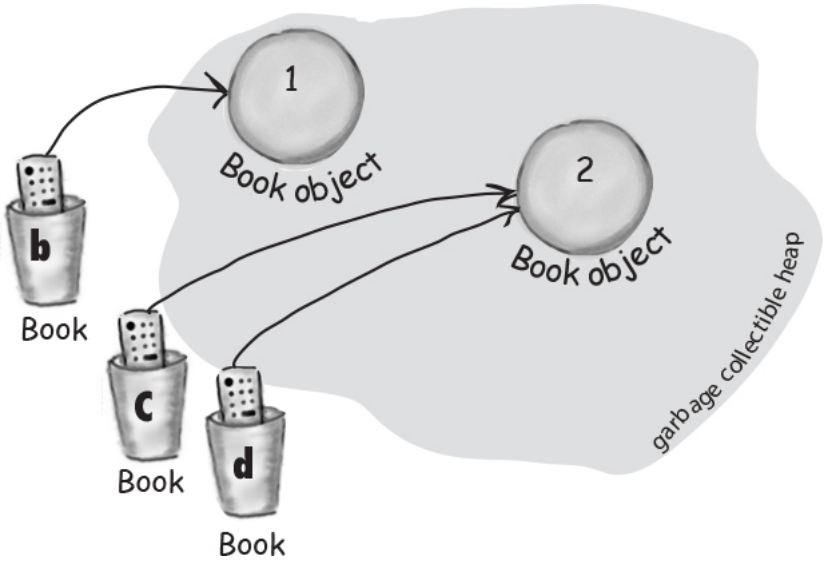
Declare a new Book reference variable. Rather than creating a new, third Book object, assign the value of variable ***c*** to variable ***d***. But what does this mean? It's like saying, "Take the bits in ***c***, make a copy of them, and stick that copy into ***d***."

Both c and d refer to the same object.

The c and d variables hold two different copies of the same value. Two remotes programmed to one TV.

References: 3

Objects: 2



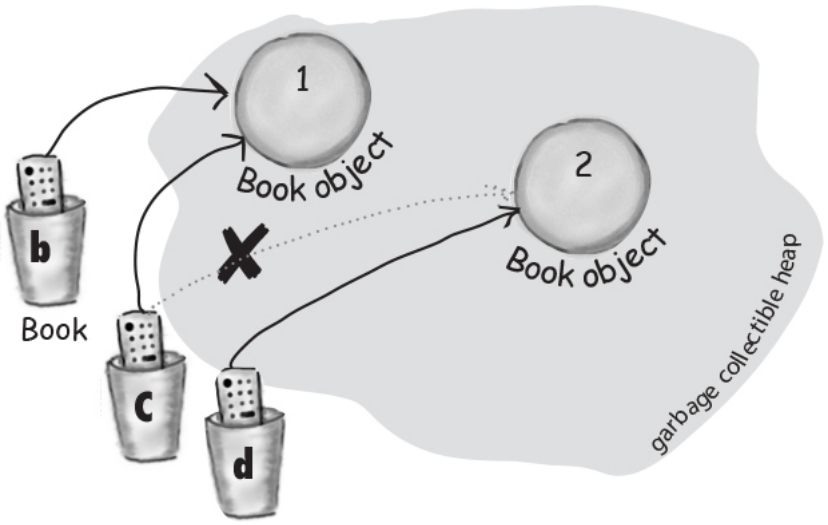

```
c = b;
```

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

Both b and c refer to the same object.

References: 3

Objects: 2



Life and death on the heap

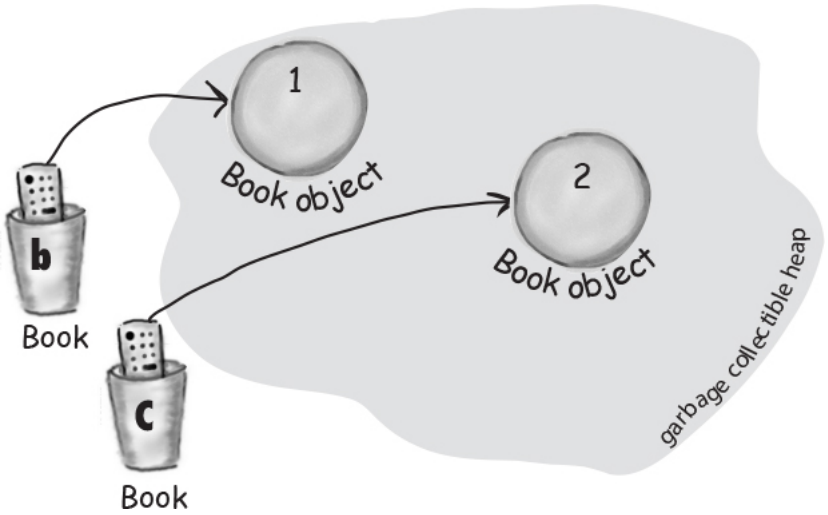
```
Book b = new Book();
Book c = new Book();
```

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

The two book objects are now living on the heap.

Active References: 2

Reachable Objects: 2




```
b = c;
```

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

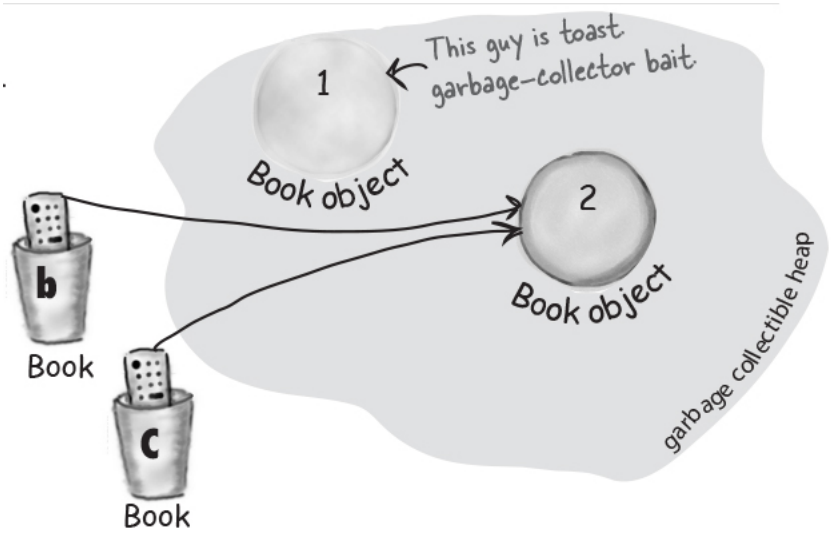
Both b and c refer to the same object. Object 1 is abandoned and eligible for Garbage Collection (GC).

Active References: 2

Reachable Objects: 1

Abandoned Objects: 1

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



`c = null;`

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

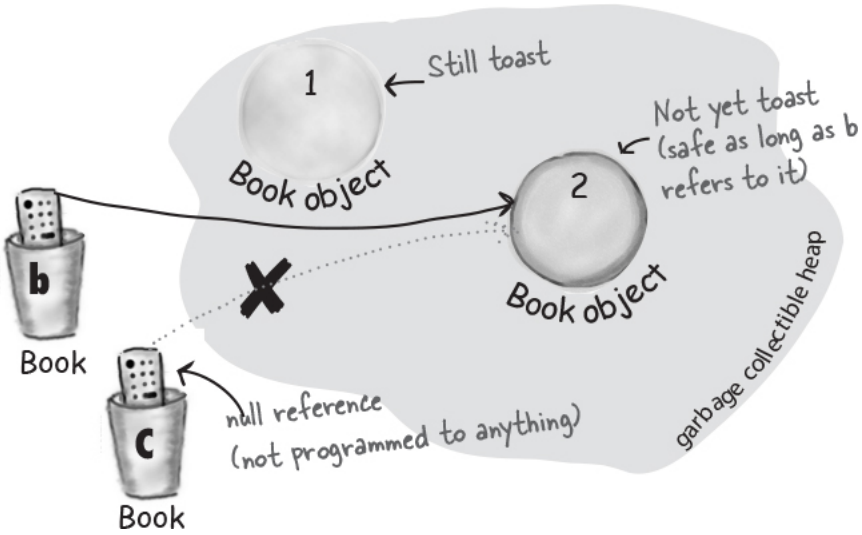
Object 2 still has an active reference (b), and as long as it does, the object is not eligible for GC.

Active References: 1

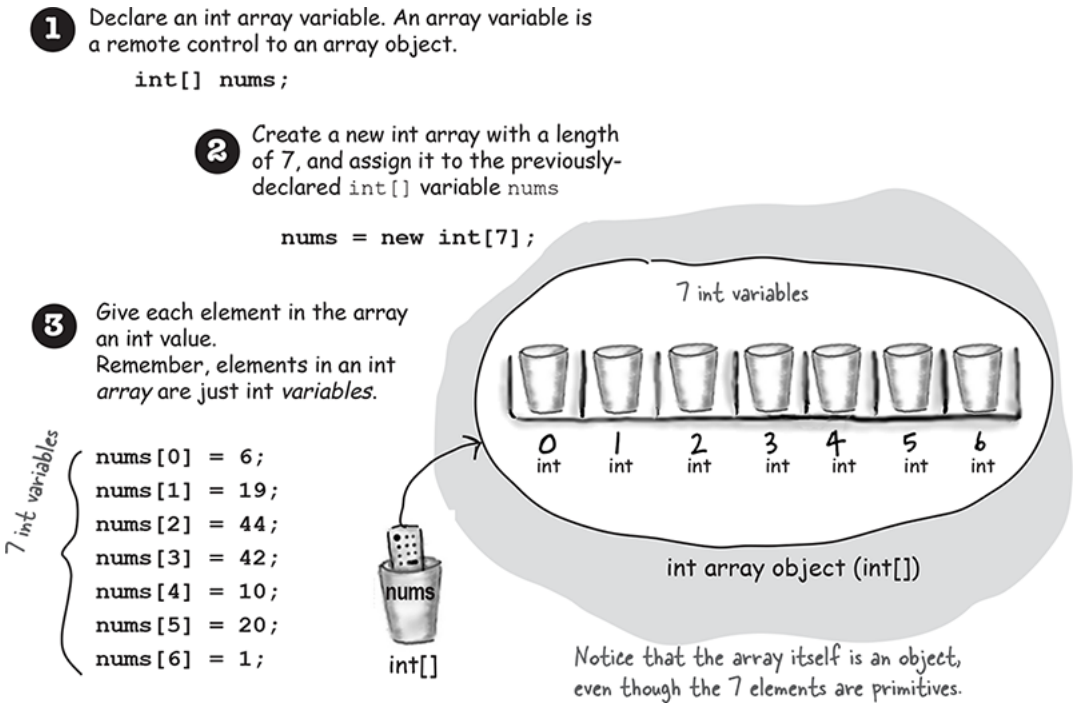
null References: 1

Reachable Objects: 1

Abandoned Objects: 1



An array is like a tray of cups



Arrays are objects too

The Java standard library includes lots of sophisticated data structures including maps, trees, and sets (see Appendix B), but arrays are great when you just want a quick, ordered, efficient list of things. Arrays give you fast random access by letting you use an index position to get to any element in the array.

Every element in an array is just a variable. In other words, one of the eight primitive variable types (think: Large Furry Dog) or a reference variable. Anything you would put in a *variable* of that type can be assigned to an *array element* of that type. So in an array of type int (int[]), each element can hold an int. In a Dog array (Dog[]) each element can hold... a Dog? No, remember that a reference variable just holds a reference (a remote control), not the object itself. So in a Dog array, each element can hold a *remote control* to a Dog. Of course, we still have to make the Dog objects... and you'll see all that on the next page.

Be sure to notice one key thing in the picture above – *the array is an object, even though it's an array of primitives.*

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

Make an array of Dogs

❶ Declare a Dog array variable

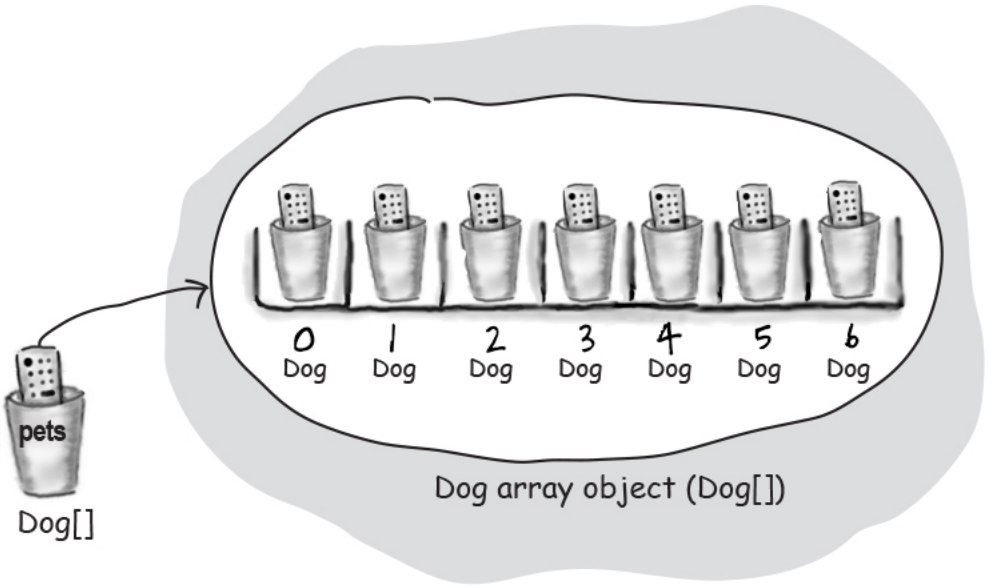
```
Dog[] pets;
```

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

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

What's missing?

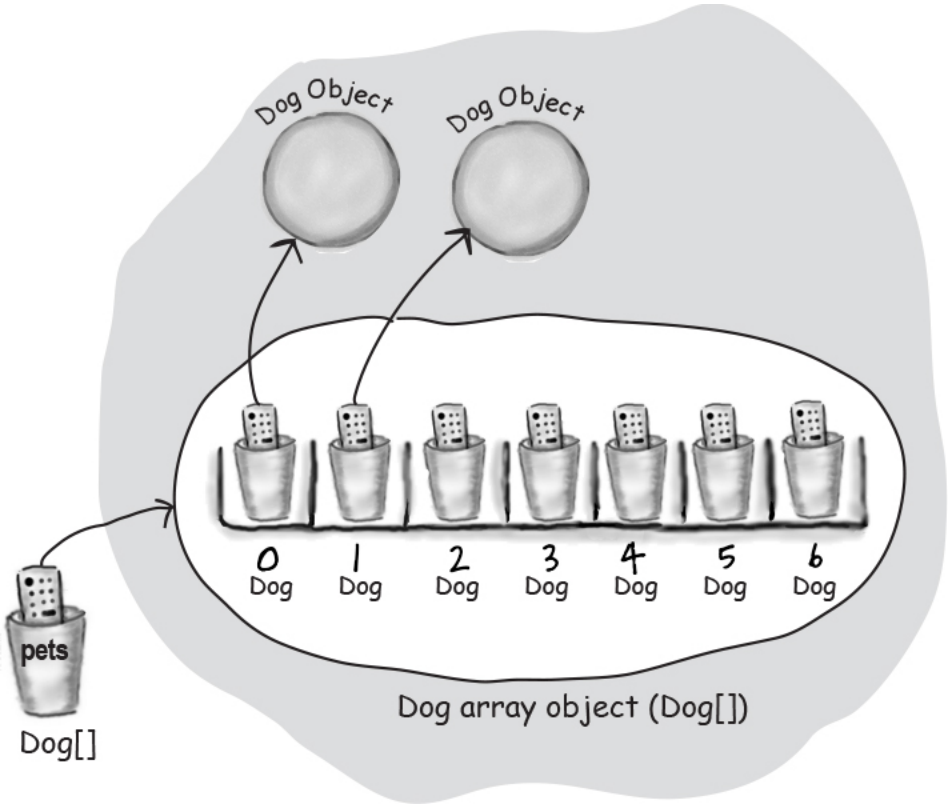
Dogs! We have an array of Dog *references*, but no actual Dog *objects*!



