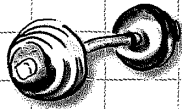


Your Brain on Java—A Learner's Guide

Head First Java™



Pump neurons with
the Brain Barbell
workouts

Gather your clues to
solve a Five-Minute
Java Mystery



See what
makes the JVM
tick and what
ticks it off

Watch Java objects
expose their inner
secrets on
Java Tabloid TV



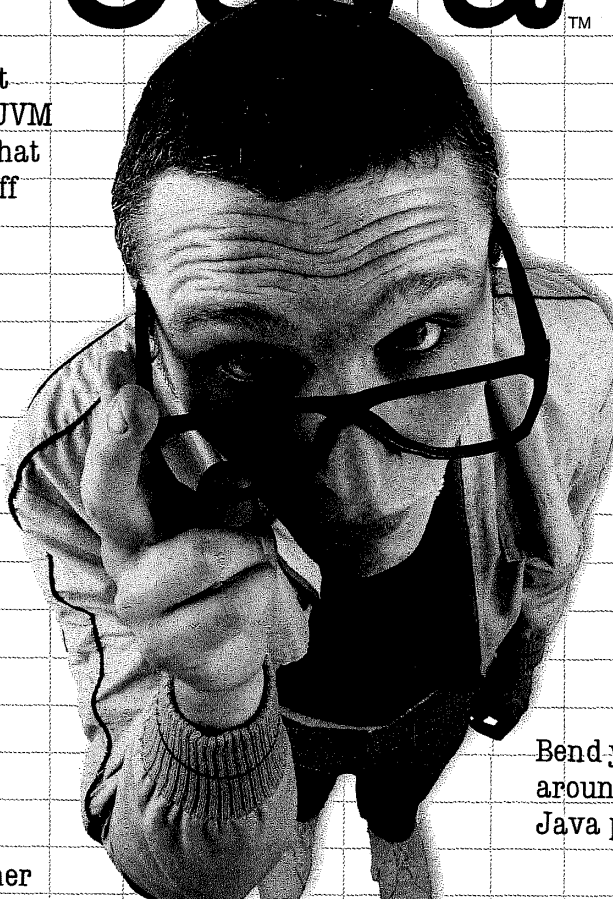
Whip up
some code with
Ready-Bake Java

Fool around in
the Java Library



Learn why Lucy
really keeps her
variables private

Bend your mind
around 42
Java puzzles



Head First Java™



Wouldn't it
be dreamy if there was
a Java book that was more
stimulating than waiting
in line at the DMV to renew
your driver's license? It's
probably just a fantasy...

Kathy Sierra
Bert Bates

O'REILLY®

Beijing • Cambridge • Köln • Paris • Sebastopol • Taipei • Tokyo

Head First Java™

by Kathy Sierra and Bert Bates

Copyright © 2003 O'Reilly & Associates, Inc. All rights reserved.

Printed in the United States of America.

Published by O'Reilly & Associates, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

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

Editor: Mike Loukides

Cover Designer: Edie Freedman

Interior Designers: Kathy Sierra and Bert Bates

Printing History:

May 2003: First Edition.

(You might want to get several copies—this could be the next Beanie Babies™ on eBay™.)

The O'Reilly logo is a registered trademark of O'Reilly & Associates, Inc. Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc., in the United States and other countries. O'Reilly & Associates, Inc. is independent of Sun Microsystems.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks.

Where those designations appear in this book, and O'Reilly & Associates, Inc. was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher and the authors assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

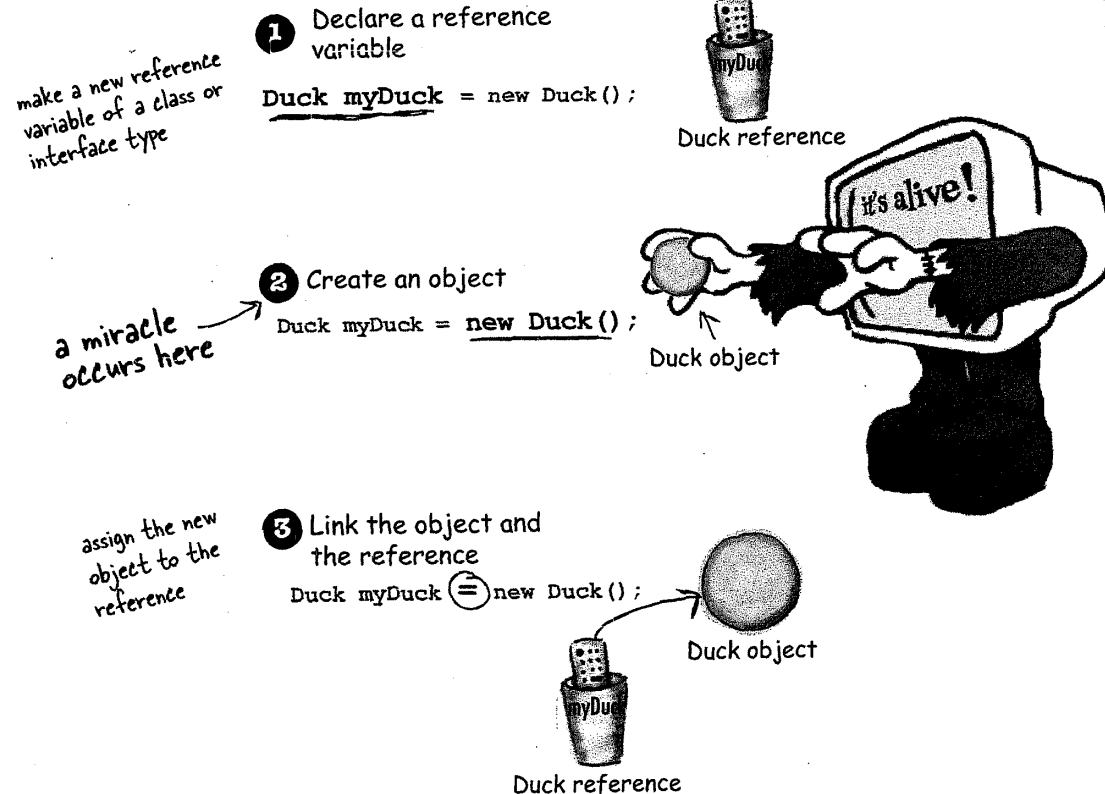
In other words, if you use anything in *Head First Java™* to, say, run a nuclear power plant or air traffic control system, you're on your own.

The miracle of object creation

Now that you know where variables and objects live, we can dive into the mysterious world of object creation. Remember the three steps of object declaration and assignment: declare a reference variable, create an object, and assign the object to the reference.

But until now, step two—where a miracle occurs and the new object is “born”—has remained a Big Mystery. Prepare to learn the facts of object life. *Hope you're not squeamish.*

Review the 3 steps of object declaration, creation and assignment:



constructing a new Duck

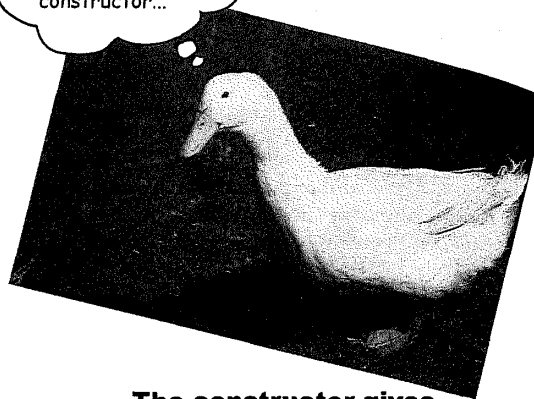
Construct a Duck

The key feature of a constructor is that it runs *before* the object can be assigned to a reference. That means you get a chance to step in and do things to get the object ready for use. In other words, before anyone can use the remote control for an object, the object has a chance to help construct itself. In our Duck constructor, we're not doing anything useful, but it still demonstrates the sequence of events.

```
public class Duck {  
  
    public Duck() {  
        System.out.println("Quack");  
    }  
}
```

← constructor code

If it Quacks like a constructor...



The constructor gives you a chance to step into the middle of new.

```
public class UseADuck {  
  
    public static void main (String[] args) {  
        Duck d = new Duck();  
    }  
}
```

← This calls the Duck constructor

```
File Edit Window Help Quack  
% java UseADuck  
Quack
```

Sharpen your pencil

A constructor lets you step into the middle of the object creation step—into the middle of **new**. Can you imagine conditions where that would be useful? Which of these might be useful in a Car class constructor, if the Car is part of a Racing Game? Check off the ones that you came up with a scenario for.

