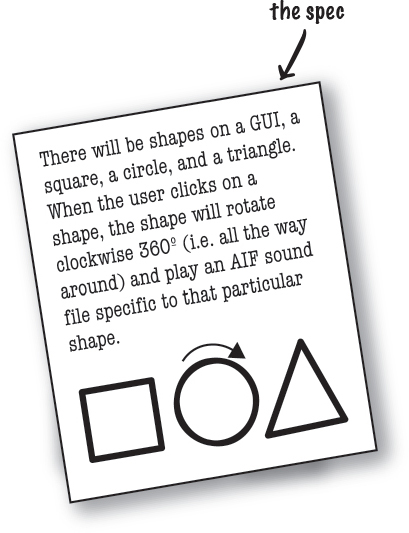
# Chapter 2. Classes and Objects: A Trip to Objectville



**I was told there would be objects.** In [Chapter 1](#_Chapter_1._Dive), 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 вытащить черта из 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 techies have. Laura, the procedural programmer, and Brad, the OO developer, both knew this would be a piece of cake.

Laura, sitting at her (non-adjustable) desk, thought to herself, “What are the things this program has to *do*? What ***procedures*** do we need?”. And she answered herself , “**rotate** and **playSound**.” So off she went to build the procedures. After all, what *is* a program if not a pile куча of procedures?

Brad, meanwhile, kicked back at the coffee shop 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 things 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 Laura built their programs, and for the answer to your burning question, ***“So, who got the Aeron and the desk?”***



**At Laura’s desk**

As she had done a gazillion times before, Laura set about writing her **Important Procedures**. She wrote **rotate** and **playSound** in no time.

**rotate(shapeNum) {**

**// make the shape rotate 360º**

**}**

**playSound(shapeNum) {**

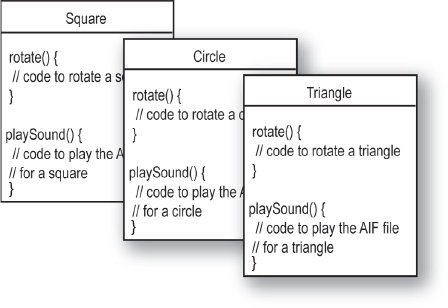
**// use shapeNum to lookup which**

**// AIF sound to play, and play it**

**}**

**At Brad’s laptop at the cafe**

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

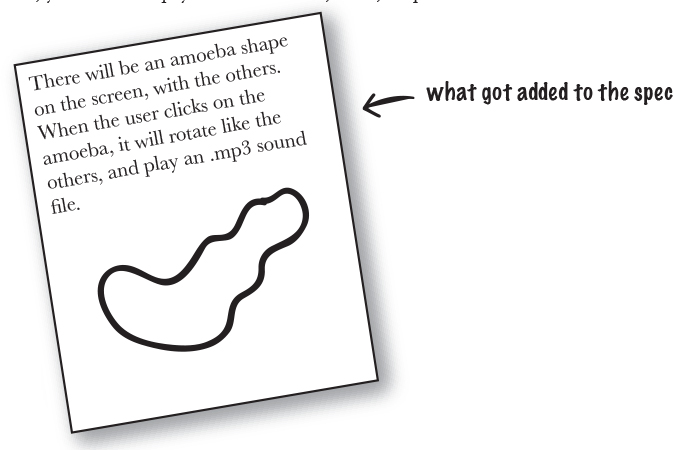


**Laura thought she’d nailed it Успешно справился. She could almost feel the rolled steel of the Aeron beneath her...**

**But wait! There’s been a spec change.**

“OK, *technically* you were first, Laura,” 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 Если бы у меня была копейка за каждый раз I’ve heard that one”*, thought Laura, knowing that spec-change-no-problem was a fantasy. *“And yet Brad looks strangely serene безмятежный. What’s up with that?”* Still, Laura held tight to her core belief that the OO way, while cute, was just slow. And that if you wanted to change her mind, you’d have to pry it from her cold вырвать его из ее холода, dead, carpal-tunnelled hands.



**Back at Laura’s desk**

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**

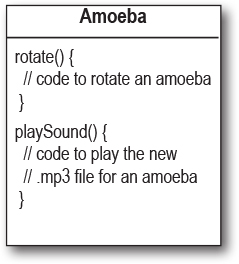
**// play amoeba .mp3 sound**

**}**

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

**At Brad’s laptop at the beach**

Brad smiled, sipped потягивание his fruit frappuccino, 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.