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

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

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

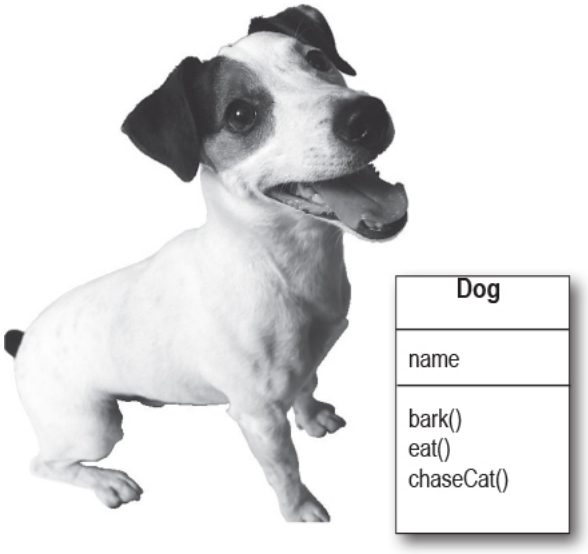


SHARPEN YOUR PENCIL

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

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

_____



JAVA CARES ABOUT TYPE.

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

For example, you can't put a Cat into a Dog array (it would be pretty awful if someone thinks that only Dogs are in the array, so they ask each one to bark, and then to their horror discover there's a cat lurking.) And you can't stick a `double` into an `int` array (spillage, remember?). You can, however, put a `byte` into an `int` array, because a `byte` will always fit into an `int` What happens if the Dog is in a Dog array? -sized cup. This is known as an **implicit widening. We'll get into the details later, for now just remember that the compiler won't let you put the wrong thing in an array, based on the array's declared type.**

Control your Dog

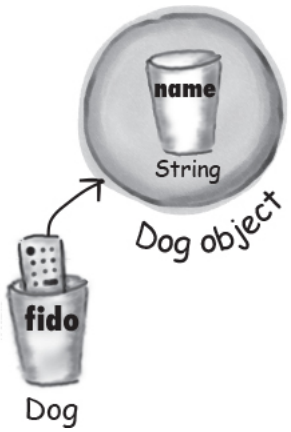
(with a reference variable)

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

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

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

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



What happens if the Dog is in a Dog array?

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

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

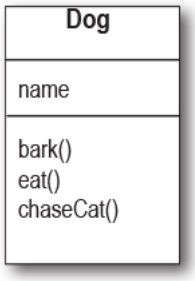
```
Dog[] myDogs = new Dog[3];
myDogs[0] = new Dog();
myDogs[0].name = "Fido";
myDogs[0].bark();
```

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

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

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

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

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

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

  public void bark() {
    System.out.println(name + " says Ruff!");
  }
  public void eat() {  }
  public void chaseCat() {  }
}
```

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

A Dog example



Output

```
File Edit Window Help Howl
%java Dog
null says Ruff!
last dog's name is Bart
Fred says Ruff!
Marge says Ruff!
Bart says Ruff!
```

BULLET POINTS

- Variables come in two flavors: primitive and reference.
- Variables must always be declared with a name and a type.
- A primitive variable value is the bits representing the value (5, 'a', true, 3.1416, etc.).
- A reference variable value is the bits representing a way to get to an object on the heap.
- A reference variable is like a remote control. Using the dot operator (.) on a reference variable is like pressing a button on the remote control to access a method or instance variable.
- A reference variable has a value of null when it is not referencing any object.
- An array is always an object, even if the array is declared to hold primitives. There is no such thing as a primitive array, only an array that *holds* primitives.

BE the compiler

Each of the Java files on this page represents a complete source file. Your job is to play compiler and determine whether each of these files will compile and run without exception. If they won't, how would you fix them?

A

```
class Books {
  String title;
  String author;
}

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

    Books [] myBooks = new Books[3];
    int x = 0;
    myBooks[0].title = “The Grapes of Java”;
    myBooks[1].title = “The Java Gatsby”;
    myBooks[2].title = “The Java Cookbook”;
    myBooks[0].author = “bob”;
    myBooks[1].author = “sue”;
    myBooks[2].author = “ian”;

    while (x < 3) {
      System.out.print(myBooks[x].title);
      System.out.print(“ by “);
      System.out.println(myBooks[x].author);
      x = x + 1;
   }
  }
}
```

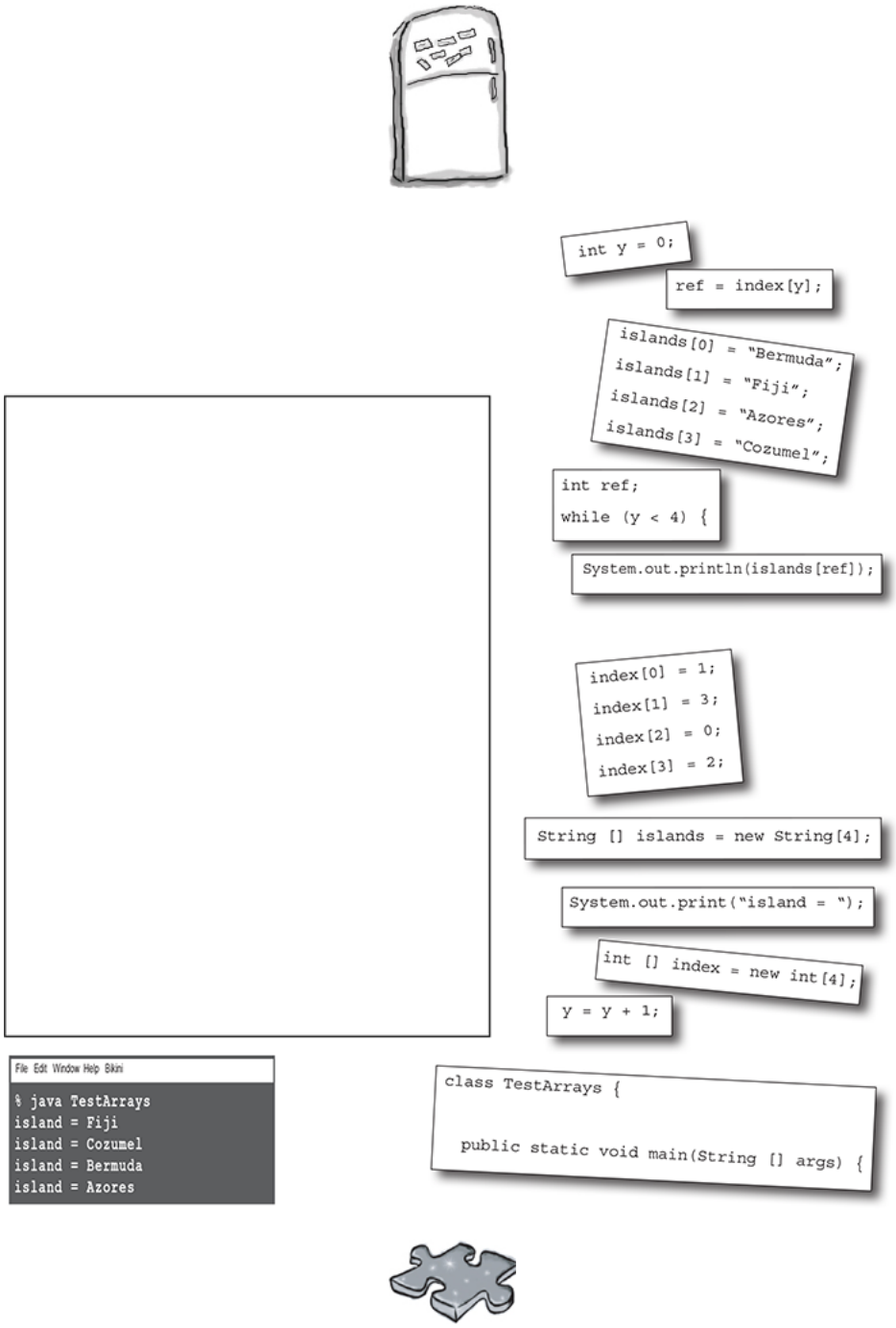
B

```
class Hobbits {

  String name;

  public static void main(String [] args) {

    Hobbits [] h = new Hobbits[3];
    int z = 0;

    while (z < 4) {
      z = z + 1;
      h[z] = new Hobbits();
      h[z].name = “bilbo”;
      if (z == 1) {
        h[z].name = “frodo”;
      }
      if (z == 2) {
        h[z].name = “sam”;
      }
      System.out.print(h[z].name + “ is a “);
      System.out.println(“good Hobbit name”);
    }
  }
}
```

Code Magnets

A working Java program is all scrambled up on the fridge. Can you reconstruct the code snippets to make a working Java program that produces the output listed below? Some of the curly braces fell on the floor and they

were too small to pick up, so feel free to add as many of those as you need!



Pool Puzzle

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

Output

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

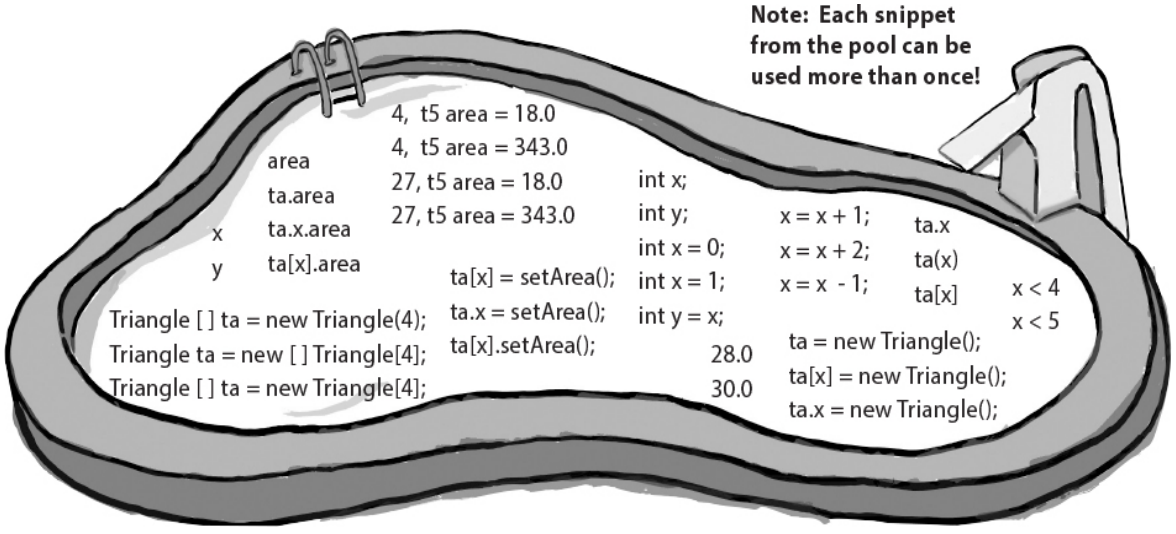
Bonus Question!

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

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

```
class Triangle {
  double area;
  int height;
  int length;
  public static void main(String [] args) {
    ____________
    _________________________
    while ( ___________ ) {
      _________________________
      ________.height = (x + 1) * 2;
      ________.length = x + 4;
      _________________________
      System.out.print("triangle "+x+", area");
      System.out.println(" = " + _______.area);
      ________________
    }
    _______________
    x = 27;
    Triangle t5 = ta[2];
    ta[2].area = 343;
    System.out.print("y = " + y);
    System.out.println(", t5 area = "+ t5.area);
  }
  void setArea() {
    _____________ = (height * length) / 2;
  }
}
```

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



A Heap o' Trouble

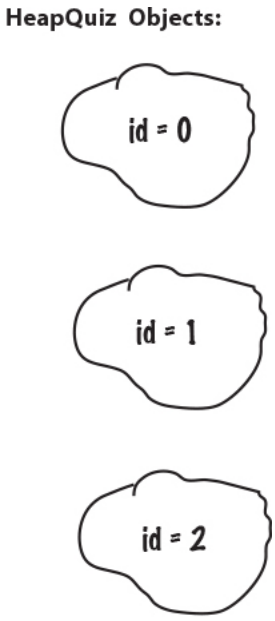
A short Java program is listed to the right. When '// do stuff' is reached, some objects and some reference variables will have been created. Your task is to determine which of the reference variables refer to which objects. Not all the reference variables will be used, and some objects might be referred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

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

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

match each reference variable with matching object(s)

You might not have to use every reference.



The case of the pilfered references

It was a dark and stormy night. Tawny strolled into the programmers' bullpen like she owned the place. She knew that all the programmers would still be hard at work, and she wanted help. She needed a new method added to the pivotal class that was to be loaded into the client's new top-secret Java-enabled cell phone. Heap space in the cell phone's memory was tight, and everyone knew it. The normally raucous buzz in the bullpen fell to silence as Tawny eased her way to the white board. She sketched a quick overview of the new method's functionality and slowly scanned the room. "Well boys, it's crunch time", she purred. "Whoever creates the most memory efficient version of this method is coming with me to the client's launch party on Maui tomorrow... to help me install the new software."

The next morning Tawny glided into the bullpen. "Gentlemen", she smiled, "the plane leaves in a few hours, show me what you've got!". Bob went first; as he began to sketch his design on the white board Tawny said, "Let's get to the point Bob, show me how you handled updating the list of contact objects." Bob quickly drew a code fragment on the board:

Five-Minute Mystery

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

"Tawny, I know we're tight on memory, but your spec said that we had to be able to access individual contact information for all ten allowable contacts, this was the best scheme I could cook up", said Bob. Kent was next, already imagining coconut cocktails with Tawny, "Bob," he said, "your solution's a bit kludgy don't you think?" Kent smirked, "Take a look at this baby":

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

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

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

Exercise Solutions

Code Magnets:

```
class TestArrays {
  public static void main(String [] args) {
    int [] index = new int[4];
    index[0] = 1;
    index[1] = 3;
    index[2] = 0;
    index[3] = 2;
    String [] islands = new String[4];
    islands[0] = "Bermuda";
    islands[1] = "Fiji";
    islands[2] = "Azores";
    islands[3] = "Cozumel";
    int y = 0;
    int ref;
    while (y < 4) {
      ref = index[y];
      System.out.print("island = ");
      System.out.println(islands[ref]);
      y = y + 1;
    }
  }
}
```

```
File Edit Window Help Bikini

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

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

Remember: We have to actually make the Books objects !

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

Remember: arrays start with element 0 !

B

Puzzle Solutions

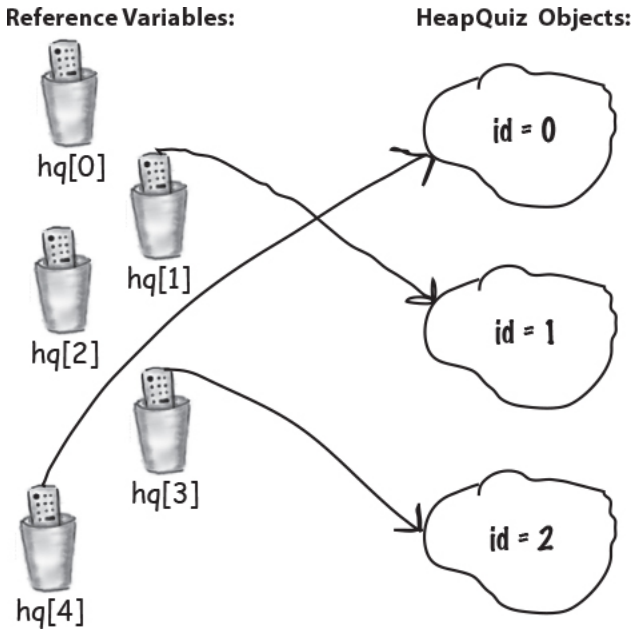
```
class Triangle {
  double area;
  int height;
  int length;
  public static void main(String [] args) {
    int x = 0;
    Triangle [ ] ta = new Triangle[4];
    while ( x < 4 ) {
      ta[x] = new Triangle();
      ta[x].height = (x + 1) * 2;
      ta[x].length = x + 4;
      ta[x].setArea();
      System.out.print("triangle "+x+", area");
      System.out.println(" = " + ta[x].area);
      x = x + 1;
    }
    int y = x;
    x = 27;
    Triangle t5 = ta[2];
    ta[2].area = 343;
    System.out.print("y = " + y);
    System.out.println(", t5 area = "+ t5.area);
  }
  void setArea() {
    area = (height * length) / 2;
  }
}
```

```
File  Edit  Window  Help  Bermuda
%java Triangle
triangle 0, area = 4.0
triangle 1, area = 10.0
triangle 2, area = 18.0
triangle 3, area = 28.0
y = 4, t5 area = 343.0
```

The case of the pilfered references

Tawny could see that Kent's method had a serious flaw. It's true that he didn't use as many reference variables as Bob, but there was no way to access any but the last of the Contact objects that his method created. With each trip through the loop, he was assigning a new object to the one reference variable, so the previously referenced object was abandoned on the heap – *unreachable*. Without access to nine of the ten objects created, Kent's method was useless.