- ☐ Increment a counter to track how many objects of this class type have been made.
- ☐ Assign runtime-specific state (data about what's happening NOW).
- ☐ Assign values to the object's important instance variables.
- ☐ Get and save a reference to the object that's *creating* the new object.
- ☐ Add the object to an ArrayList.
- ☐ Create HAS-A objects.
- ☐ _____ (your idea here)

Nanoreview: four things to remember about constructors

- ① A constructor is the code that runs when somebody says `new` on a class type

```
Duck d = new Duck() ;
```

- ② A constructor must have the same name as the class, and *no* return type

```
public Duck(int size) { }
```

- ③ If you don't put a constructor in your class, the compiler puts in a default constructor. The default constructor is always a no-arg constructor.

```
public Duck() { }
```

- ④ You can have more than one constructor in your class, as long as the argument lists are different. Having more than one constructor in a class means you have overloaded constructors.

```
public Duck() { }
```

```
public Duck(int size) { }
```

```
public Duck(String name) { }
```

```
public Duck(String name, int size) { }
```

Doing all the Brain Barbells has been shown to produce a 42% increase in neuron size. And you know what they say, "Big neurons..."



Brain Barbell

What about superclasses?

When you make a Dog, should the Canine constructor run too?

If the superclass is abstract, should it even *have* a constructor?

We'll look at this on the next few pages, so stop now and think about the implications of constructors and superclasses.

^{there are no} Dumb Questions

Q: Do constructors have to be public?

A: No. Constructors can be **public**, **private**, or **default** (which means no access modifier at all). We'll look more at *default* accessn chapter 16 and appendix B.

Q: How could a **private** constructor ever be useful? Nobody could ever call it, so nobody could ever make a new object!

A: But that's not exactly right. Marking something **private** doesn't mean *nobody* can access it, it just means that *nobody outside the class* can access it. Bet you're thinking "Catch 22". Only code from the *same* class as the class-with-private-constructor can make a new object from that class, but without first making an object, how do you ever get to run code from that class in the first place? How do you ever get to anything in that class? *Patience grasshopper*. We'll get there in the next chapter.

you are here > 245

space for an object's superclass parts

Wait a minute... we never DID talk about superclasses and inheritance and how that all fits in with constructors.

Here's where it gets fun. Remember from the last chapter, the part where we looked at the Snowboard object wrapping around an inner core representing the Object portion of the Snowboard class? The Big Point there was that every object holds not just its *own* declared instance variables, but also *everything from its superclasses* (which, at a minimum, means class Object, since *every* class extends Object).

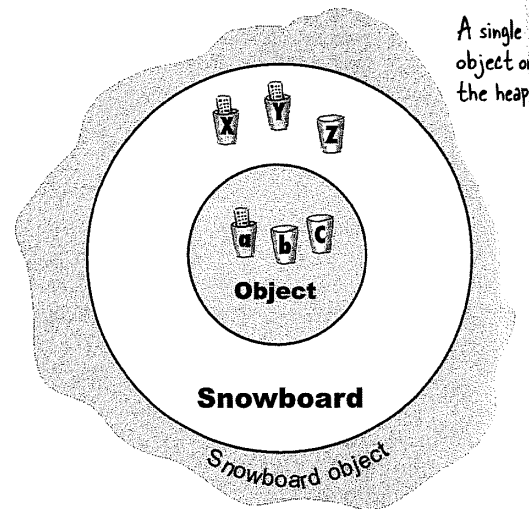
So when an object is created (because somebody said **new**; there is *no other way* to create an object other than someone, somewhere saying **new** on the class type), the object gets space for *all* the instance variables, from all the way up the inheritance tree. Think about it for a moment... a superclass might have setter methods encapsulating a private variable. But that variable has to live *somewhere*. When an object is created, it's almost as though *multiple* objects materialize—the object being new'd and one object per each superclass. Conceptually, though, it's much better to think of it like the picture below, where the object being created has *layers* of itself representing each superclass.

Object
Foo a; int b; int c;
equals() getClass() hashCode() toString()

Object has instance variables encapsulated by access methods. Those instance variables are created when any subclass is instantiated. (These aren't the *REAL* Object variables, but we don't care what they are since they're encapsulated)

Snowboard
Foo x Foo y int z
turn() shred() getAir() loseControl()

Snowboard also has instance variables of its own, so to make a Snowboard object we need space for the instance variables of both classes.



There is only *ONE* object on the heap here. A Snowboard object. But it contains both the Snowboard parts of itself and the Object parts of itself. All instance variables from both classes have to be here.

The role of superclass constructors in an object's life.

All the constructors in an object's inheritance tree must run when you make a new object.

Let that sink in.

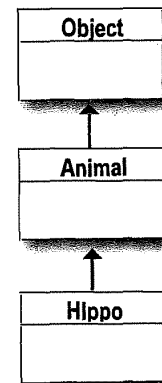
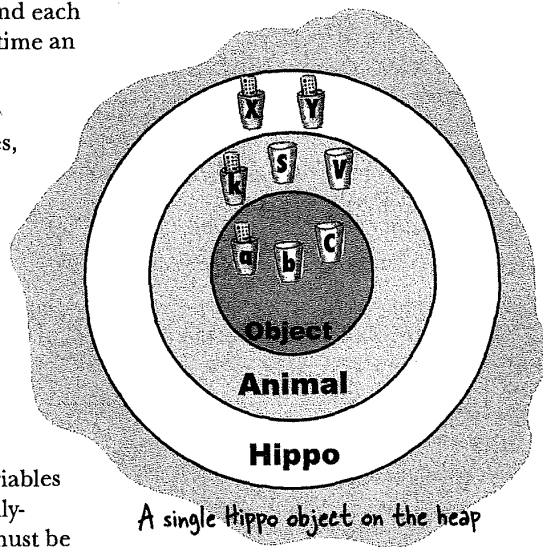
That means every superclass has a constructor (because every class has a constructor), and each constructor up the hierarchy runs at the time an object of a subclass is created.

Saying **new** is a Big Deal. It starts the whole constructor chain reaction. And yes, even abstract classes have constructors. Although you can never say **new** on an abstract class, an abstract class is still a superclass, so its constructor runs when someone makes an instance of a concrete subclass.

The super constructors run to build out the superclass parts of the object. Remember, a subclass might inherit methods that depend on superclass state (in other words, the value of instance variables in the superclass). For an object to be fully-formed, all the superclass parts of itself must be fully-formed, and that's why the super constructor *must* run. All instance variables from every class in the inheritance tree have to be declared and initialized. Even if **Animal** has instance variables that **Hippo** doesn't inherit (if the variables are private, for example), the **Hippo** still depends on the **Animal** methods that *use* those variables.

When a constructor runs, it immediately calls its superclass constructor, all the way up the chain until you get to the class **Object** constructor.

On the next few pages, you'll learn how superclass constructors are called, and how you can call them yourself. You'll also learn what to do if your superclass constructor has arguments!



A new Hippo object also IS-A Animal and IS-A Object. If you want to make a Hippo, you must also make the Animal and Object parts of the Hippo.

This all happens in a process called Constructor Chaining.

Java

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

—Ken Arnold, co-author
(with James Gosling, creator of Java),
The Java Programming Language

"Beyond the engaging style
that drags you forward
from know-nothing into
exalted Java warrior
status, *Head First Java*
covers a huge amount of
practical matters that
other texts leave as the
dreaded 'exercise for the
reader...'"

—Dr. Dan Russell,
Director of User Sciences
and Experience Research,
IBM Almaden Research Center

"This stuff is so fricking
good it makes me wanna
WEEP! Unbelievable.
I've never seen anything
like it."

—Floyd Jones, BEA Systems



We know what you're thinking—why is there a duck in the
memory management section? Can a girl in a bathtub illus-
trate the fine points of polymorphism? Is this for real?

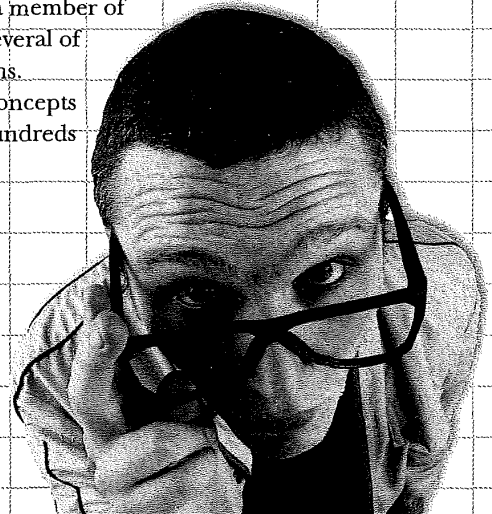
Prepare to have your mind opened! *Head First Java* combines puzzles,
strong visuals, mysteries, and soul-searching interviews with famous Java
objects to engage you in many different ways. It's fast, it's fun, and it's
effective. The more different ways you process the material, the more
you'll remember, and the better you'll be able to put what you learn to use.

Despite its playful appearance, *Head First Java* is serious stuff—a complete
introduction to object-oriented (OO) programming and Java. You'll learn
everything from fundamentals to advanced topics, including threads, net-
work sockets, and distributed programming with RMI. More importantly,
you'll learn how to think like an object-oriented developer. And you won't
just be reading and solving puzzles. You'll be writing real Java, and lots of
it, including the "sink a dot com" game, a trivia quiz, a networked musical
chat system, and a universal service browser.

If you want to be bored, go buy some other book. Please. (We'd tell you
which book, but we'd probably be sued.) But if you want to learn Java, get
a high-paying job (right), and improve your social life at the same time,
you need *Head First Java*.

Kathy Sierra has been a master trainer for Sun Microsystems, teaching
Sun's own instructors how to teach the latest Java technologies, and she
founded one of the largest Java community web sites in the world,
javaranch.com. Bert Bates is a member of
the development team for several of
Sun's Java Certification exams.

Together, they've used the concepts
in *Head First Java* to teach hundreds
of trainers and developers.



O'REILLY®



X0047SB2L3
New - Head First Java: Your Br
ain on Java - A Learner's Guid
e



once upon a time in Objectville

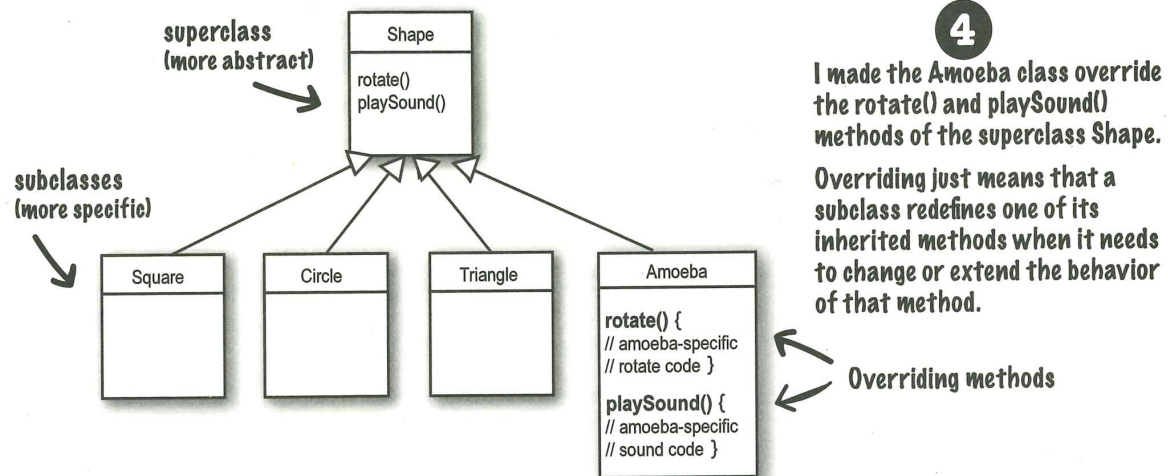
What about the Amoeba rotate()?

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

BRAD: Method.

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

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



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**.

LARRY: Well, I suppose I *could* become an OO programmer... you know, to woo women.

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

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

The suspense is killing me. Who got the chair?



Amy from the second floor.

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

What do you like about OO?

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

-Joy, 27, software architect

"Not messing around with code I've already tested, just to add a new feature."

-Brad, 32, programmer

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

-Josh, 22, beer drinker

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

-Chris, 39, project manager

"I can't believe Chris just said that. He hasn't written a line of code in 5 years."

-Daryl, 44, works for Chris

"Besides the chair?"

-Amy, 34, programmer



Brain Barbell

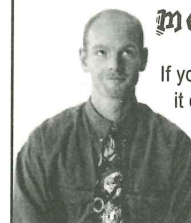
Time to pump some neurons.

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

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

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

metacognitive tip



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

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

- things the object **knows**
- things the object **does**

ShoppingCart
cartContents
addToCart() removeFromCart() checkout()

knows

does

Button
label color
setColor() setLabel() dePress() unDepress()

knows

does

Alarm
alarmTime alarmMode
setAlarmTime() getAlarmTime() isAlarmSet() snooze()

knows

does

Things an object *knows* about itself are called

- instance variables

Things an object can *do* are called

- methods

**instance
variables**
(state)

methods
(behavior)

Song
title artist
setTitle() setArtist() play()

knows

does

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.

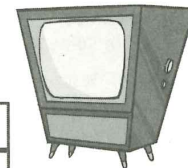
Things an object can *do* are called **methods**. When you design a class, you think about the data an object will need to know about itself, and you also design the methods that operate on that data. It's common for an object to have methods that read or write the values of the instance variables. For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

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

 **Sharpen your pencil**

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

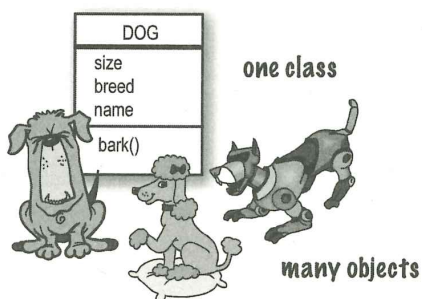
Television



**instance
variables**

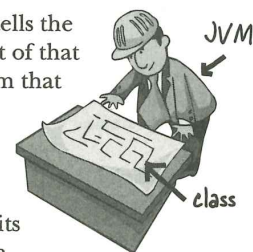
methods

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



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

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



Look at it this way...



An object is like one entry in your address book.

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

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

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

Making your first object

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

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

1 Write your class

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

    void bark() {
        System.out.println("Ruff! Ruff!");
    }
}
```

instance variables

a method

DOG
size
breed
name
bark()

2 Write a tester (TestDrive) class

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

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

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

```
class DogTestDrive {
    public static void main (String[] args) {
        Dog d = new Dog();
        d.size = 40;
        d.bark();
    }
}
```

make a Dog object

use the dot operator (.)

to set the size of the Dog

and to call its bark() method

dot
operator

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;
    }
}
```



MOVIE
title
genre
rating
playIt()

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

object 1

title
genre
rating

object 2

title
genre
rating

object 3

title
genre
rating

get the heck out of `main`

Quick! Get out of `main`!

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

The two uses of `main`:

- to test your real class
- to launch/start your Java application

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

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

The Guessing Game

Summary:

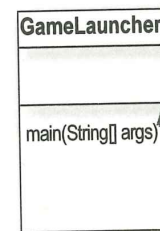
The guessing game involves a 'game' object and three 'player' objects. The game generates a random number between 0 and 9, and the three player objects try to guess it. (We didn't say it was a really *exciting* game.)