(The software was a huge success and the client gave Tawny and Bob an extra week in Hawaii. We'd like to tell you that by finishing this book you too will get stuff like that.)



Chapter 4. methods use instance variables: How Objects Behave

State affects behavior, behavior affects state. We know that objects have **state** and **behavior**, represented by **instance variables** and **methods**. But until now, we haven't looked at how state and behavior are *related*. We already know that each instance of a class (each object of a particular type) can have its own unique values for its instance variables. Dog A can have a

name “Fido” and a *weight* of 70 pounds. Dog B is “Killer” and weighs 9 pounds. And if the Dog class has a method makeNoise(), well, don’t you think a 70-pound dog barks a bit deeper than the little 9-pounder? (Assuming that annoying yippy sound can be considered a *bark*.) Fortunately, that’s the whole point of an object—it has *behavior* that acts on its *state*. In other words, ***methods use instance variable values***. Like, “if dog is less than 14 pounds, make yippy sound, else...” or “increase weight by 5”. ***Let’s go change some state.***

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

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

Can every object of that type have different method behavior?

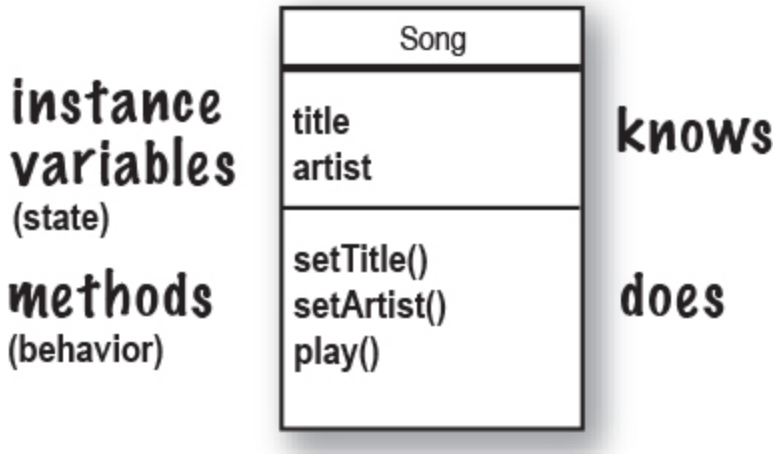
Well... ***sort of.****

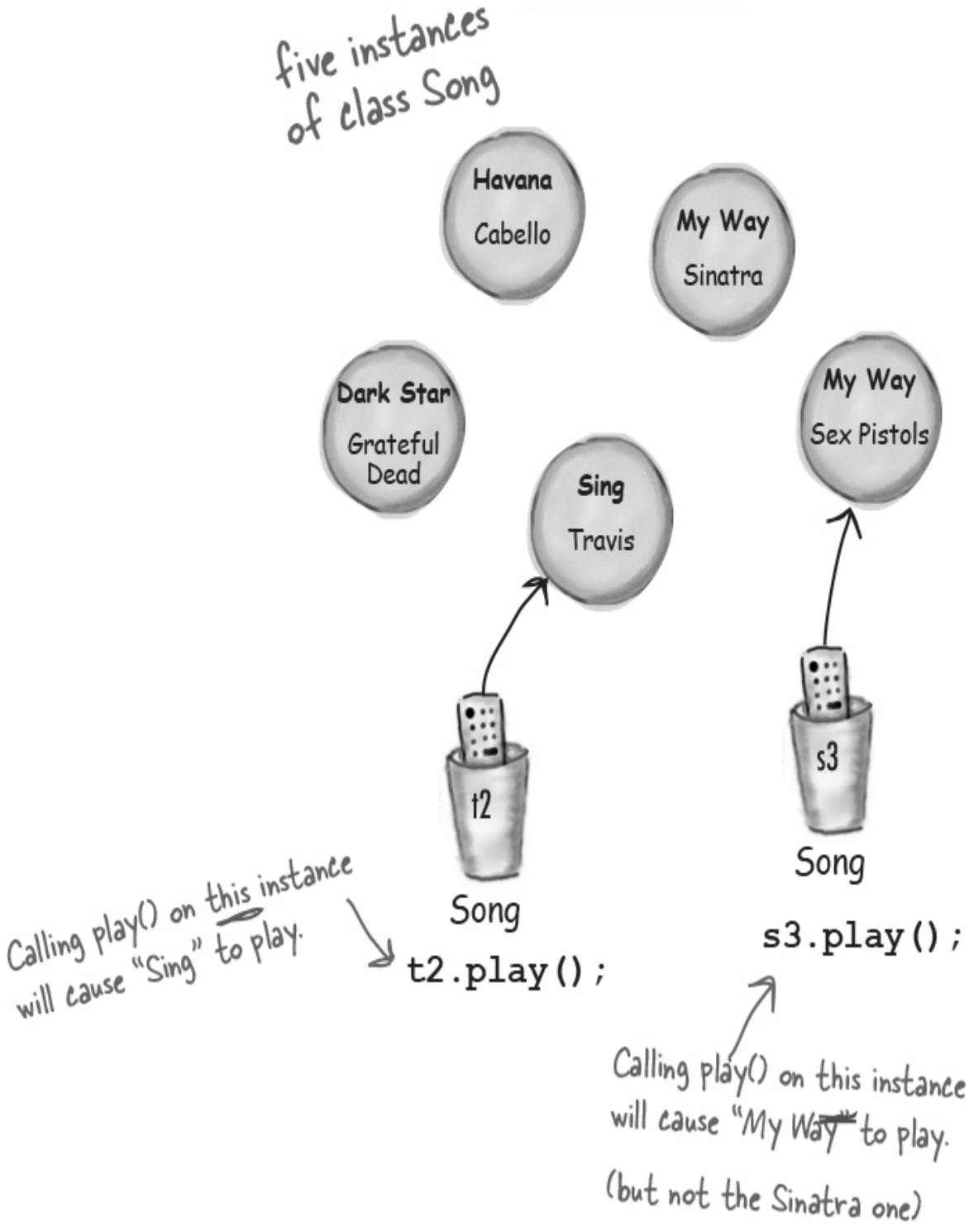
Every instance of a particular class has the same methods, but the methods can *behave* differently based on the value of the instance variables.

The Song class has two instance variables, *title* and *artist*. The play() method plays a song, but the instance you call play() on will play the song represented by the value of the *title* instance variable for that instance. So, if you call the play() method on one instance you’ll hear the song “Havana”, while another instance plays “Blinding Lights”. The method code, however, is the same.

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

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



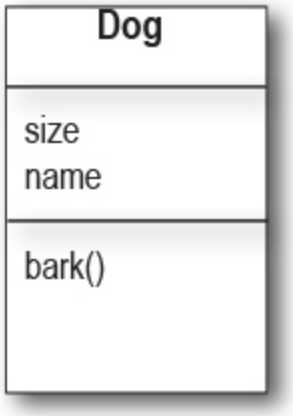


*Yes, another stunningly clear answer!

The size affects the bark

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

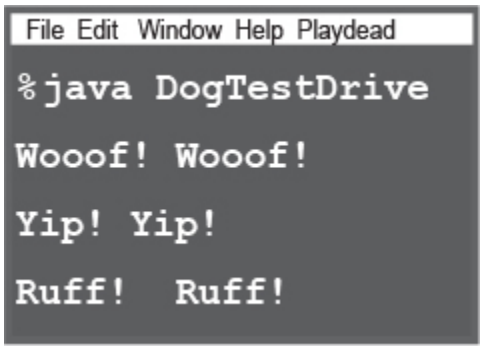
The Dog class has an instance variable *size*, that the *bark()* method uses to decide what kind of bark sound to make.

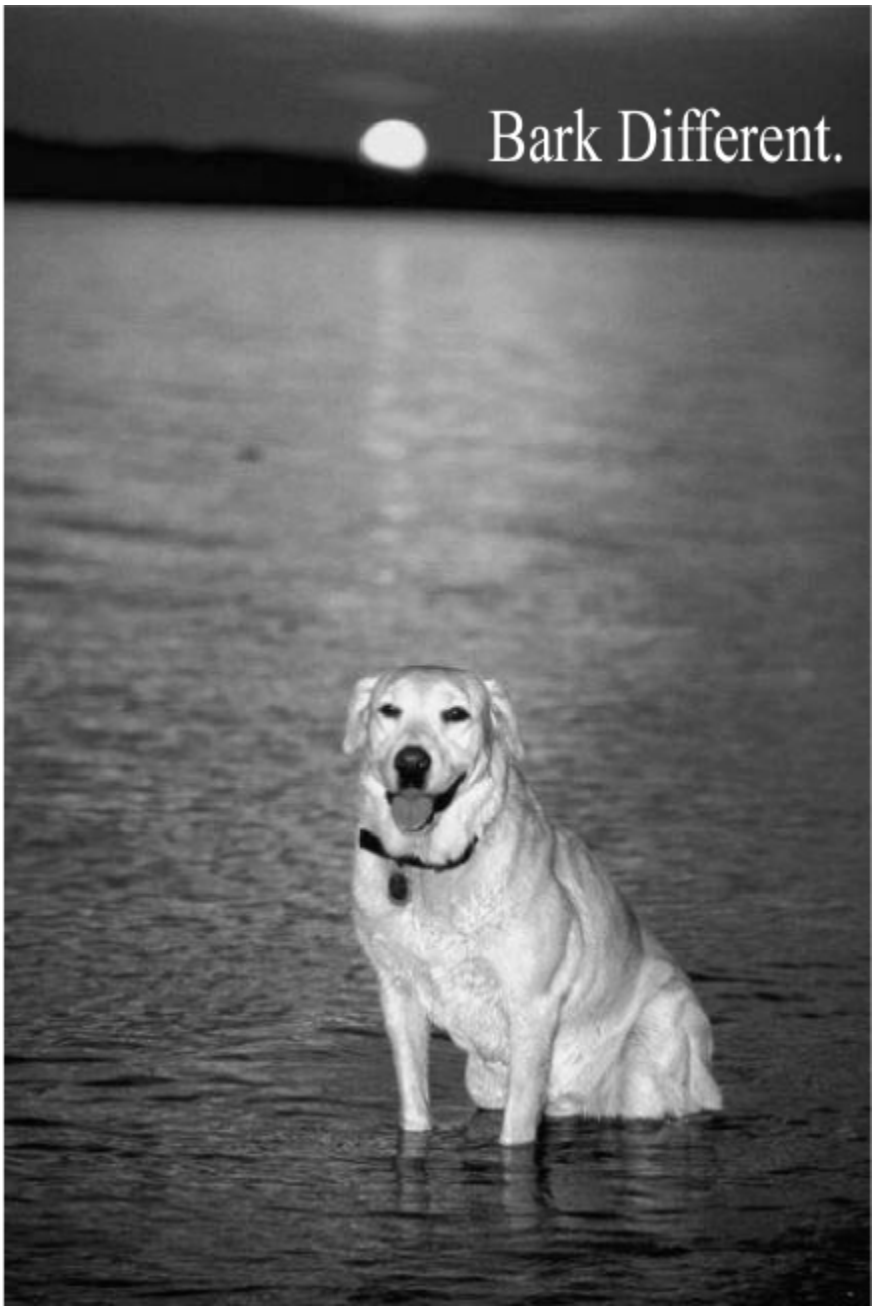



```
class Dog {
  int size;
  String name;


  void bark() {
    if (size > 60) {
       System.out.println("Wooof! Wooof!");
    } else if (size > 14) {
       System.out.println("Ruff! Ruff!");
    } else {
       System.out.println("Yip! Yip!");
    }
  }
}
```

```
class DogTestDrive {

   public static void main (String[] args) {
     Dog one = new Dog();
     one.size = 70;
     Dog two = new Dog();
     two.size = 8;
```

```
      Dog three = new Dog();
      three.size = 35;

      one.bark();
      two.bark();
      three.bark();
    }
}
```





You can send things to a method

Just as you expect from any programming language, you can pass values into your methods. You might, for example, want to tell a Dog object how many times to bark by calling:

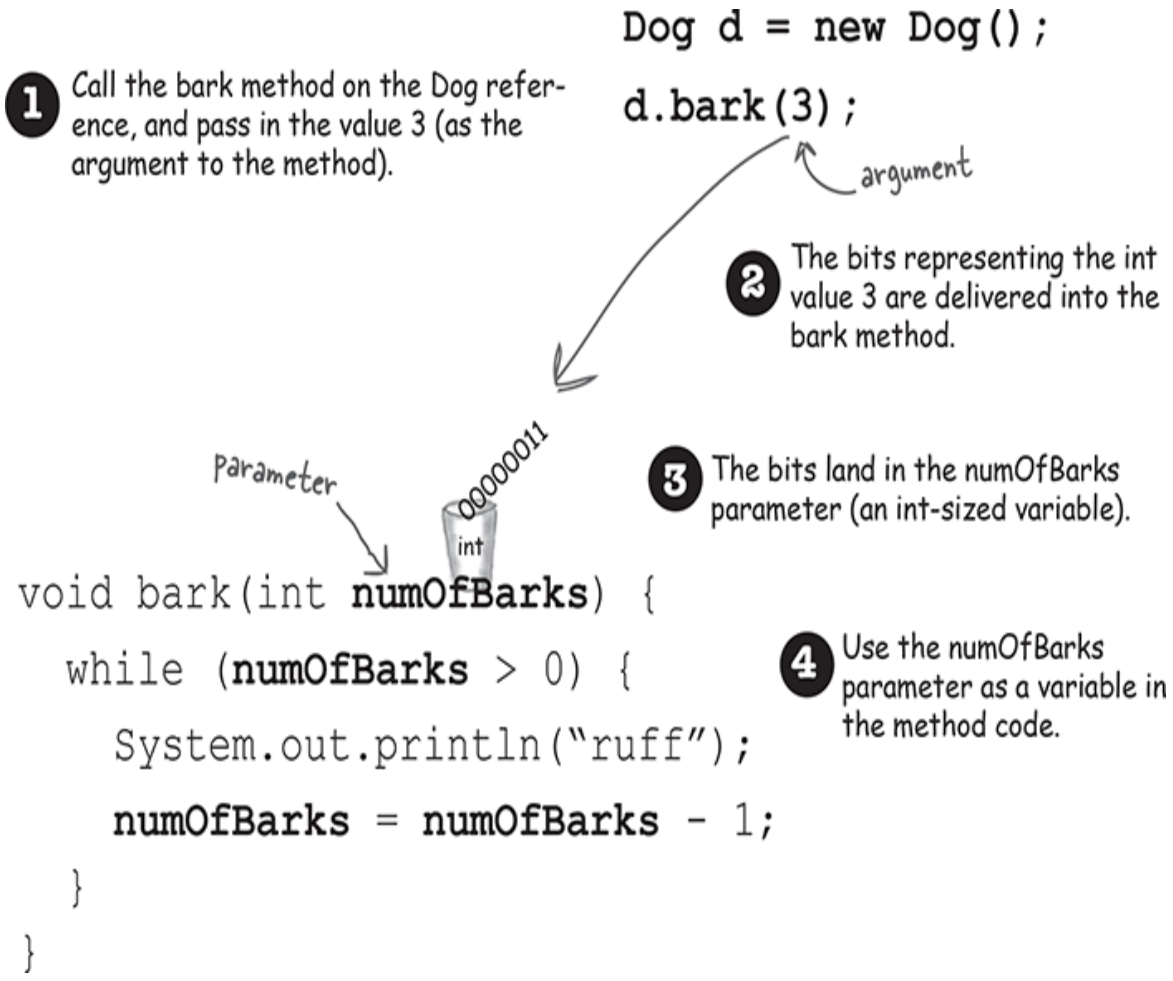
```
d.bark(3);
```

Depending on your programming background and personal preferences, *you* might use the term *arguments* or perhaps *parameters* for the values passed into a method. Although there *are* formal computer science distinctions that people who wear lab coats and who will almost certainly not read this book, make, we have bigger fish to fry in this book. So *you* can call them whatever you like (arguments, donuts, hair-balls, etc.) but we're doing it like this:

A method uses parameters. A caller passes arguments.