Classes:

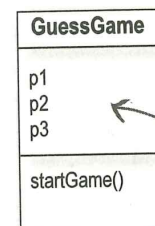
`GuessGame.class` `Player.class` `GameLauncher.class`

The Logic:

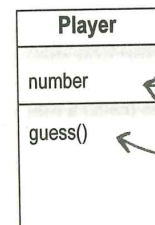
- 1) The `GameLauncher` class is where the application starts; it has the `main()` method.
- 2) In the `main()` method, a `GuessGame` object is created, and its `startGame()` method is called.
- 3) The `GuessGame` object's `startGame()` method is where the entire game plays out. It creates three players, then "thinks" of a random number (the target for the players to guess). It then asks each player to guess, checks the result, and either prints out information about the winning player(s) or asks them to guess again.



makes a `GuessGame` object and tells it to `startGame`



instance variables for the three players



the number this player guessed

method for making a guess

```

public class GuessGame {
    Player p1;
    Player p2;
    Player p3;

    public void startGame() {
        p1 = new Player();
        p2 = new Player();
        p3 = new Player();

        int guessp1 = 0;
        int guessp2 = 0;
        int guessp3 = 0;

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

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

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

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

            guessp1 = p1.number;
            System.out.println("Player one guessed " + guessp1);

            guessp2 = p2.number;
            System.out.println("Player two guessed " + guessp2);

            guessp3 = p3.number;
            System.out.println("Player three guessed " + guessp3);

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

            if (plisRight || p2isRight || p3isRight) {

                System.out.println("We have a winner!");
                System.out.println("Player one got it right? " + plisRight);
                System.out.println("Player two got it right? " + p2isRight);
                System.out.println("Player three got it right? " + p3isRight);
                System.out.println("Game is over.");
                break; // game over, so break out of the loop

            } else {
                // we must keep going because nobody got it right!
                System.out.println("Players will have to try again.");
            } // end if/else
        } // end loop
    } // end method
} // end class

```

GuessGame has three instance variables for the three Player objects

create three Player objects and assign them to the three Player instance variables

declare three variables to hold the three guesses the Players make

declare three variables to hold a true or false based on the player's answer

make a 'target' number that the players have to guess

call each player's guess() method

get each player's guess (the result of their guess() method running) by accessing the number variable of each player

check each player's guess to see if it matches the target number. If a player is right, then set that player's variable to be true (remember, we set it false by default)

if player one OR player two OR player three is right... (the || operator means OR)

otherwise, stay in the loop and ask the players for another guess.

Running the Guessing Game

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

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

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



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 the next chapter, you'll learn more about how this works.

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

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

there are no
Dumb Questions

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

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

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

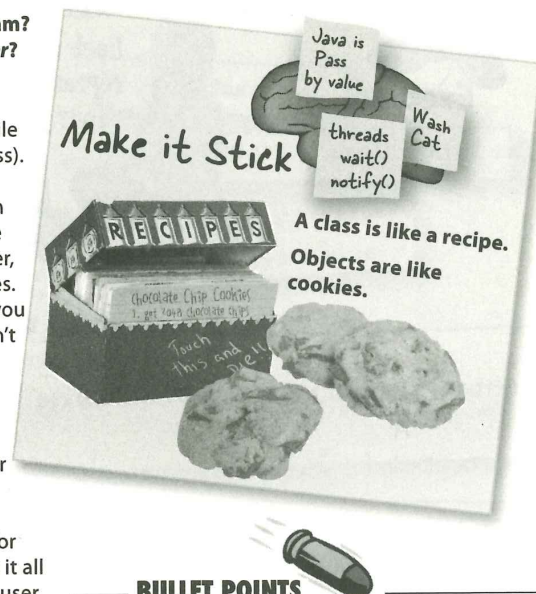
A: First of all, everything in Java goes in a class. So the constant for *pi* and the method for `random()`, although both `public` and `static`, are defined within the `Math` class. And you must keep in mind that these static (global-like) things are the exception rather than the rule in Java. They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

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

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

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

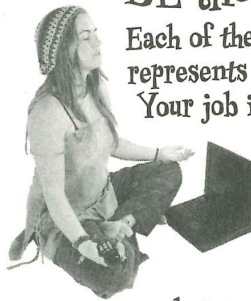
A: Yes, it would be a big pain to deliver a huge bunch of individual files to your end-users, but you won't have to. You can put all of your application files into a Java Archive – a *jar* file – that's based on the `pkzip` format. In the *jar* file, you can include a simple text file formatted as something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



BULLET POINTS

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

exercise: Be the Compiler



BE the compiler

Each of the Java files on this page represents a complete source file.

Your job is to play compiler and determine whether each of these files will compile. If they won't compile, how would you fix them, and if they do compile, what would be their output?

A

```
class TapeDeck {

    boolean canRecord = false;

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

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

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

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

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

B

```
class DVDPlayer {

    boolean canRecord = false;

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

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

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

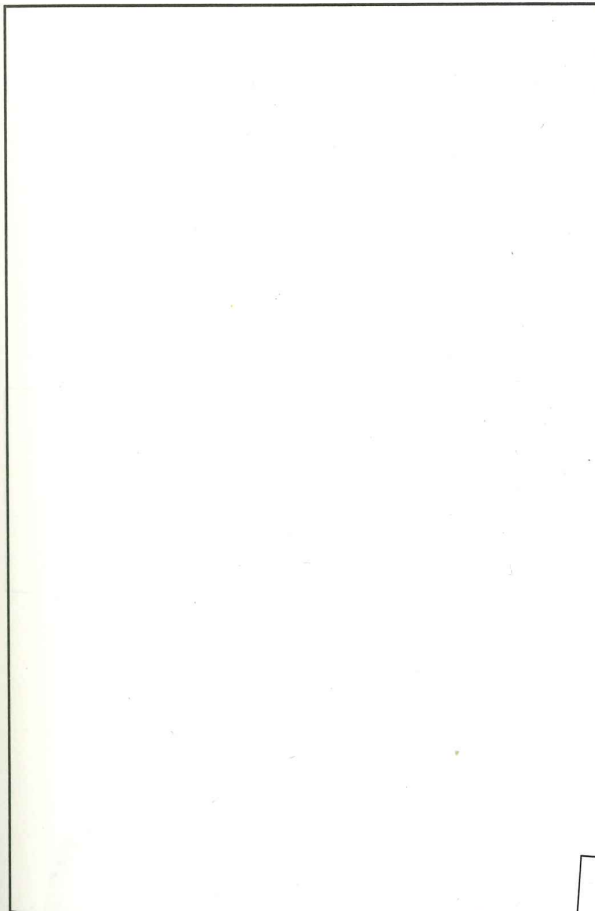
        if (d.canRecord == true) {
            d.recordDVD();
        }
    }
}
```



Exercise

Code Magnets

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



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

classes and objects

```
d.playSnare();
```

```
DrumKit d = new DrumKit();
```

```
boolean topHat = true;
boolean snare = true;
```

```
void playSnare() {
    System.out.print("bang bang da-bang");
}
```

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

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

```
        d.snare = false;
```

```
    class DrumKitTestDrive {
```

```
        d.playTopHat();
```

```
        class DrumKit {
```

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

you are here ▶ 41

puzzle: Pool Puzzle



Pool Puzzle



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

Output

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

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();

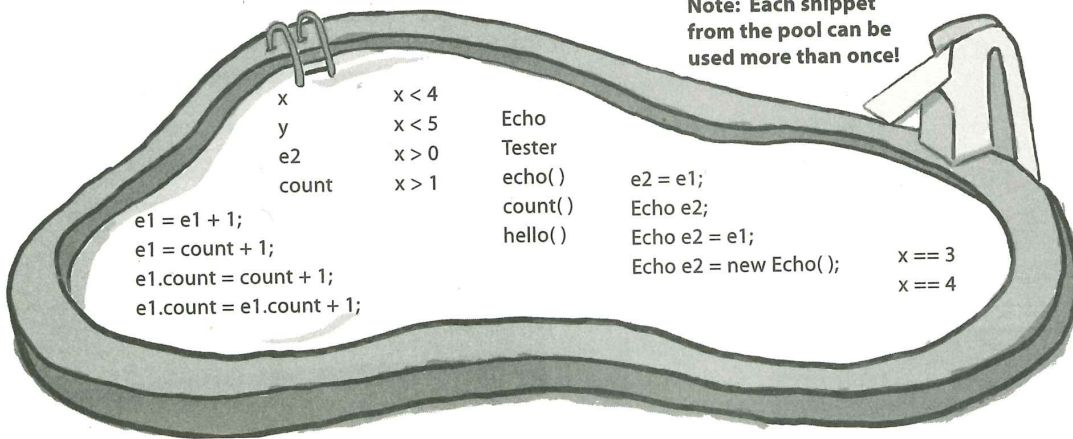
            _____

            if ( _____ ) {
                e2.count = e2.count + 1;
            }

            if ( _____ ) {
                e2.count = e2.count + e1.count;
            }

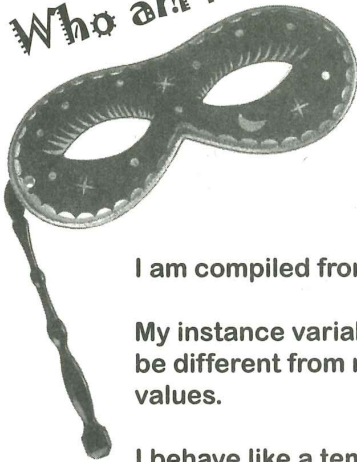
            x = x + 1;
        }
        System.out.println(e2.count);
    }
}
```

```
class _____ {
    int _____ = 0;
    void _____ {
        System.out.println("helloooo... ");
    }
}
```





Who am I?