Arguments are the things you pass into the methods. An ***argument*** (a value like 2, "Foo", or a reference to a Dog) lands face-down into a... wait for it... ***parameter***. And a parameter is nothing more than a local variable. A variable with a type and a name, that can be used inside the body of the method.

But here's the important part: **If a method takes a parameter, you *must* pass it something.** And that something must be a value of the appropriate type.



You can get things *back* from a method.

Methods can return values. Every method is declared with a return type, but until now we've made all of our methods with a **void** return type, which means they don't give anything back.

```
void go() {
}
```

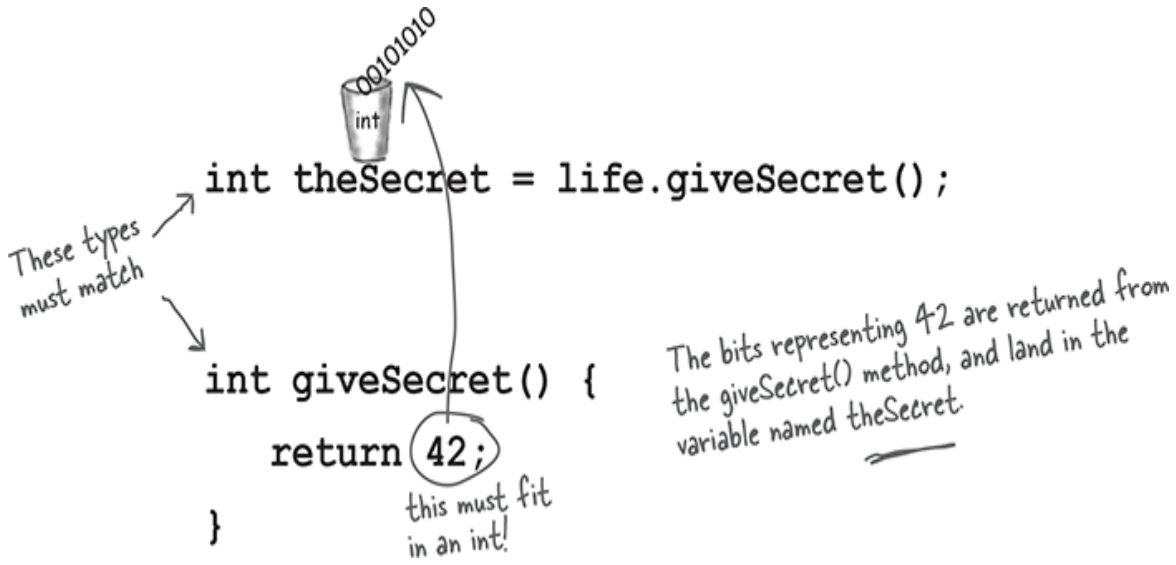
But we can declare a method to give a specific type of value back to the caller, such as:

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

If you declare a method to return a value, you *must* return a value of the declared type! (Or a value that is *compatible* with the declared type. We'll get into that more when we talk about polymorphism in chapter 7 and chapter 8.)

Whatever you say you'll give back, you better give back!

The compiler won't let you return the wrong type of thing.



You can send more than one thing to a method

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

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

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

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

The arguments you pass land in the same order you passed them. First argument lands in the first parameter, second argument in the second parameter, and so on.

You can pass variables into a method, as long as the variable type matches the parameter type.

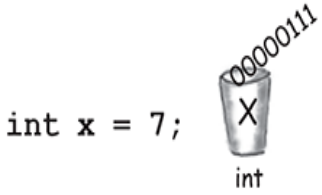
```
void go() {
   int foo = 7;
   int bar = 3;
   t.takeTwo(foo, bar);
}

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

The values of foo and bar land in the x and y parameters. So now the bits in x are identical to the bits in foo (the bit pattern for the integer '7') and the bits in y are identical to the bits in bar.

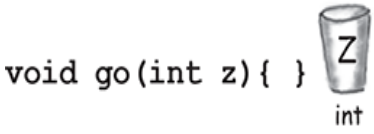
What's the value of z? It's the same result you'd get if you added foo + bar at the time you passed them into the takeTwo method

Java is pass-by-value.

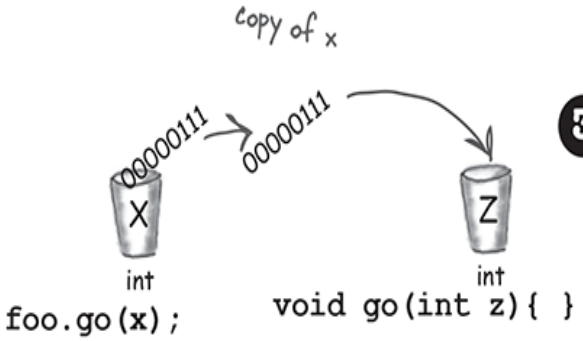
That means pass-by-copy.



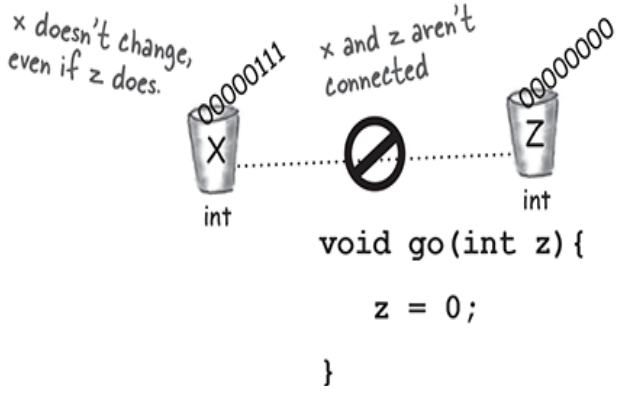
1 Declare an int variable and assign it the value '7'. The bit pattern for 7 goes into the variable named x.



2 Declare a method with an int parameter named z.



3 Call the go() method, passing the variable x as the argument. The bits in x are copied, and the copy lands in z.



4 Change the value of z inside the method. The value of x doesn't change! The argument passed to the z parameter was only a copy of x.

The method can't change the bits that were in the calling variable x.

There are no Dumb Questions

Q: What happens if the argument you want to pass is an object instead of a primitive?

A: You'll learn more about this in later chapters, but you already *know* the answer. Java passes *everything* by value. ***Everything***. But... *value* means *bits inside the variable*. And remember, you don't stuff objects into variables; the variable is a remote control—*a reference to an object*. So if you pass a reference to an object into a method, you're passing a *copy of the remote control*. Stay tuned, though, we'll have lots more to say about this.

Q: Can a method declare multiple return values? Or is there some way to return more than one value?

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

Q: Do I have to return the exact type I declared?

A: You can return anything that can be *implicitly* promoted to that type. So, you can pass a byte where an int is expected. The caller won't care, because the byte fits just fine into the int the caller will use for assigning the result. You must use an *explicit* cast when the declared type is *smaller* than what you're trying to return.

Q: Do I have to do something with the return value of a method? Can I just ignore it?

A: Java doesn't require you to acknowledge a return value. You might want to call a method with a non-void return type, even though you don't care about the return value. In this case, you're calling the method for the work it does *inside* the method, rather than for what the method gives *returns*. In Java, you don't have to assign or use the return value.

Reminder: Java cares about type!

You can’t return a Giraffe when the return type is declared as a Rabbit. Same thing with parameters. You can’t pass a Giraffe into a method that takes a Rabbit.

BULLET POINTS 

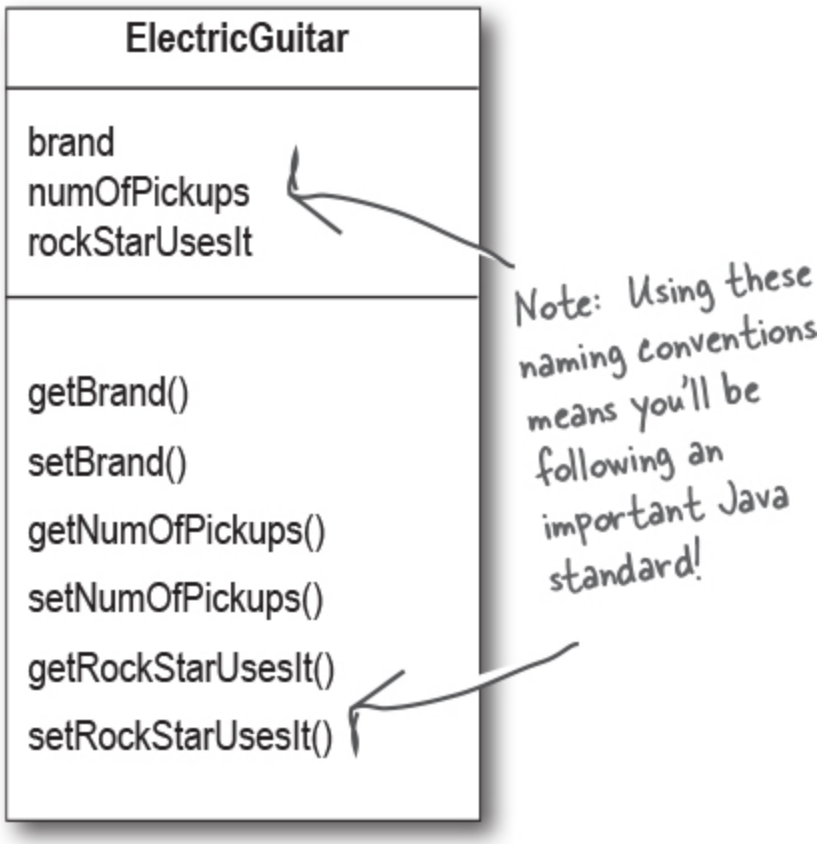
- Classes define what an object knows and what an object does.
- Things an object knows are its **instance variables** (state).
- Things an object does are its **methods** (behavior).
- Methods can use instance variables so that objects of the same type can behave differently.
- A method can have parameters, which means you can pass one or more values in to the method.
- The number and type of values you pass in must match the order and type of the parameters declared by the method.
- Values passed in and out of methods can be implicitly promoted to a larger type or explicitly cast to a smaller type.
- The value you pass as an argument to a method can be a literal value (2, ‘c’, etc.) or a variable of the declared parameter type (for example, *x* where *x* is an int variable). (There are other things you can pass as arguments, but we’re not there yet.)
- A method *must* declare a return type. A void return type means the method doesn’t return anything.
- If a method declares a non-void return type, it *must* return a value compatible with the declared return type.

Cool things you can do with parameters and return types

Now that we’ve seen how parameters and return types work, it’s time to put them to good use: **Getters** and **Setters**. If you’re into being all formal about

it, you might prefer to call them *Accessors* and *Mutators*. But that's a waste of perfectly good syllables. Besides, Getters and Setters fits the Java naming convention, so that's what we'll call them.

Getters and Setters let you, well, *get and set things*. Instance variable values, usually. A Getter's sole purpose in life is to send back, as a return value, the value of whatever it is that particular Getter is supposed to be Getting. And by now, it's probably no surprise that a Setter lives and breathes for the chance to take an argument value and use it to *set* the value of an instance variable.




```
class ElectricGuitar {

   String brand;
   int numOfPickups;
   boolean rockStarUsesIt;

   String getBrand() {
```

```
        return brand;
    }

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

    int getNumOfPickups() {
        return numOfPickups;
    }

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

    boolean getRockStarUsesIt() {
        return rockStarUsesIt;
    }

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

Encapsulation

Do it or risk humiliation and ridicule.

Until this most important moment, we've been committing one of the worst OO faux pas (and we're not talking minor violation like showing up without the 'B' in BYOB). No, we're talking Faux Pas with a capital 'F'. And 'P'.

Our shameful transgression?

Exposing our data!

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

You may have already experienced that vaguely unsettling feeling that comes with leaving your instance variables exposed.

Exposed means reachable with the dot operator, as in:

```
theCat.height = 27;
```

Think about this idea of using our remote control to make a direct change to the Cat object's size instance variable. In the hands of the wrong person, a reference variable (remote control) is quite a dangerous weapon. Because what's to prevent:

```
theCat.height = 0;
```

yikes! We can't let this happen!

This would be a Bad Thing. We need to build setter methods for all the instance variables, and find a way to force other code to call the setters

rather than access the data directly.

> **NOTE**
>
> By forcing everybody to call a setter method, we can protect the cat from unacceptable size changes.

```
public void setHeight(int ht) {
    if (ht > 9) {
        height = ht;
    }
}
```

We put in checks to guarantee a minimum cat height

Hide the data

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

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

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

> **NOTE**
>
> Mark instance variables **private**.
>
> Mark getters and setters **public**.

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

(overheard at the water cooler).

Java Exposed

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

HeadFirst: What’s the big deal about encapsulation?

Object: OK, you know that dream where you’re giving a talk to 500 people when you suddenly realize– you’re *naked*?

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

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

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

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

HeadFirst: Can I get you anything? Water?

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

HeadFirst: So what does encapsulation protect you from?

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

HeadFirst: Can you give me an example?

Object: Doesn’t take a PhD here. Most instance variable values are coded with certain assumptions about the boundaries of the values. Like, think of

all the things that would break if negative numbers were allowed. Number of bathrooms in an office. Velocity of an airplane. Birthdays. Barbell weight. Cell phone numbers. Microwave oven power.

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

Object: By forcing other code to go through setter methods. That way, the setter method can validate the parameter and decide if it's do-able. Maybe the method will reject it and do nothing, or maybe it'll throw an Exception (like if it's a null social security number for a credit card application), or maybe the method will round the parameter sent in to the nearest acceptable value. The point is, you can do whatever you want in the setter method, whereas you can't do *anything* if your instance variables are public.

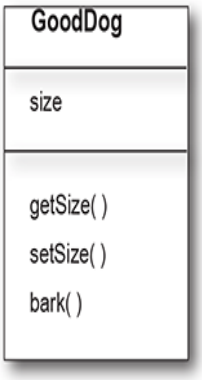
HeadFirst: But sometimes I see setter methods that simply set the value without checking anything. If you have an instance variable that doesn't have a boundary, doesn't that setter method create unnecessary overhead? A performance hit?

Object: The point to setters (and getters, too) is that ***you can change your mind later, without breaking anybody else's code!*** Imagine if half the people in your company used your class with public instance variables, and one day you suddenly realized, "Oops– there's something I didn't plan for with that value, I'm going to have to switch to a setter method." You break everyone's code. The cool thing about encapsulation is that *you get to change your mind.* And nobody gets hurt. The performance gain from using variables directly is so miniscule and would rarely—*if ever*— be worth it.

Encapsulating the GoodDog class

Make the instance variable private.

Make the getter and setter methods public.