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

Tonight's attendees:

Class Method Object Instance variable

I am compiled from a .java file.

class

My instance variable values can be different from my buddy's values.

I behave like a template.

I like to do stuff.

I can have many methods.

I represent 'state'.

I have behaviors.

I am located in objects.

I live on the heap.

I am used to create object instances.

My state can change.

I declare methods.

I can change at runtime.

exercise solutions



Exercise Solutions



Code Magnets:

```
class DrumKit {

    boolean topHat = true;
    boolean snare = true;

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

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

}

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

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

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

File Edit Window Help Dance

```
% java DrumKitTestDrive
bang bang ba-bang
ding ding da-ding
```

Be the Compiler:

A

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

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

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

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

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

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

B

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

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



Pool Puzzle

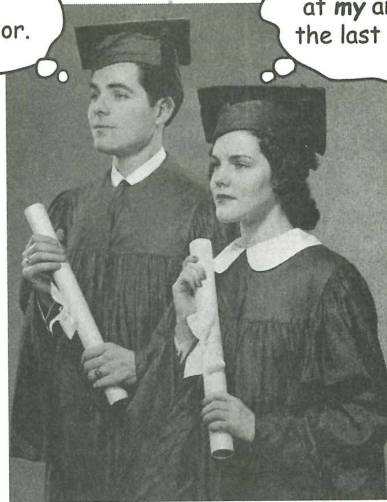
```
public class EchoTestDrive {
    public static void main(String [] args) {
        Echo e1 = new Echo();
        Echo e2 = new Echo(); // the correct answer
        - or -
        Echo e2 = e1; // is the bonus answer!
        int x = 0;
        while ( x < 4 ) {
            e1.hello();
         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

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

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

Finally...
the answers I've
worked so hard for.

He's been looking
at **my** answers for
the last four years.

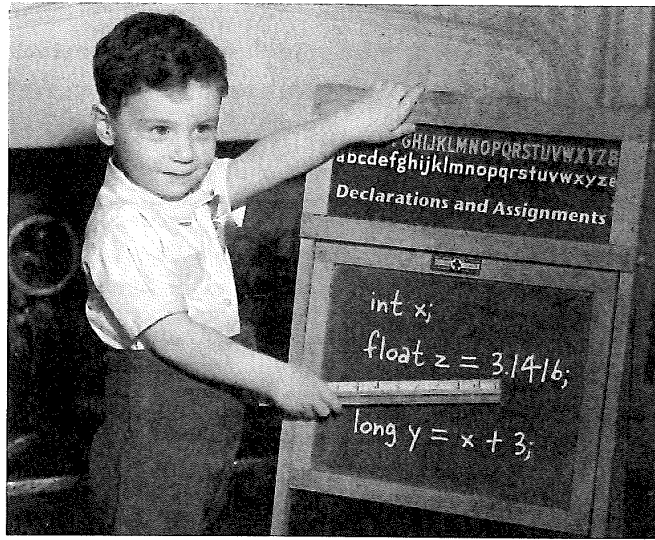


Who am I?

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 behavior.	object, class
Objects use me.	method, instance variable
I live on the heap.	object
I am used to create object instances.	class
My state can change.	object
I declare methods.	class
I can change at runtime.	object, instance variable

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

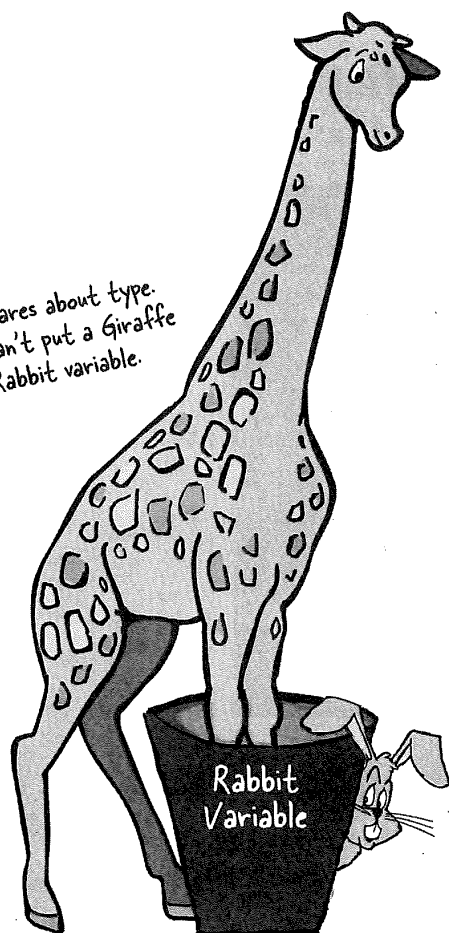
Know Your Variables



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

this is a new chapter 47

Java cares about type.
You can't put a Giraffe
in a Rabbit variable.



Declaring a variable

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

The compiler can spot most problems:

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

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

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

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

variables must have a type

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

variables must have a name

```
int count;
```

↑ ↓
type name

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

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

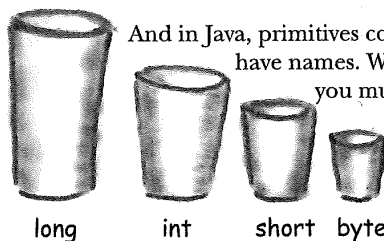
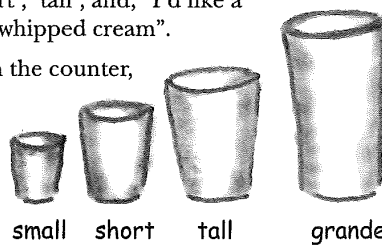
When you think of Java variables, think of cups. Coffee cups, tea cups, giant cups that hold lots and lots of beer, those big cups the popcorn comes in at the movies, cups with curvy, sexy handles, and cups with metallic trim that you learned can never, ever go in the microwave.

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

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

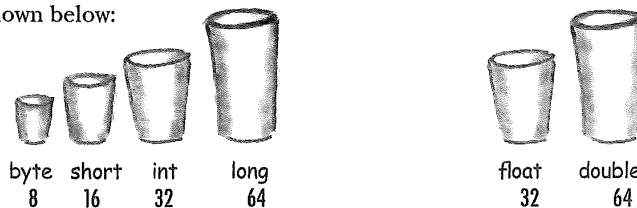
Primitives are like the cups they have at the coffeehouse. If you've been to a Starbucks, you know what we're talking about here. They come in different sizes, and each has a name like 'short', 'tall', and, "I'd like a 'grande' mocha half-caff with extra whipped cream".

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



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

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



Primitive Types

Type	Bit Depth	Value Range
------	-----------	-------------

boolean and char

boolean	(JVM-specific)	true or false
---------	----------------	---------------

char	16 bits	0 to 65535
------	---------	------------

numeric (all are signed)

integer

byte	8 bits	-128 to 127
------	--------	-------------

short	16 bits	-32768 to 32767
-------	---------	-----------------

int	32 bits	-2147483648 to 2147483647
-----	---------	---------------------------

long	64 bits	-huge to huge
------	---------	---------------

floating point

float	32 bits	varies
-------	---------	--------

double	64 bits	varies
--------	---------	--------

Primitive declarations with assignments:

```
int x;
```

```
x = 234;
```

```
byte b = 89;
```

```
boolean isFun = true;
```

```
double d = 3456.98;
```

```
char c = 'f';
```

```
int z = x;
```

```
boolean isPunkRock;
```

```
isPunkRock = false;
```

```
boolean powerOn;
```

```
powerOn = isFun;
```

```
long big = 3456789;
```

```
float f = 32.5f;
```

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

you are here ▶

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

Be sure the value can fit into the variable.



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

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

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

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

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

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

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

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

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

Sharpen your pencil

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

Based on what you know about the size and type of the primitive variables, see if you can figure out which of these are legal and which aren't. We haven't covered all the rules yet, so on some of these you'll have to use your best judgement. **Tip:** The compiler always errors 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;`
13. `int p = 3 * g + y;`

Back away from that keyword!