Even though the methods don't really add new functionality, the cool thing is that you can change your mind later. You can come back and make a method safer, faster, better.

```
class GoodDog {

  private int size;

  public int getSize() {
    return size;
  }

  public void setSize(int s) {
    size = s;
  }

  void bark() {
    if (size > 60) {
      System.out.println("Wooof! Wooof!");
    } else if (size > 14) {
      System.out.println("Ruff!  Ruff!");
    } else {
      System.out.println("Yip! Yip!");
    }
  }
}

class GoodDogTestDrive {

  public static void main (String[] args) {
    GoodDog one = new GoodDog();
    one.setSize(70);
    GoodDog two = new GoodDog();
    two.setSize(8);
    System.out.println("Dog one: " + one.getSize());
    System.out.println("Dog two: " + two.getSize());
    one.bark();
    two.bark();
  }
}
```

Any place where a particular value can be used, a *method call that returns that type can be used.*

instead of:

```
int x = 3 + 24;
```

you can say:

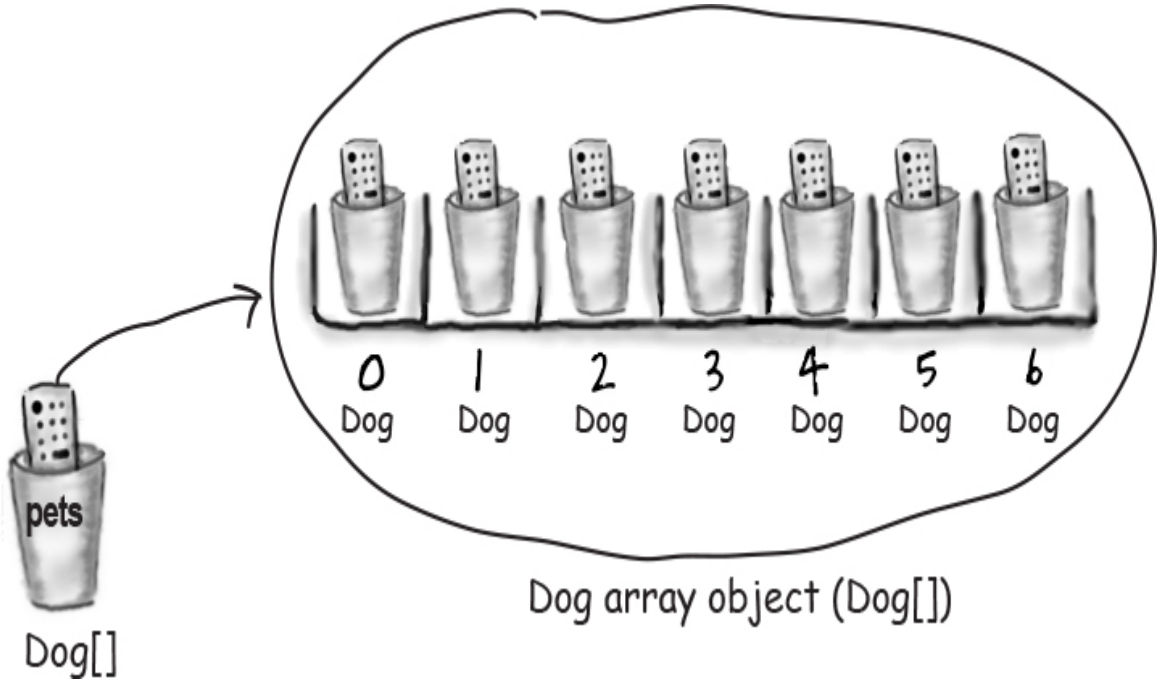
```
int x = 3 + one.getSize();
```

How do objects in an array behave?

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

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

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



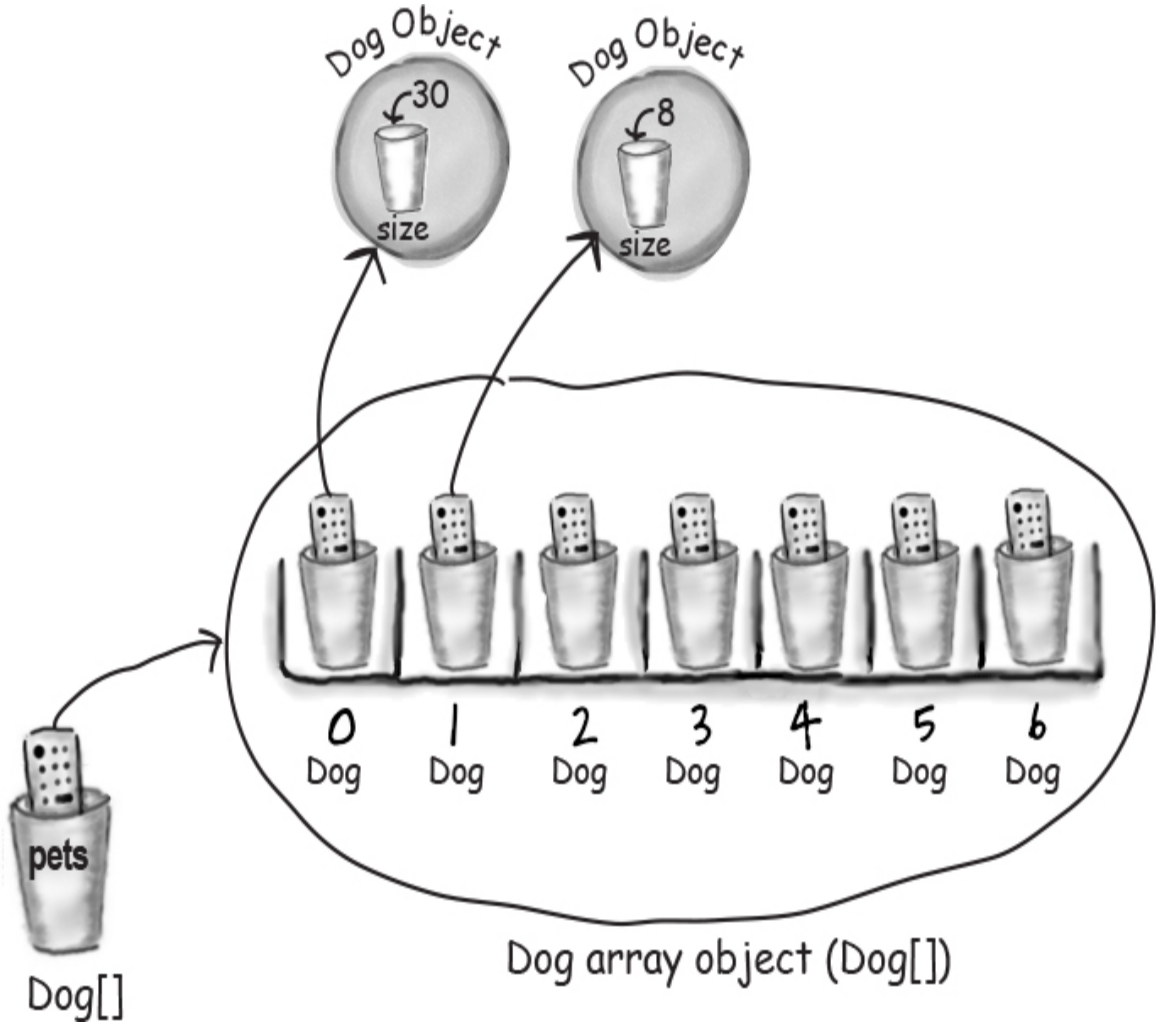
❷ Create two new Dog objects, and assign them to the first two array elements.

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

❸ Call methods on the two Dog objects.

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

```
pets[1].setSize(8);
```



Declaring and initializing instance variables

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

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

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

```
int size = 420;
String name = “Donny”;
```

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

```
class PoorDog {

   private int size;
   private String name;

   public int getSize() {
      return size;
   }
   public String getName() {
      return name;
   }

}

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

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

What will these return??

What do you think? Will this even compile?

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

You don't have to initialize instance variables, because they always have a default value. Number primitives (including char) get 0, booleans get false, and object reference variables get null.

(Remember, null just means a remote control that isn't controlling / programmed to anything. A reference, but no actual object.)

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

integers	0
floating points	0.0
booleans	false
references	null

The difference between instance and local variables

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

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

> **NOTE**
>
> **Local variables do NOT get a default value! The compiler complains if you try to use a local variable before the variable is initialized.**

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

```
class AddThing {
   int a;
   int b = 12;
```

```
    public int add() {
       int total = a + b;
       return total;
    }
}
```

❸ **Local** variables MUST be initialized before use!

```
class Foo {
   public void go() {
      int x;
      int z = x + 3;
   }
}
```

Won't compile!! You can declare x without a value, but as soon as you try to USE it, the compiler freaks out.

```
File Edit Window Help Yikes
% javac Foo.java
Foo.java:4: variable x might
not have been initialized
      int z = x + 3;
1 error       ^
```

There are no Dumb Questions

Q: What about method parameters? How do the rules about local variables apply to them?

A: Method parameters are virtually the same as local variables—they're declared *inside* the method (well, technically they're declared in the *argument list* of the method rather than within the *body* of the method, but they're still local variables as opposed to instance variables). But method parameters will never be uninitialized, so you'll never get a compiler error telling you that a parameter variable might not have been initialized.

But that's because the compiler will give you an error if you try to invoke a method without sending arguments that the method needs. So parameters are ALWAYS initialized, because the compiler guarantees that methods are always called with arguments that match the parameters declared for the method, and the arguments are assigned (automatically) to the parameters.

Comparing variables (primitives or references)

Sometimes you want to know if two *primitives* are the same. That's easy enough, just use the == operator. Sometimes you want to know if two reference variables refer to a single object on the heap. Easy as well, just use the == operator. But sometimes you want to know if two *objects* are equal. And for that, you need the .equals() method. The idea of equality for objects depends on the type of object. For example, if two different String objects have the same characters (say, "expeditious"), they are meaningfully equivalent, regardless of whether they are two distinct objects on the heap. But what about a Dog? Do you want to treat two Dogs as being equal if they happen to have the same size and weight? Probably not. So whether two different objects should be treated as equal depends on what makes sense for that particular object type. We'll explore the notion of object equality again in later chapters (and appendix B), but for now, we need to understand that the == operator is used *only* to compare the bits in two variables. *What* those bits represent doesn't matter. The bits are either the same, or they're not.

NOTE

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

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

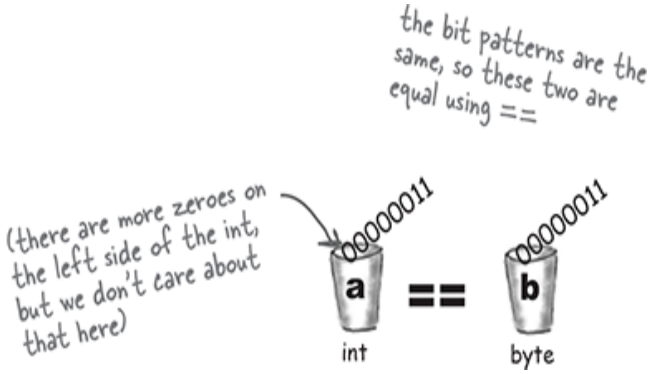
(Such as two different String objects that both represent the characters in "Fred")

To compare two primitives, use the == operator

The == operator can be used to compare two variables of any kind, and it simply compares the bits.

if (a == b) {...} looks at the bits in a and b and returns true if the bit pattern is the same (although it doesn't care about the size of the variable, so all the extra zeroes on the left end don't matter).

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

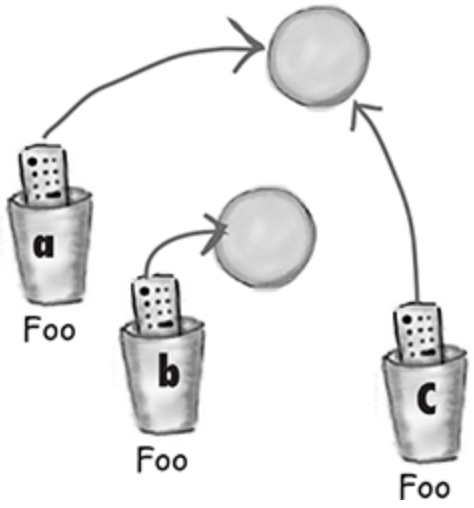


To see if two references are the same (which means they refer to the same object on the heap) use the == operator

Remember, the == operator cares only about the pattern of bits in the variable. The rules are the same whether the variable is a reference or primitive. So the == operator returns true if two reference variables refer to the same object! In that case, we don't know what the bit pattern is (because it's dependent on the JVM, and hidden from us) but we *do* know that whatever it looks like, *it will be the same for two references to a single object*.

```
Foo a = new Foo();
Foo b = new Foo();
Foo c = a;
if (a == b) { // false }
if (a == c) { // true }
if (b == c) { // false }
```

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



a == c is true

a == b is false

Make it Stick

Roses are red,
this poem is choppy,
passing by value
is passing by copy.

Oh, like you can do better? Try it. Replace our dumb second line with your own. Better yet, replace the whole thing with your own words and you'll *never* forget it.

SHARPEN YOUR PENCIL

What's legal?

Given the method below, which of the method calls listed on the right are legal? Put a checkmark next to the ones that are legal. (Some statements are there to assign values used in the method calls).


```
int calcArea(int height, int width) {
   return height * width;
}
int a = calcArea(7, 12);
short c = 7;
calcArea(c,15);
int d = calcArea(57);
calcArea(2,3);
long t = 42;
int f = calcArea(t,17);
int g = calcArea();
calcArea();
byte h = calcArea(4,20);
int j = calcArea(2,3,5);
```

BE the compiler

Each of the Java files on this page represents a complete source file. Your job is to play compiler and determine whether each of these files will compile. If they won't compile, how would you fix them, and if they do compile, what would be their output?

A

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

  int go(int arg) {
    arg = arg * 2;
```

```
    return arg;

  }

}


class Clock {
  String time;

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

  void getTime() {
    return time;
  }
}

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

    Clock c = new Clock();

    c.setTime("1245");
    String tod = c.getTime();
    System.out.println("time: " + tod);
  }
}
```

A bunch of Java components, in full costume, are playing a party game, "Who am I?" They give you a clue, and you try to guess who they are, based on what they say. Assume they always tell the truth about themselves. If they happen to say something that could be true for more than one guy, then write down all for whom that sentence applies. Fill in the blanks next to the sentence with the names of one or more attendees.

Tonight's attendees:

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

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

Mixed Messages

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

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

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

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

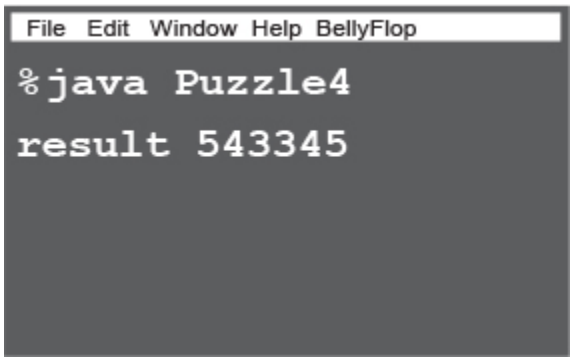
Candidates:

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

Pool Puzzle

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

Output

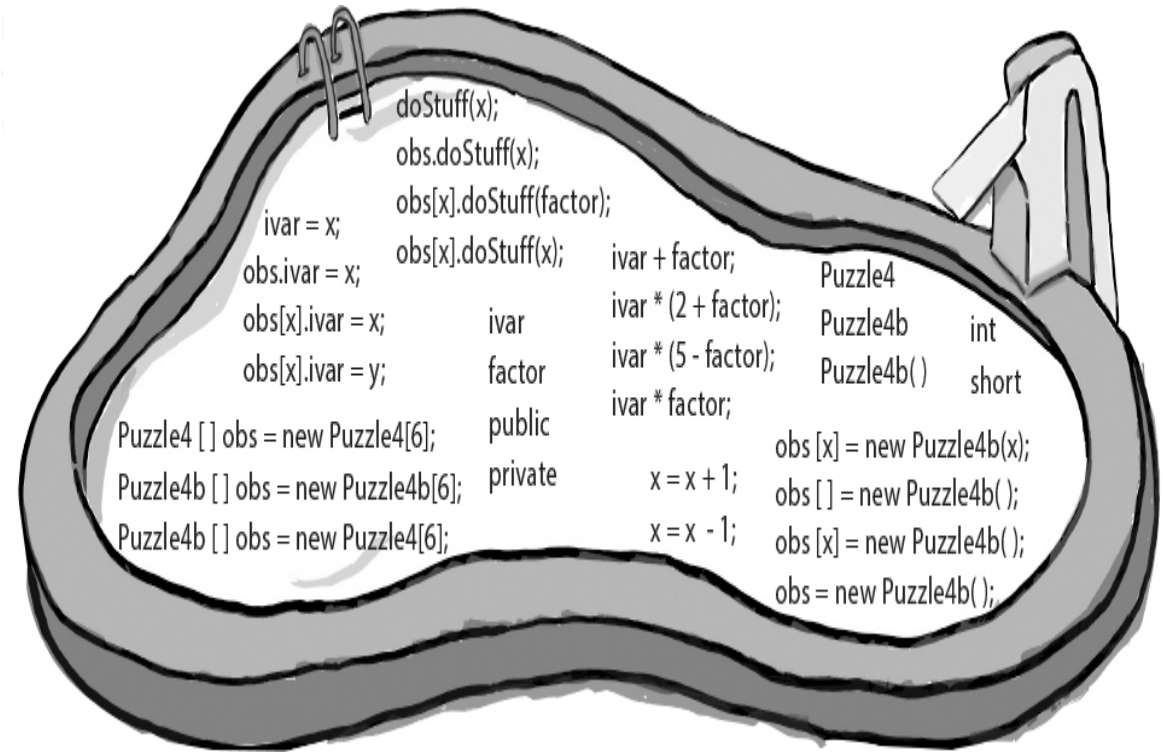



```
public class Puzzle4 {
  public static void main(String [] args) {
    _____________________________________
    int y = 1;
    int x = 0;
    int result = 0;
    while (x < 6) {
      _____________________________
      _____________________________
      y = y * 10;
      ___________________
    }
    x = 6;
    while (x > 0) {
      ___________________
      result = result + ____________________
    }
    System.out.println("result " + result);
```

```
 }
}
class ___________ {
  int ivar;
  ________ ______ doStuff(int _________) {
    if (ivar > 100) {
      return ____________________________
    } else {
      return ____________________________
    }
  }
}
```

Note

Each snippet from the pool can be used only once!



Fast Times in Stim-City

When Buchanan jammed his twitch-gun into Jai's side, Jai froze. Jai knew that Buchanan was as stupid as he was ugly and he didn't want to spook the big guy. Buchanan ordered Jai into his boss's office, but Jai'd done nothing wrong, (lately), so he figured a little chat with Buchanan's boss Leveler couldn't be too bad. He'd been moving lots of neural-stimmers in the west side lately and he figured Leveler would be pleased. Black market stimmers weren't the best money pump around, but they were pretty harmless. Most of the stim-junkies he'd seen tapped out after a while and got back to life, maybe just a little less focused than before.

Five-Minute Mystery

Leveler's 'office' was a skungy looking skimmer, but once Buchanan shoved him in, Jai could see that it'd been modified to provide all the extra speed and armor that a local boss like Leveler could hope for. "Jai my boy", hissed Leveler, "pleasure to see you again". "Likewise I'm sure...", said Jai, sensing the malice behind Leveler's greeting, "We should be square Leveler, have I missed something?" "Ha! You're making it look pretty good Jai, your volume is up, but I've been experiencing, shall we say, a little 'breach' lately..." said Leveler.

Jai winced involuntarily, he'd been a top drawer jack-hacker in his day. Anytime someone figured out how to break a street-jack's security, unwanted attention turned toward Jai. "No way it's me man", said Jai, "not worth the downside. I'm retired from hacking, I just move my stuff and mind my own business". "Yeah, yeah", laughed Leveler, "I'm sure you're clean on this one, but I'll be losing big margins until this new jack-hacker is shut out!" "Well, best of luck Leveler, maybe you could just drop me here and I'll go move a few more 'units' for you before I wrap up today", said Jai.

“I’m afraid it’s not that easy Jai, Buchanan here tells me that word is you’re current on Java NE 37.3.2”, insinuated Leveler. “Neural Edition? sure I play around a bit, so what?”, Jai responded feeling a little queasy. “Neural edition’s how I let the stimjunkies know where the next drop will be”, explained Leveler. “Trouble is, some stim-junkie’s stayed straight long enough to figure out how to hack into my Warehousing database.” “I need a quick thinker like yourself Jai, to take a look at my StimDrop Java NE class; methods, instance variables, the whole enchilada, and figure out how they’re getting in. It should...”, “HEY!”, exclaimed Buchanan, “I don’t want no scum hacker like Jai nosin’ around my code!” “Easy big guy”, Jai saw his chance, “I’m sure you did a top rate job with your access modi... “Don’t tell me - bit twiddler!”, shouted Buchanan, “I left all of those junkie level methods public, so they could access the drop site data, but I marked all the critical Ware-Housing methods private. Nobody on the outside can access those methods buddy, nobody!”

“I think I can spot your leak Leveler, what say we drop Buchanan here off at the corner and take a cruise around the block”, suggested Jai. Buchanan reached for his twitch-gun but Leveler’s stunner was already on Buchanan’s neck, “Let it go Buchanan”, sneered Leveler, “Drop the twitcher and step outside, I think Jai and I have some plans to make”.

What did Jai suspect?

Will he get out of Leveler’s skimmer with all his bones intact?

Exercise Solutions

A Class ‘XCopy’ compiles and runs as it stands ! The output is: ‘42 84’. Remember Java is pass by value, (which means pass by copy), the variable ‘orig’ is not changed by the go( ) method.

```
class Clock {
  String time;
  void setTime(String t) {
    time = t;
B   }
  String getTime() {
    return time;
  }
}

class ClockTestDrive {
  public static void main(String [] args) {
    Clock c = new Clock();
    c.setTime(“1245”);
    String tod = c.getTime();
    System.out.println(“time: “ + tod);
  }
}
```

NOTE

Note: ‘Getter’ methods have a return type by definition.

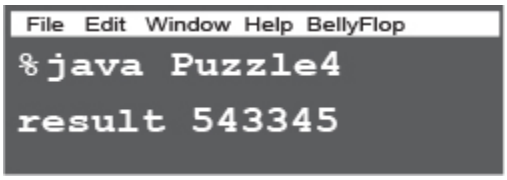
A class can have any number of these.	**instance variables, getter, setter, method**
A method can have only one of these.	**return**
This can be implicitly promoted.	**return, argument**
I prefer my instance variables private.	**encapsulation**
It really means ‘make a copy’.	**pass by value**
Only setters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

Puzzle Solutions

```
public class Puzzle4 {
  public static void main(String [] args) {
    Puzzle4b [ ] obs = new Puzzle4b[6];
    int y = 1;
    int x = 0;
    int result = 0;
    while (x < 6) {
      obs[x] = new Puzzle4b( );
      obs[x] . ivar = y;
      y = y * 10;
      x = x + 1;
    }
    x = 6;
    while (x > 0) {
      x = x - 1;
      result = result + obs[x].doStuff(x);
```

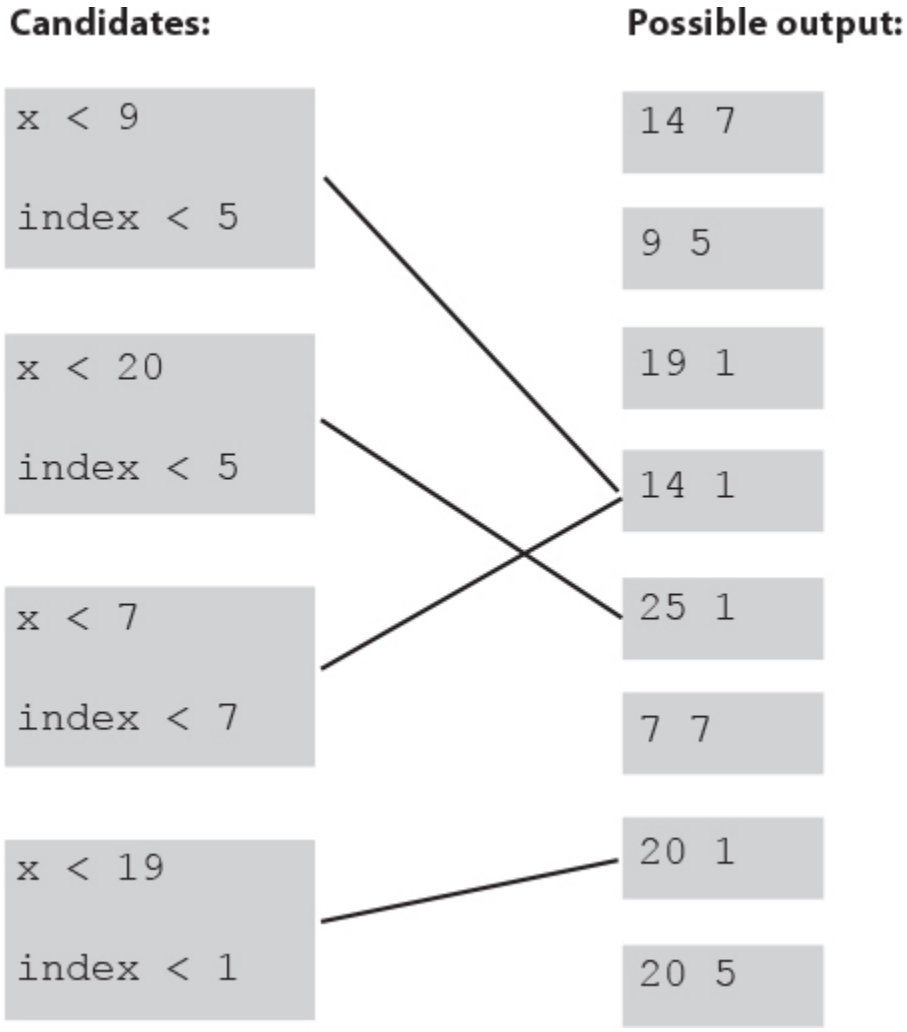
```
    }
    System.out.println("result " + result);
  }
}
class Puzzle4b {
  int ivar;
  public int doStuff(int factor) {
     if (ivar > 100) {
       return ivar * factor;
     } else {
       return ivar * (5 - factor);
     }
   }
}
```

Output



Answer to the 5-minute mystery...

Jai knew that Buchanan wasn't the sharpest pencil in the box. When Jai heard Buchanan talk about his code, Buchanan never mentioned his instance variables. Jai suspected that while Buchanan did in fact handle his methods correctly, he failed to mark his instance variables `private`. That slip up could have easily cost Leveler thousands.


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

Chapter 5. Writing a Program: Extra-Strength Methods

Let's put some muscle in our methods. We dabbled with variables, played with a few objects, and wrote a little code. But we were weak. We need more tools. Like **operators**. We need more operators so we can do something a little more interesting than, say, *bark*. And **loops**. We need

loops, but what's with the wimpy *while* loops? We need ***for*** loops if we're really serious. Might be useful to **generate random numbers**. And **turn a String into an int**, yeah, that would be cool. Better learn that too. And why don't we learn it all by *building* something real, to see what it's like to write (and test) a program from scratch. **Maybe a game**, like Battleships. That's a heavy-lifting task, so it'll take *two* chapters to finish. We'll build a simple version in this chapter, and then build a more powerful deluxe version in chapter 6.

To Do: find 3 short safe, fake statup names.

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

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

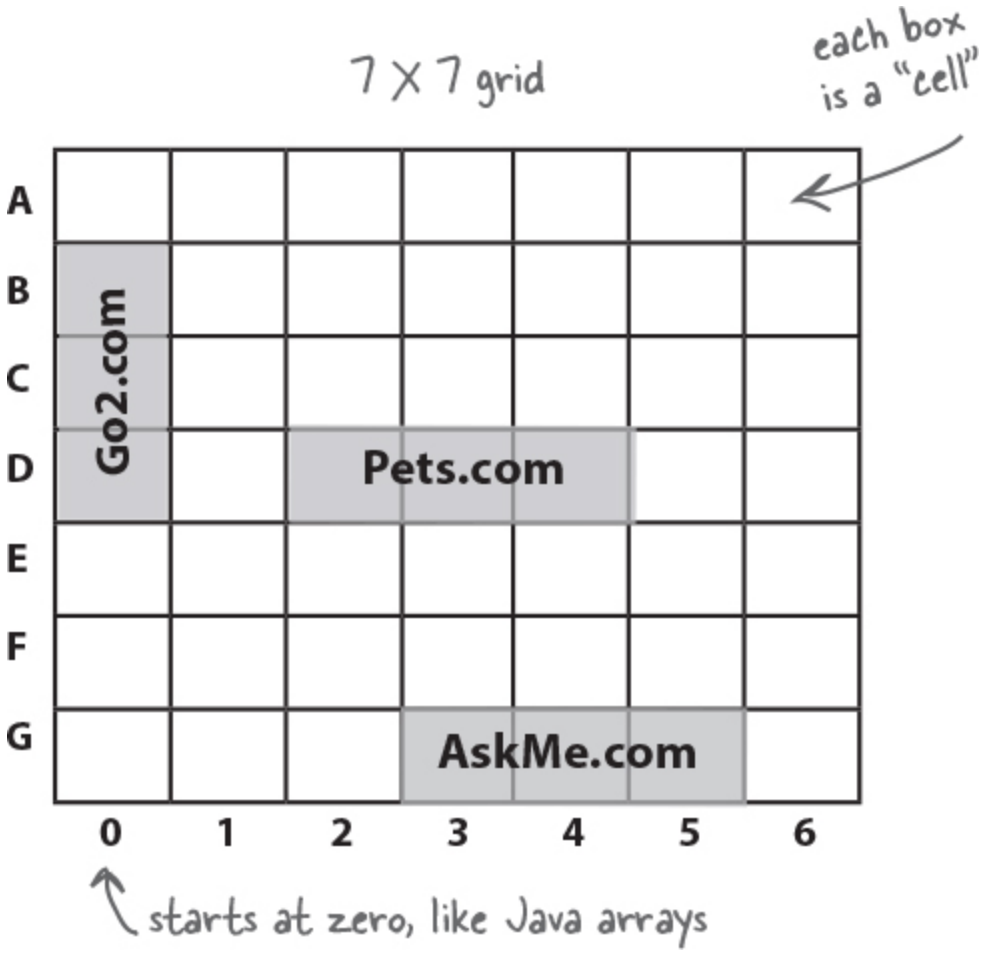
Oh, and we aren't sinking ships. We're killing ill-advised, Silicon Valley Startups. (Thus establishing business relevancy so you can expense the cost of this book).

> **NOTE**
>
> **You're going to build the Sink a Startup game, with a 7 x 7 grid and three Startups. Each Startup takes up three cells.**

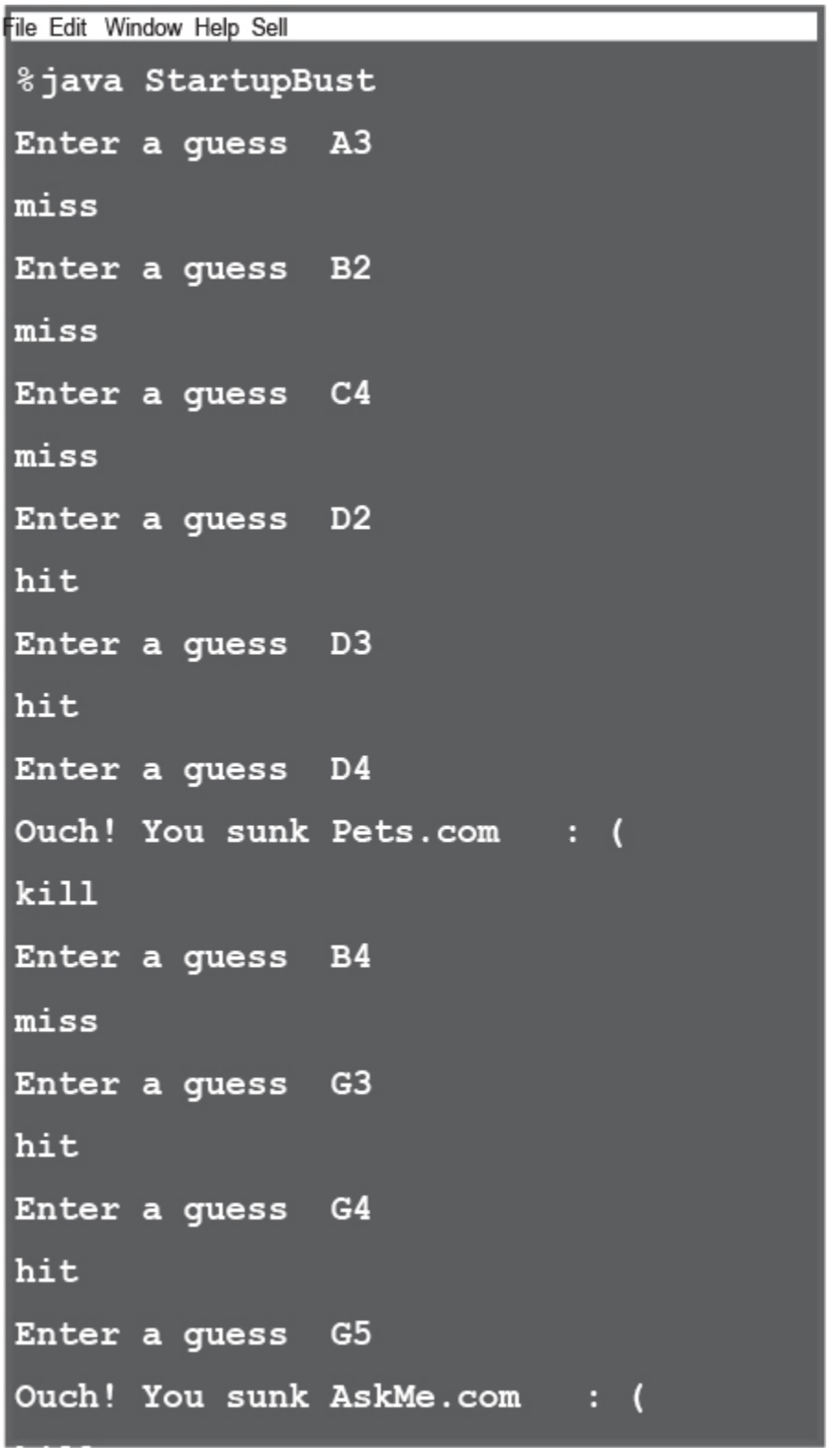
Goal: Sink all of the computer's Startups in the fewest number of guesses. You're given a rating or level, based on how well you perform.