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

You already know the primitive types.

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

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

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

You've already seen some reserved words when we looked at writing our first main class:

```
public static void
```

don't use any of these
for your own names.

And the primitive types are reserved as well:

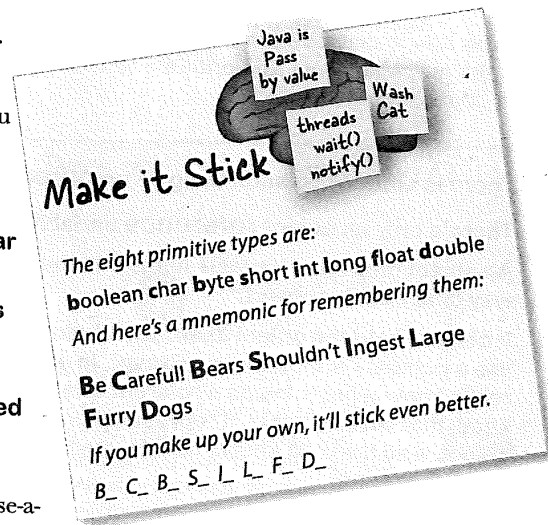
```
boolean char byte short int long float double
```

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

This table reserved.

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

Java's keywords and other reserved words. Don't use these for names, or there'll be hell to pay with the compiler.



Controlling your Dog object

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

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

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

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

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

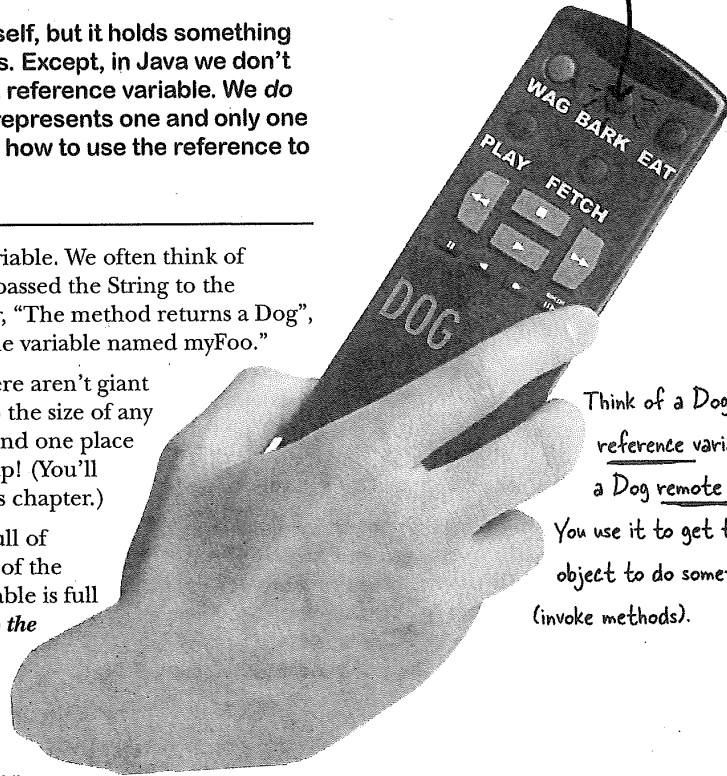
You use the dot operator (`.`) on a reference variable to say, "use the thing *before* the dot to get me the thing *after* the dot." For example:

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

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

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

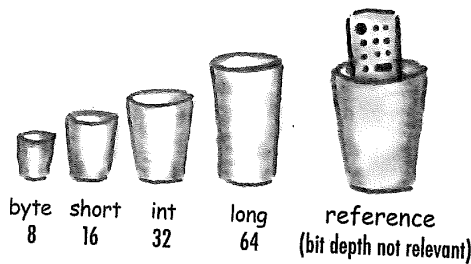
think of this
like this



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

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

();



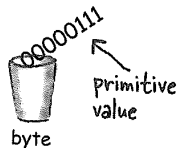
An object reference is just another variable value.

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

Primitive Variable

`byte x = 7;`

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

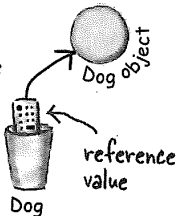


Reference Variable

`Dog myDog = new Dog();`

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

The Dog object itself does not go into the variable!



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

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

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

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

primitives and references

The 3 steps of object declaration, creation and assignment

1 `Dog myDog` **3** `=` **2** `new Dog();`

1 Declare a reference variable

`Dog myDog = new Dog();`

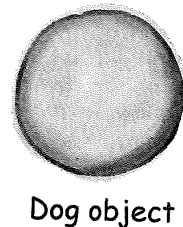
Tells the JVM to allocate space for a reference variable, and names that variable `myDog`. The reference variable is, forever, of type `Dog`. In other words, a remote control that has buttons to control a Dog, but not a Cat or a Button or a Socket.



2 Create an object

`Dog myDog = new Dog();`

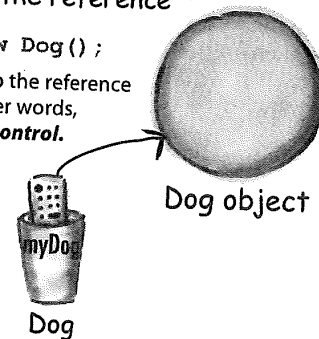
Tells the JVM to allocate space for a new Dog object on the heap (we'll learn a lot more about that process, especially in chapter 9.)



3 Link the object and the reference

`Dog myDog = new Dog();`

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



you are here ▶ 53

there are no
Dumb Questions

Q: How big is a reference variable?

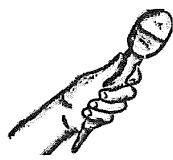
A: You don't know. Unless you're cozy with someone on the JVM's development team, you don't know how a reference is represented. There are pointers in there somewhere, but you can't access them. You won't need to. (OK, if you insist, you might as well just imagine it to be a 64-bit value.) But when you're talking about memory allocation issues, you're 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*? Sniff. Why, *why* can't I be a primitive? *I hate being a reference*. The responsibility, all the broken attachments...

Life on the garbage-collectible heap

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

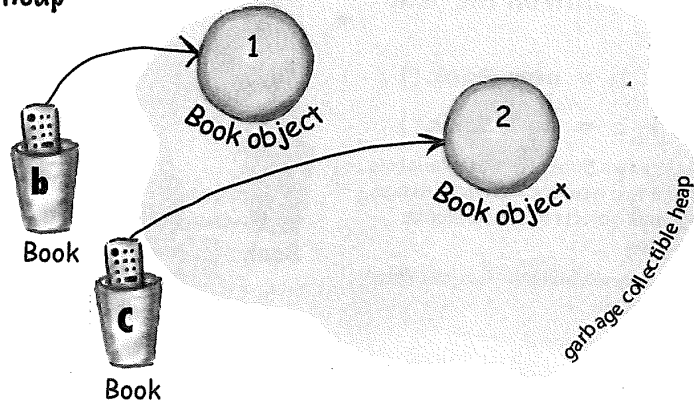
```
Book c = new Book();
```

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

The two Book objects are now living on the heap.

References: 2

Objects: 2



```
Book d = c;
```

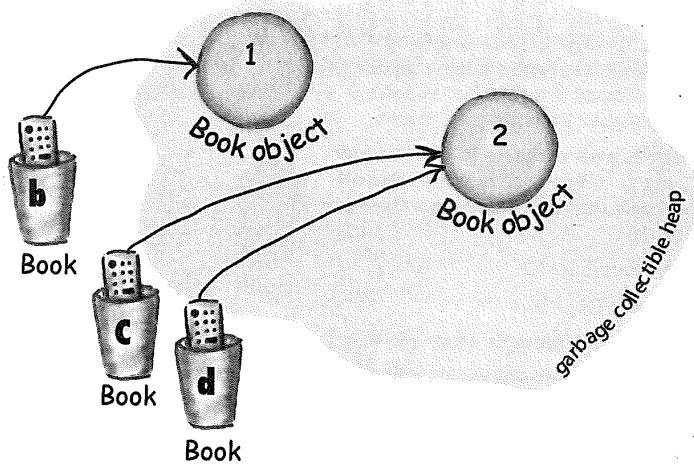
Declare a new Book reference variable. Rather than creating a new, third Book object, assign the value of variable *c* to variable *d*. But what does this mean? It's like saying, "Take the bits in *c*, make a copy of them, and stick that copy into *d*."