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

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



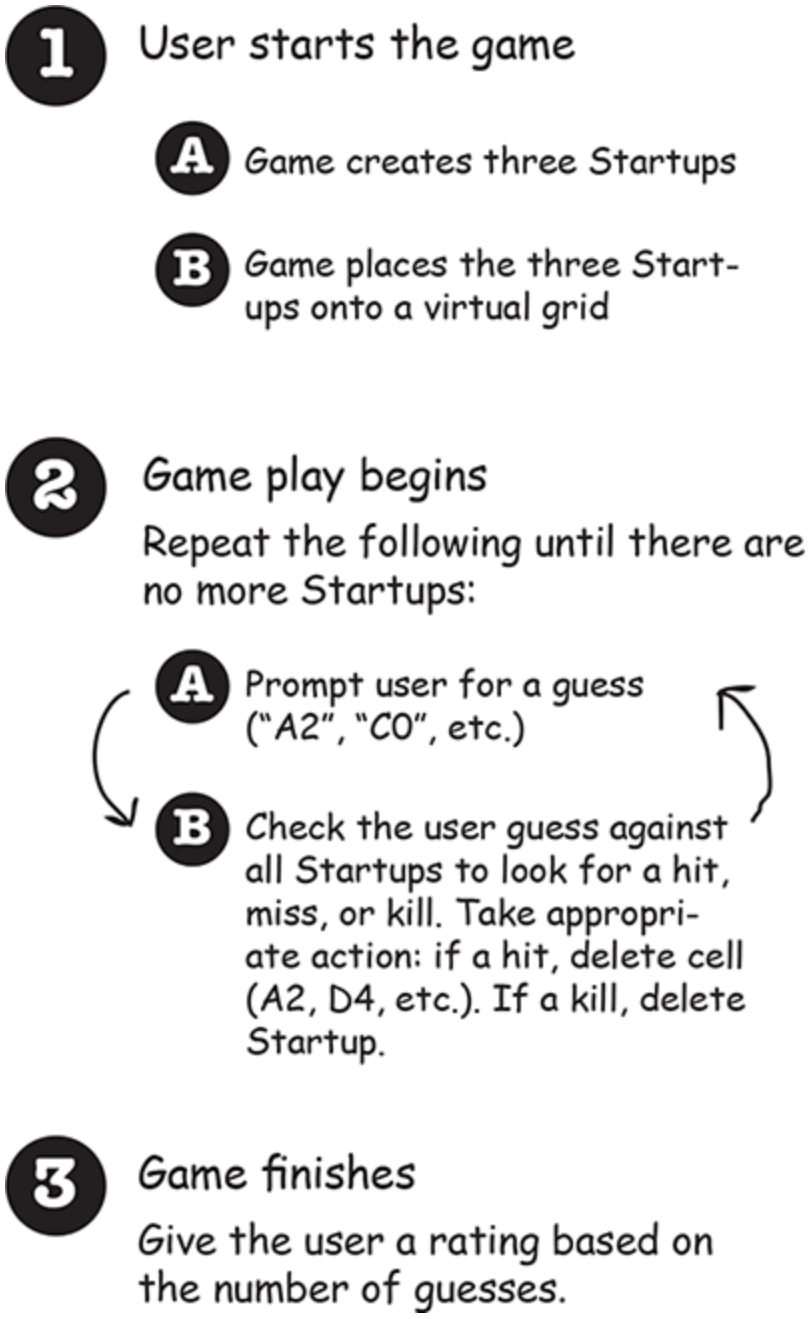
part of a game interaction


File Edit Window Help Sell
%java StartupBust
Enter a guess A3
miss
Enter a guess B2
miss
Enter a guess C4
miss
Enter a guess D2
hit
Enter a guess D3
hit
Enter a guess D4
Ouch! You sunk Pets.com : (
kill
Enter a guess B4
miss
Enter a guess G3
hit
Enter a guess G4
hit
Enter a guess G5
Ouch! You sunk AskMe.com : (

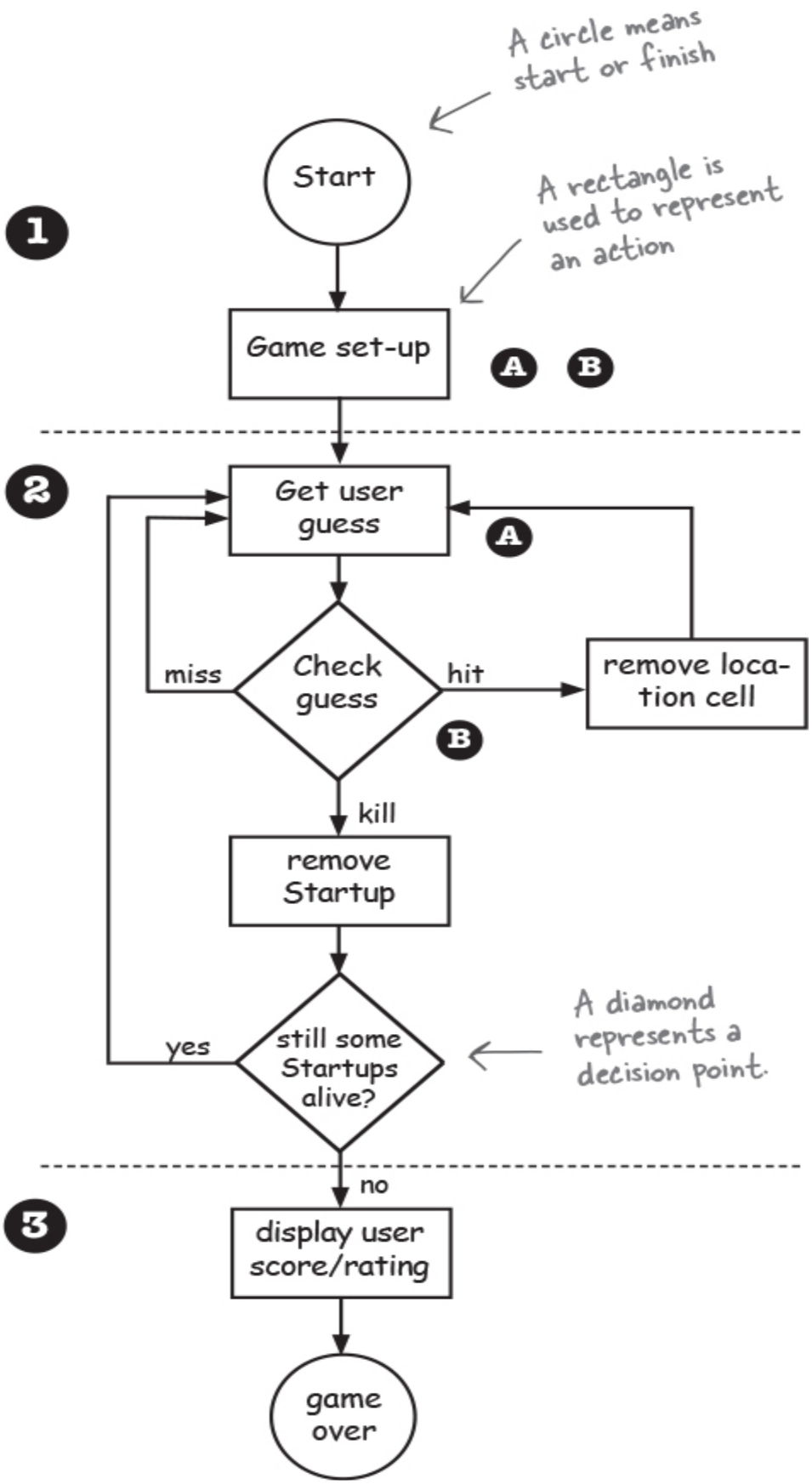
First, a high-level design

We know we'll need classes and methods, but what should they be? To answer that, we need more information about what the game should do.

First, we need to figure out the general flow of the game. Here's the basic idea:



Now we have an idea of the kinds of things the program needs to do. The next step is figuring out what kind of **objects** we'll need to do the work. Remember, think like Brad rather than Larry; focus first on the ***things*** in the program rather than the ***procedures***.


A circle means start or finish
Start
A rectangle is used to represent an action
1
Game set-up
A
B
2
Get user guess
A
miss
Check guess
hit
remove location cell
B
kill
remove Startup
still some Startups alive?
yes
A diamond represents a decision point.
no
3
display user score/rating
game over

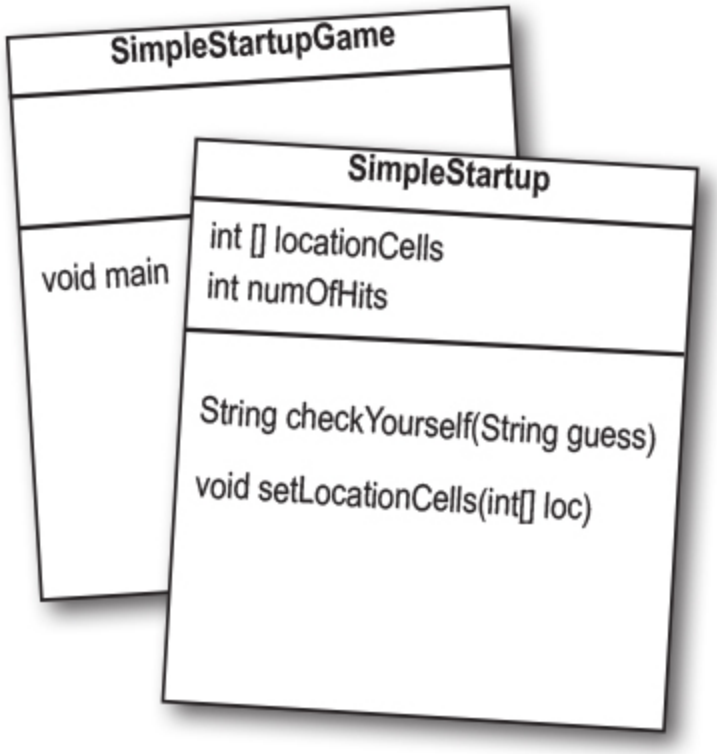
Whoa. A real flow chart.

The “Simple Startup Game” a gentler introduction

It looks like we’re gonna need at least two classes, a Game class and a Startup class. But before we build the full monty ***Sink a Startup*** game, we’ll start with a stripped-down, simplified version, ***Simple Startup Game.*** We’ll build the simple version in *this* chapter, followed by the deluxe version that we build in the *next* chapter.

Everything is simpler in this game. Instead of a 2-D grid, we hide the Startup in just a single *row.* And instead of *three* Startups, we use *one*.

The goal is the same, though, so the game still needs to make a Startup instance, assign it a location somewhere in the row, get user input, and when all of the Startup’s cells have been hit, the game is over. This simplified version of the game gives us a big head start on building the full game. If we can get this small one working, we can scale it up to the more complex one later.



In this simple version, the game class has no instance variables, and all the game code is in the main() method. In other words, when the program is launched and main() begins to run, it will make the one and only Startup instance, pick a location for it (three consecutive cells on the single virtual seven-cell row), ask the user for a guess, check the guess, and repeat until all three cells have been hit.

Keep in mind that the virtual row is... *virtual*. In other words, it doesn't exist anywhere in the program. As long as both the game and the user know that the Startup is hidden in three consecutive cells out of a possible seven (starting at zero), the row itself doesn't have to be represented in code. You might be tempted to build an array of seven ints and then assign the Startup to three of the seven elements in the array, but you don't need to. All we need is an array that holds just the three cells the Startup occupies.

❶ **Game starts**, and creates ONE Startup and gives it a location on three cells in the single row of seven cells.

Instead of "A2", "C4", and so on, the locations are just integers (for example: 1,2,3 are the cell locations in this picture):

❷ **Game play begins**. Prompt user for a guess, then check to see if it hit any of the Startup's three cells. If a hit, increment the numOfHits variable.

❸ **Game finishes** when all three cells have been hit (the numOfHits variable value is 3), and tells the user how many guesses it took to sink the Startup.

A complete game interaction

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

Developing a Class

As a programmer, you probably have a methodology/ process/approach to writing code. Well, so do we. Our sequence is designed to help you see (and learn) what we're thinking as we work through coding a class. It isn't necessarily the way we (or *you)* write code in the Real World. In the Real World, of course, you'll follow the approach your personal preferences,

project, or employer dictate. We, however, can do pretty much whatever we want. And when we create a Java class as a “learning experience”, we usually do it like this:

- ☐ Figure out what the class is supposed to *do*.
- ☐ List the **instance variables and methods**.
- ☐ Write **prepcode** for the methods. (You’ll see this in just a moment.)
- ☐ Write **test code** for the methods.
- ☐ **Implement** the class.
- ☐ **Test** the methods.
- ☐ **Debug** and **reimplement** as needed.
- ☐ Express gratitude that we don’t have to test our so-called *learning experience* app on actual live users.

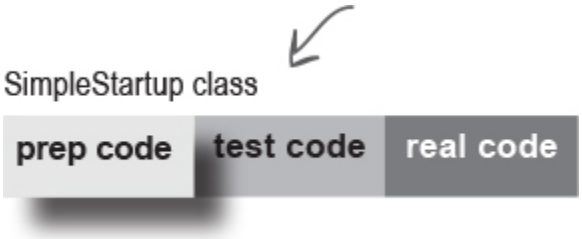
Brain Power

Flex those dendrites.

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

The three things we’ll write for each class:

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



prep code

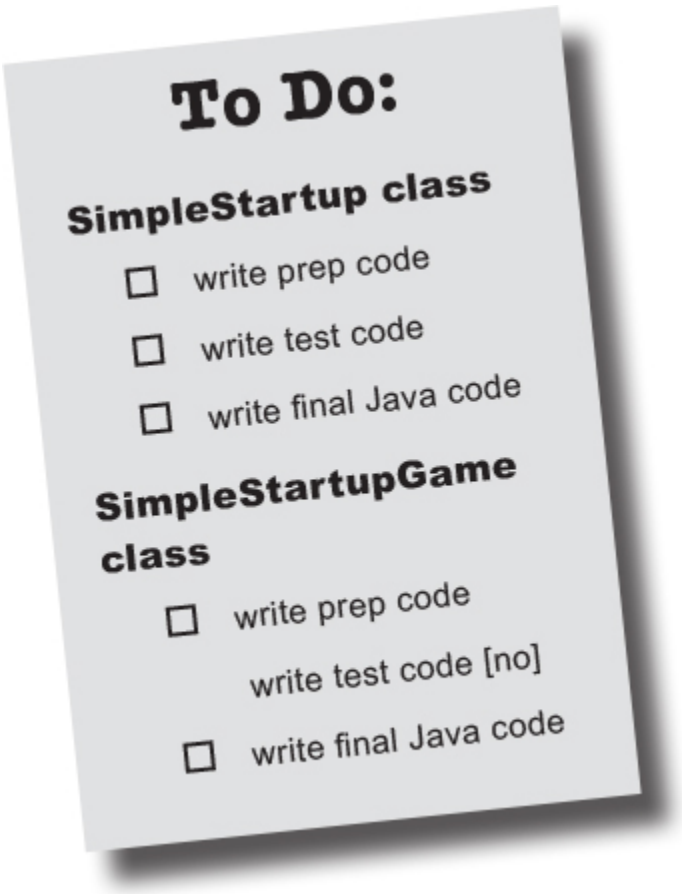
A form of pseudocode, to help you focus on the logic without stressing about syntax.

test code

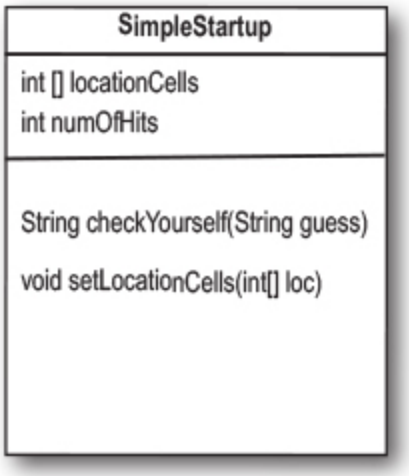
A class or methods that will test the real code and validate that it's doing the right thing.

real code

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



SimpleStartup class



You'll get the idea of how prepcode (our version of pseudocode) works as you read through this example. It's sort of half-way between real Java code and a plain English description of the class. Most prepcode includes three parts: instance variable declarations, method declarations, method logic. The most important part of prepcode is the method logic, because it defines *what* has to happen, which we later translate into *how*, when we actually write the method code.

DECLARE an *int array* to hold the location cells. Call it *locationCells*.

DECLARE an *int* to hold the number of hits. Call it *numOfHits* and **SET** it to 0.

DECLARE a *checkYourself()* method that takes a *String* for the user's guess ("1", "3", etc.), checks it, and returns a result representing a "hit", "miss", or "kill".

DECLARE a *setLocationCells()* setter method that takes an *int array* (which has the three cell locations as *ints* (2,3,4, etc.).

```
METHOD: String checkYourself(String userGuess)
    GET the user guess as a String parameter
    CONVERT the user guess to an int
    REPEAT with each of the location cells in the int array
        // COMPARE the user guess to the location cell
        IF the user guess matches
            INCREMENT the number of hits
            // FIND OUT if it was the last location cell:
            IF number of hits is 3, RETURN "kill" as the result
            ELSE  it was not a kill, so RETURN"hit"
            END IF
        ELSE the user guess did not match, so RETURN "miss"
        END IF
    END REPEAT
END METHOD

METHOD: void setLocationCells(int[] cellLocations)
    GET the cell locations as an int array parameter
    ASSIGN the cell locations parameter to the cell locations instance variable
END METHOD
```

Writing the method implementations

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

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

there’s anything to test!

The concept of writing the test code first is one of the practices of Test-Driven Development (TDD) , and it can make it easier (and faster) for you to write your code. We’re not necessarily saying you should use TDD but we do like the part about writing tests first. And TDDjust *sounds* cool.

TEST-DRIVEN DEVELOPMENT (TDD)

Back in 1999, Extreme Programming(XP) was a newcomer to the software development methodology world. One of the central ideas in XP was to write test code before writing the actual code. Since then, the idea of writing test code first has spun off of XP and become the core of a newer, more popular subset of XP called TDD. (Yes, yes, we know we've just grossly oversimplified this, please cut us a little slack here.)

TDD is a LARGE topic, and we're only going to scratch the surface in this book. But we hope that the way we're going about developing the "Sink a Startup" game gives you some sense of TDD.

Here is a partial list of key ideas in TDD:

- Write the test code *first*.

- Develop in iteration cycles.

- Keep it (the code), simple.

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

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

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

- No killer schedules; work regular hours.

Set realistic schedules, based around small releases.

Writing test code for the SimpleStartup class

We need to write test code that can make a SimpleStartup object and run its methods. For the SimpleStartup class, we really care about only the *checkYourself()* method, although we *will* have to implement the

setLocationCells() method in order to get the *checkYourself()* method to run correctly.

Take a good look at the prepcode below for the *checkYourself()* method (the *setLocationCells()* method is a no-brainer setter method, so we're not worried about it, but in a 'real' application we might want a more robust 'setter' method, which we *would* want to test).

Then ask yourself, "If the checkYourself() method were implemented, what test code could I write that would prove to me the method is working correctly?"

Based on this prepcode:

```
METHOD String checkYourself(String userGuess)
  GET the user guess as a String parameter
  CONVERT the user guess to an int
  REPEAT with each of the location cells in the int array
    // COMPARE the user guess to the location cell
    IF the user guess matches
          INCREMENT the number of hits
          // FIND OUT if it was the last location cell:
          IF number of hits is 3, RETURN "Kill" as the result
          ELSE it was not a kill, so RETURN"Hit"
          END IF
    ELSE the user guess did not match, so RETURN "Miss"
    END IF
 END REPEAT
END METHOD
```

Here's what we should test:

1. Instantiate a SimpleStartup object.
2. Assign it a location (an array of 3 ints, like {2,3,4}).
3. Create a String to represent a user guess ("2", "0", etc.).
4. Invoke the checkYourself() method passing it the fake user guess.
5. Print out the result to see if it's correct ("passed" or "failed").

There are no Dumb Questions

Q: Maybe I'm missing something here, but how exactly do you run a test on something that doesn't yet exist!?

A: You don't. We never said you start by *running* the test; you start by *writing* the test. At the time you write the test code, you won't have anything to run it against, so you probably won't be able to compile it until you write 'stub' code that can compile, but that will always cause the test to fail (like, return null.)

Q: Then I still don't see the point. Why not wait until the code is written, and then whip out the test code?

A: The act of thinking through (and writing) the test code helps clarify your thoughts about what the method itself needs to do.

As soon as your implementation code is done, you already have test code just waiting to validate it. Besides, you *know* if you don't do it now, you'll *never* do it. There's always something more interesting to do.

Ideally, write a little test code, then write *only* the implementation code you need in order to pass that test. Then write a little *more* test code and write *only* the new implementation code needed to pass *that* new test. At each test iteration, you run *all* the previously-written tests, so that you always prove that your latest code additions don't break previously-tested code.

Test code for the SimpleStartup class

```
public class SimpleStartupTestDrive {

   public static void main (String[] args) {

      SimpleStartup dot = new SimpleStartup();

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

      dot.setLocationCells(locations);

      String userGuess = "2";

      String result = dot.checkYourself(userGuess);

      String testResult = "failed";

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

           testResult = "passed";

       }

       System.out.println(testResult);

    }

}
```

instantiate a SimpleStartup object

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

invoke the setter method on the Startup.

make a fake user guess

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

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

print out the test result ("passed or failed")

sharpen your Pencil

In the next couple of pages we implement the SimpleStartup class, and then later we return to the test class. Looking at our test code above, what else should be added? What are we *not* testing in this code, that we *should* be testing for? Write your ideas (or lines of code) below:

The checkYourself() method

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

Just the new stuff

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

There are no Dumb Questions

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

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

Q: In the beginning of the book, there was an example of a *for* loop that was really different from this one—are there two different styles of *for* loops?

A: Yes! From the first version of Java there has been a single kind of *for* loop (explained later in this chapter) that looks like this:

```
for (int i = 0; i < 10; i++) {
   // do something 10 times
}
```

You can use this format for any kind of loop you need. But... beginning with Java 5.0 (Tiger), you can also use the *enhanced* for loop (that's the official description) when your loop needs to iterate over the elements in an array (or *another* kind of collection, as you'll see in the *next* chapter). You can always use the plain old for loop to iterate over an array, but the *enhanced* for loop makes it easier.

Final code for SimpleStartup and SimpleStartupTester

```
public class SimpleStartupTestDrive {

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


public class SimpleStartup {

   int[] locationCells;
   int numOfHits = 0;
```

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

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

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

There’s a little bug lurking here. It compiles and runs, but sometimes... don’t worry about it for now, but we *will* have to face it a little later.

SHARPEN YOUR PENCIL

We built the test class, and the SimpleStartup class. But we still haven't made the actual *game*. Given the code on the opposite page, and the spec for the actual game, write in your ideas for prepcode for the game class. We've given you a few lines here and there to get you started. The actual game code is on the next page, so ***don't turn the page until you do this exercise!***

You should have somewhere between 12 and 18 lines (including the ones we wrote, but not including lines that have only a curly brace).

Prepcode for the SimpleStartupGame class

Everything happens in main()

There are some things you'll have to take on faith. For example, we have one line of prepcode that says, "GET user input from command-line". Let me tell you, that's a little more than we want to implement from scratch right now. But happily, we're using OO. And that means you get to ask some *other* class/object to do something for you, without worrying about ***how*** it does it. When you write prepcode, you should assume that *somehow* you'll be able to do whatever you need to do, so you can put all your brainpower into working out the logic.

```
public static void main (String [] args)
       DECLARE an int variable to hold the number of user
guesses, named numOfGuesses, set it to 0.
       MAKE a new SimpleStartup instance
       COMPUTE a random number between 0 and 4 that will be the
starting location cell position
       MAKE an int array with 3 ints using the randomly-generated
```

```
number, that number incremented by
       1, and that number incremented by 2 (example: 3,4,5)
       INVOKE the setLocationCells() method on the SimpleStartup
instance
       DECLARE a boolean variable representing the state of the
game, named isAlive. SET it to true

       WHILE the Startup is still alive (isAlive == true) :
          GET user input from the command line
          // CHECK the user guess
          INVOKE the checkYourself() method on the SimpleStartup
instance
          INCREMENT numOfGuesses variable
          // CHECK for Startup death
          IF result is "kill"
               SET isAlive to false (which means we won't enter
the loop again)
               PRINT the number of user guesses
          END IF
    END WHILE
END METHOD
```

METACOGNITIVE TIP

Don't work one part of the brain for too long a stretch at one time. Working just the left side of the brain for more than 30 minutes is like working just your left *arm* for 30 minutes. Give each side of your brain a break by switching sides at regular intervals. When you shift to one side, the other side gets to rest and recover. Left-brain activities include things like step-by-step sequences, logical problem-solving, and analysis, while the right-brain kicks in for metaphors, creative problem-solving, pattern-matching, and visualizing.

BULLET POINTS 

- Your Java program should start with a high-level design.
- Typically you'll write three things when you create a new class:
    - ***prepcode***
    - ***testcode***
    - ***real (Java) code***
- Prepcode should describe *what* to do, not *how* to do it. Implementation comes later.
- Use the prepcode to help design the test code.
- Write test code *before* you implement the methods.
- Choose *for* loops over *while* loops when you know how many times you want to repeat the loop code.
- Use the pre/post *increment* operator to add 1 to a variable (`x++;`)
- Use the pre/post *decrement* to subtract 1 from a variable (`x--;`)
- Use **`Integer.parseInt()`** to get the int value of a String.
- **`Integer.parseInt()`** works only if the String represents a digit ("0","1","2", etc.)
- Use *break* to leave a loop early (i.e. even if the boolean test condition is still true).
- **How many hits did you get last month?** Including repeat visitors?

The game's main() method

Just as you did with the SimpleStartup class, be thinking about parts of this code you might want (or need) to improve. The numbered things are for ❶ stuff we want to point out. They're explained on the opposite page. Oh, if you're wondering why we skipped the test code phase for this class, we don't need a test class for the game. It has only one method, so what would you do in your test code? Make a separate class that would call main() on this class? We didn't bother.

random() and getUserInput()

Two things that need a bit more explaining, are on this page. This is just a quick look to keep you going; more details on the GameHelper class are at the end of this chapter.

One last class: GameHelper

We made the *Startup* class.

We made the *game* class.

All that's left is the *helper* class— the one with the getUserInput() method. The code to get command-line input is more than we want to explain right now. It opens up way too many topics best left for later. (Later, as in chapter 14.)

Just copy* the code below and compile it into a class named GameHelper. Drop all three classes (SimpleStartup, SimpleStartupGame, GameHelper)

into the same directory, and make it your working directory.

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

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

*We know how much you enjoy typing, but for those rare moments when you'd rather do something else, we've made the Ready-bake Code available on wickedlysmart.com.

Let's play

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

A complete game interaction (your mileage may vary)

What's this? A bug ?

Gasp!

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

A different game interaction (yikes)

SHARPEN YOUR PENCIL

It's a cliff-hanger!

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

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

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

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

More about for loops

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

Regular (non-enhanced) for loops

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

How the compiler sees it:

* create a variable *i* and set it to 0.

* repeat while *i* is less than 100.

* at the end of each loop iteration, add 1 to *i*

Part One: *initialization*

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

Part Two: *boolean test*

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

Part Three: *iteration expression*

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

Trips through a loop

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

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

Difference between for and while

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

output:

++ --

Pre and Post Increment/Decrement Operator

The shortcut for adding or subtracting 1 from a variable.

```
x++;
```

is the same as:

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

They both mean the same thing in this context: “add 1 to the current value of x” or “***increment*** x by 1”

And:

```
x--;
```

is the same as:

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

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

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

produces: x is 1, z is 1

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

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

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

The enhanced for loop

Beginning with Java 5.0 (Tiger), the Java language added a second kind of *for* loop called the *enhanced for*, that makes it easier to iterate over all the elements in an array or other kinds of collections (you'll learn about *other* collections in the next chapter). That's really all that the enhanced for gives you—a simpler way to walk through all the elements in the collection. We'll revisit the *enhanced for loop* in the next chapter, when we talk about collections that *aren't* arrays. (Later in the book we'll be talking about a newer, more super-charged way of looping called "streams", stay tuned!)

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

How the compiler sees it:

* Create a String variable called *name* and set it to null.

* Assign the first value in *nameArray* to name.

* Run the body of the loop (the code block bounded by curly braces).

* Assign the next value in *nameArray* to name.

* Repeat while *there are still elements in the array.*

Part One: *iteration variable declaration*

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

Part Two: *the actual collection*

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

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

CONVERTING A STRING TO AN INT

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

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

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

For example, ***this won’t work:***

String num = “2”;

int x = 2;

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

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

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

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

Casting primitives

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

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

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

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

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

```
long y = 40002;
// 40002 exceeds the 16-bit limit of a short

short x = (short) y; // x now equals -25534!
```

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

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

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

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

BE the JVM

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

Code Magnets

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

Across

1. Fancy computer word for build

4. Multi-part loop

6. Test first

7. 32 bits

10. Method’s answer

11. Prepcode-esque

13. Change

15. The big toolkit

17. An array unit

18. Instance or local

20. Automatic toolkit

22. Looks like a primitive, but..

25. Un-castable

26. Math method

28. Converter method

29. Leave early

Down

2. Increment type

3. Class’s workhorse

5. Pre is a type of _____

6. For’s iteration _____

7. Establish first value

8. While or For

9. Update an instance variable

12. Towards blastoff

14. A cycle

16. Talkative package

19. Method messenger (abbrev.)

21. As if

23. Add after

24. Pi house

26. Compile it and ____

27. ++ quantity

Inline JavaCross

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

Inline Mixed Messages

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

Inline Exercise Solutions

Be the JVM:

```
class Output {

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

```
  }

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

       if (y > 14) {
         System.out.println(" x = " + x);
         break;
       }
     }
   }
}
```

Code Magnets:

```
class MultiFor {

  public static void main(String [] args) {

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

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

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

Inline Puzzle Solutions

1. 1. Dive in A Quick Dip: Breaking the Surface
    a. The Way Java Works
    b. What you'll do in Java
    c. A Very Brief History of Java
        i. Speed and Memory Usage
        ii. Sharpen your pencil Answers
    d. Code structure in Java
        i. What goes in a source file?
        ii. What goes in a class?
        iii. What goes in a method?
    e. Anatomy of a class
    f. Writing a class with a main
    g. What can you say in the main method?
        i. Looping and looping and...
        ii. Simple boolean tests
    h. There are no dumb Questions
    i. Example of a while loop
    j. Conditional branching
    k. Coding a Serious Business Application
        i. Monday Morning at Bob's Java-Enabled House
    l. Phrase-O-Matic
    m. Code Magnets