Both *c* and *d* refer to the same object.

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

References: 3

Objects: 2



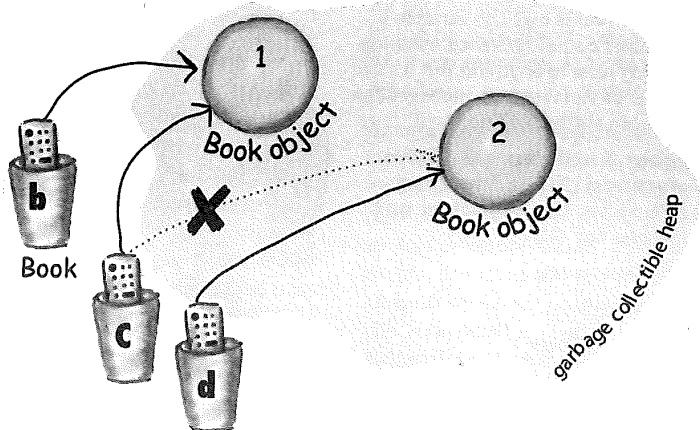
```
c = b;
```

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

Both *b* and *c* refer to the same object.

References: 3

Objects: 2



Life and death on the heap

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

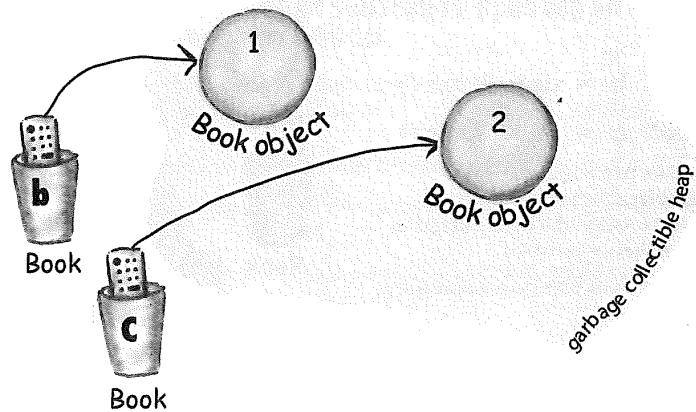
```
Book c = new Book();
```

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

The two book objects are now living on the heap.

Active References: 2

Reachable Objects: 2



```
b = c;
```

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

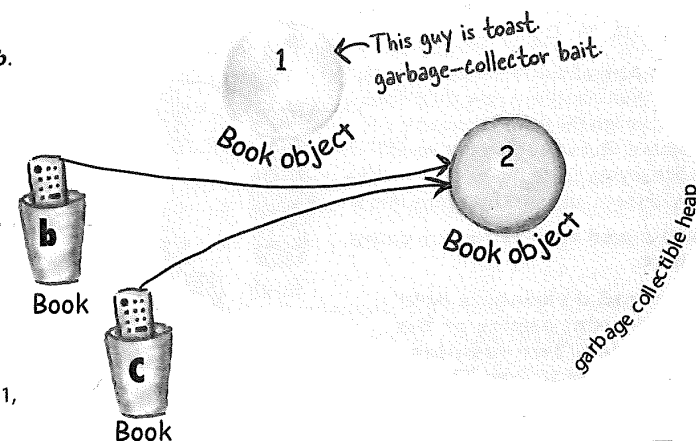
Both **b and **c** refer to the same object. Object 1 is abandoned and eligible for Garbage Collection (GC).**

Active References: 2

Reachable Objects: 1

Abandoned Objects: 1

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



```
c = null;
```

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

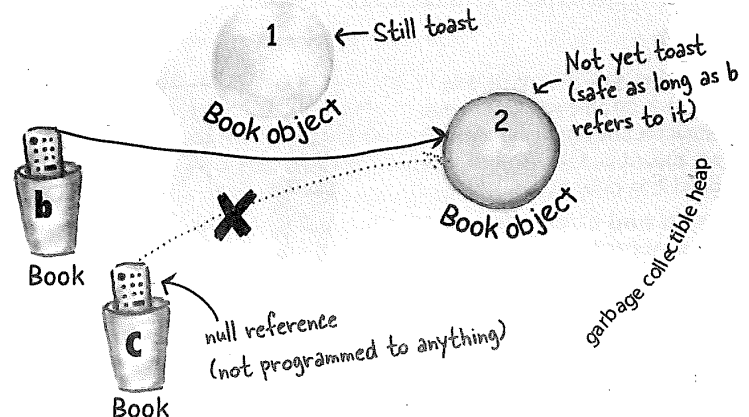
Object 2 still has an active reference (b**), and as long as it does, the object is not eligible for GC.**

Active References: 1

null References: 1

Reachable Objects: 1

Abandoned Objects: 1



An array is like a tray of cups.

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

```
int[] nums;
```

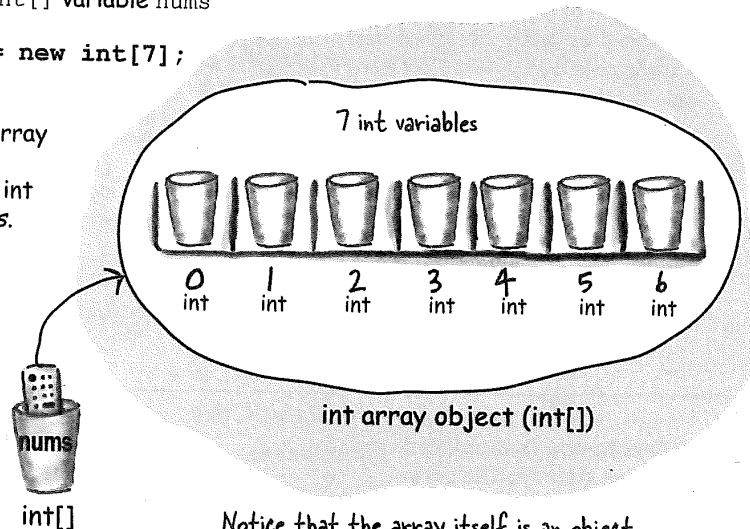
- 2 Create a new int array with a length of 7, and assign it to the previously-declared int[] variable nums

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

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

7 int variables

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



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

Arrays are objects too.

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

Every element in an array is just a variable. In other words, one of the eight primitive variable types (think: Large Furry Dog) or a

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

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

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

you are here ▶

57

an array of objects

Make an array of Dogs

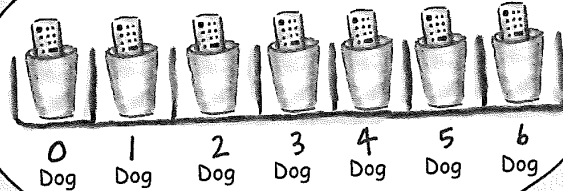
- 1 Declare a Dog array variable
`Dog[] pets;`

- 2 Create a new int 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!



Dog array object (Dog[])

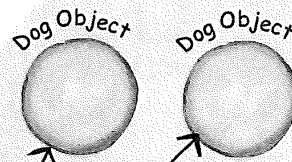
- 3 Create new Dog objects, and assign them to the array elements. Remember, elements in a Dog array are just Dog reference variables. We still need Dogs!

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

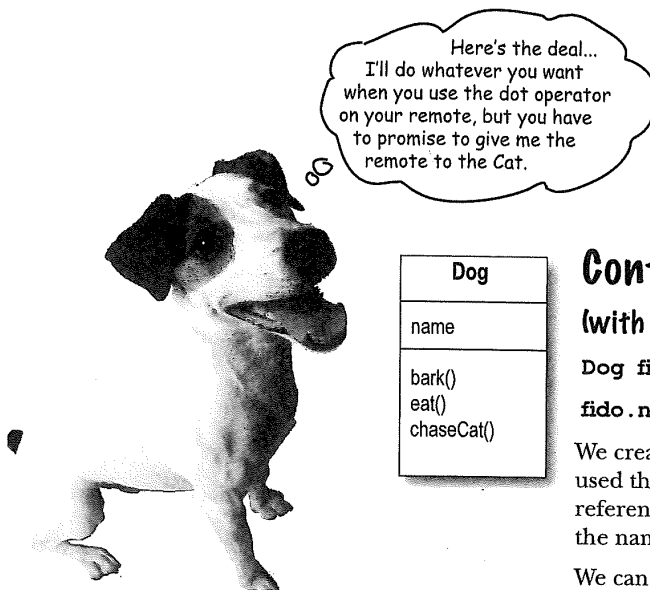
Sharpen your pencil

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

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



Dog array object (Dog[])



Dog
name
bark() eat() chaseCat()

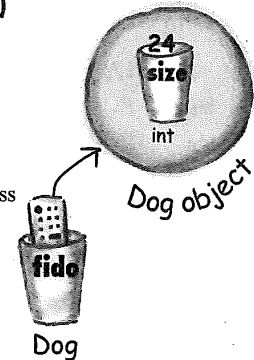
Control your Dog (with a reference variable)

```
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();
```



Java cares about type.

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

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

What happens if the Dog is in a Dog array?

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

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

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

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

using references

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

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

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

        // Hmmm... 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

Dog
name
bark() eat() chaseCat()

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

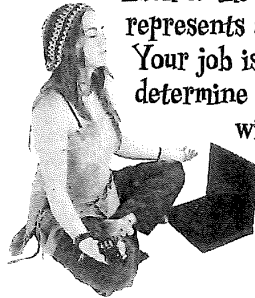
- Variables come in two flavors: primitive and reference.
- Variables must always be declared with a name and a type.
- A primitive variable value is the bits representing the value (5, 'a', true, 3.1416, etc.).
- A reference variable value is the bits representing a way to get to an object on the heap.
- 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.



Exercise

BE the compiler

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



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");
        }
    }
}
```

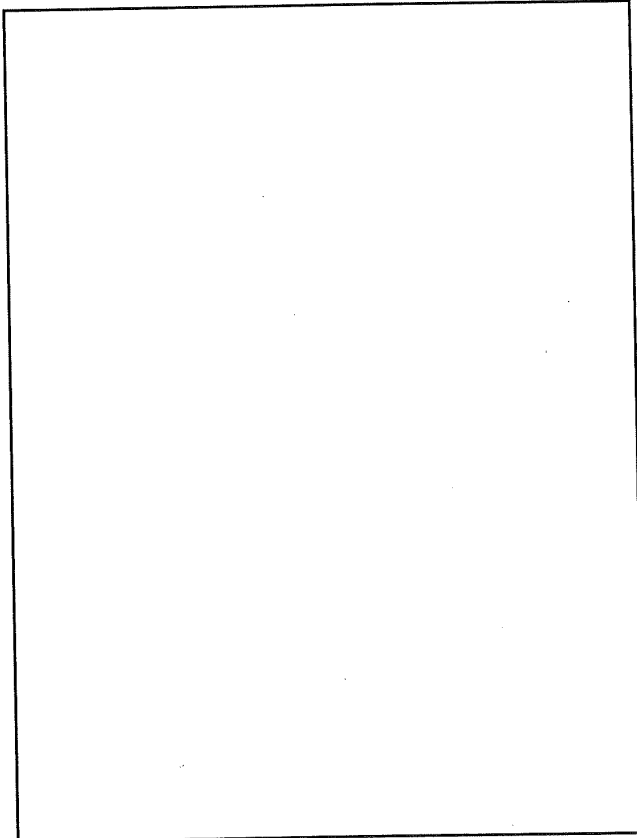
you are here ▶ 61

exercise: Code Magnets



Code Magnets

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



File Edit Window Help Bikini

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

```
int y = 0;
```

```
ref = index[y];
```

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

```
int ref;  
while (y < 4) {
```

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

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

```
String [] islands = new String[4];
```

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

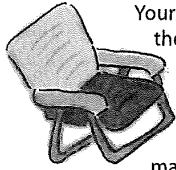
```
int [] index = new int[4];
```

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

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



Pool Puzzle



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

Output

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

Bonus Question!

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

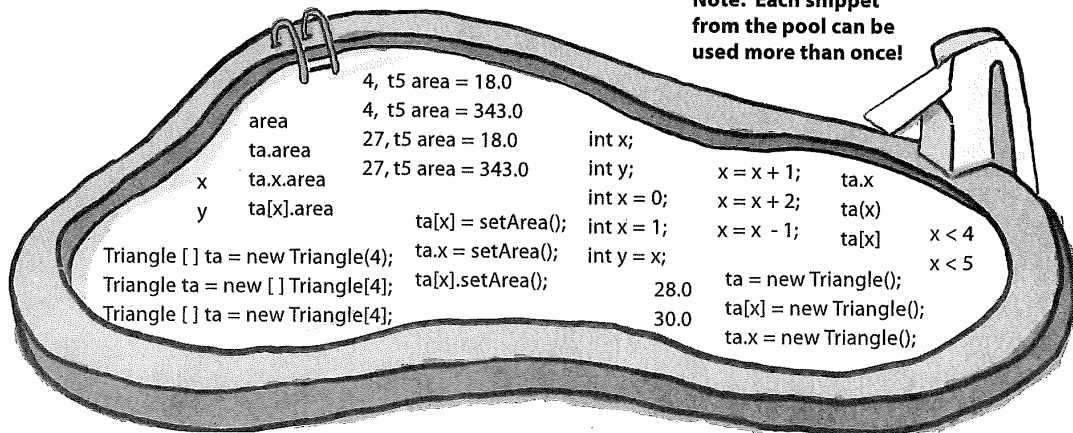
primitives and references

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

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

```
        _____
        while ( _____ ) {
            _____
            _____
            _____
            _____
            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!



you are here ▶

63

puzzle: Heap o' Trouble



A Heap o' Trouble

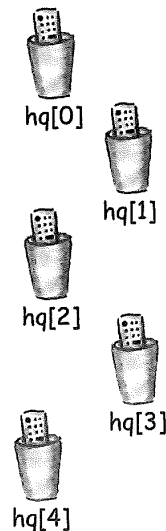
A short Java program is listed to the right. When '// do stuff' is reached, some objects and some reference variables will have been created. Your task is to determine which of the reference variables refer to which objects. Not all the reference variables will be used, and some objects might be referred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

Tip: Unless you're way smarter than us, you probably need to draw diagrams like the ones on page 10 and 12 of this chapter. Use a pencil so you can draw and then erase reference links (the arrows going from a reference remote control to an object).

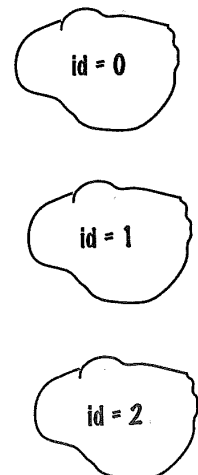
```
class HeapQuiz {
    int id = 0;
    public static void main(String [] args) {
        int x = 0;
        HeapQuiz [ ] hq = new HeapQuiz[5];
        while ( x < 3 ) {
            hq[x] = new HeapQuiz();
            hq[x].id = x;
            x = x + 1;
        }
        hq[3] = hq[1];
        hq[4]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

match each reference variable with matching object(s)
You might not have to use every reference.

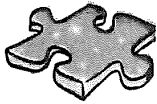
Reference Variables:



HeapQuiz Objects:



Actually we *could* do it in our heads, without pencils and paper. If we wanted to. Not that we've actually *tried* to do it that way. But we could. If we wanted to.



The case of the pilfered references

Five-Minute Mystery



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

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

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

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

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

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

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

exercise solutions



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
```

66 chapter 3

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

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

Remember: We have to actually make the Books objects!

```
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!



I'm looking at the answers, and it's, uh, not what we thought...



```
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
```

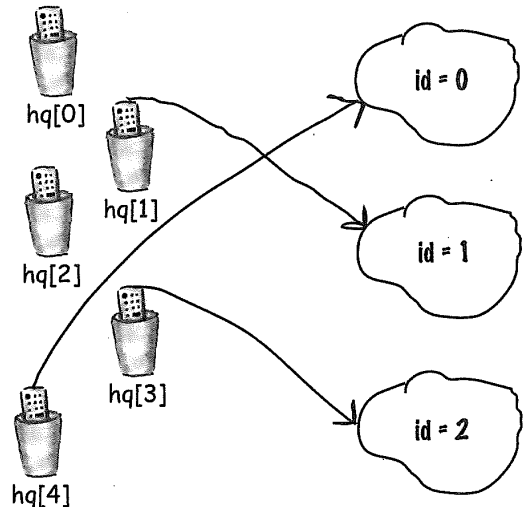
The case of the pilfered references

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.)

Reference Variables:

HeapQuiz Objects:



you are here >

67