**Laura delivered just moments ahead of Brad.**

(Hah! So much for that foofy глупый OO nonsense). But the smirk ухмылка on Laura’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 прямоугольник 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 Laura, visualizing charred обгорелый 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 her head said, *“Big Mistake. Do you honestly think the spec won’t change again?”*



**Back at Laura’s desk**

She figured she better add rotation point arguments to the rotate procedure. ***A lot of code was affected*** ***затронутый***. 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**

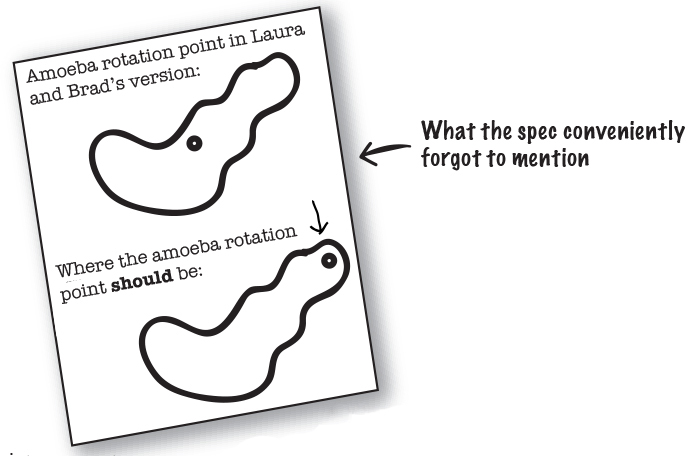
**// else**

**// use the xPt and yPt as**

**// the rotation point offset**

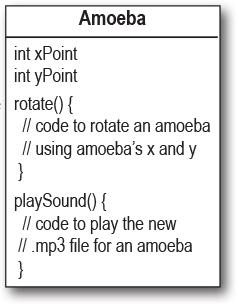
**// and then rotate**

**}**

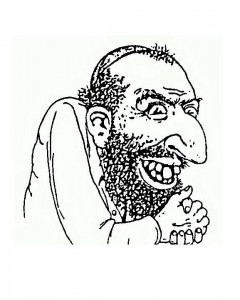


**At Brad’s laptop on his lawn газон chair at the Telluride Bluegrass Festival**

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 (via free festival WiFi) the revised program during a single Bela Fleck set.



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

** **



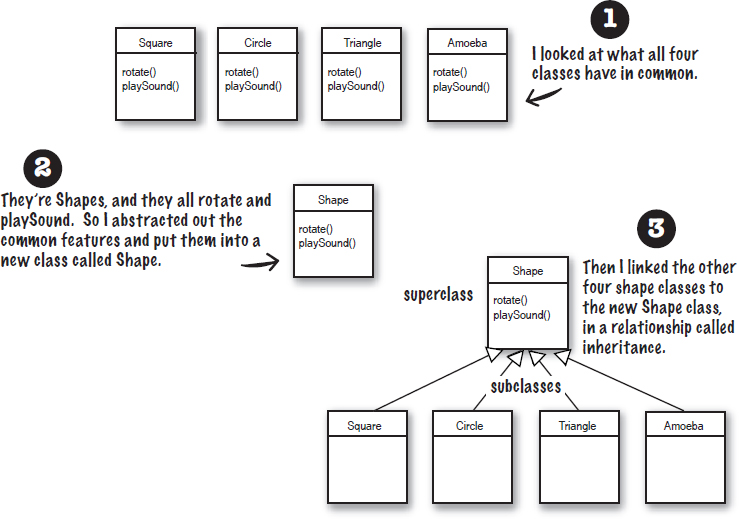
***Not so fast.*** Laura found a flaw изъян in Brad’s approach. And, since she was sure that if she got the chair and desk, she’d also be next in line for a promotion, she had to turn this thing around.

**LAURA:** 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*.

**LAURA:** 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, Laura.



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()?**

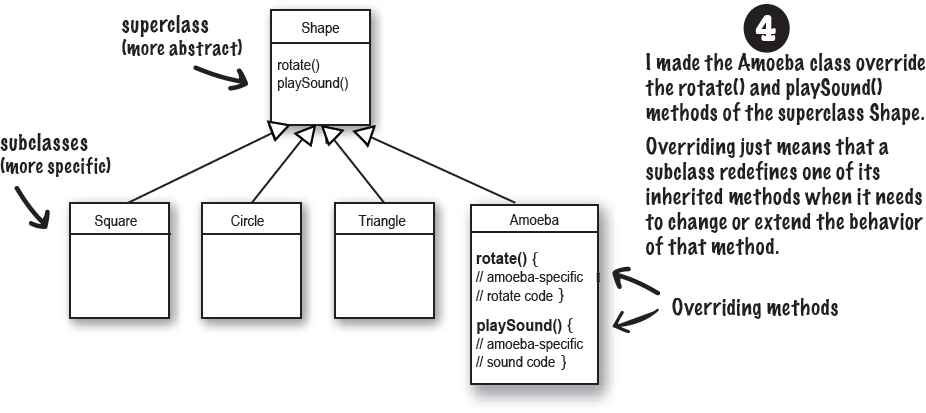
**LAURA:** Wasn’t that the whole problem here — that the amoeba shape had a completely different rotate and playSound procedure?



**BRAD: Method.**

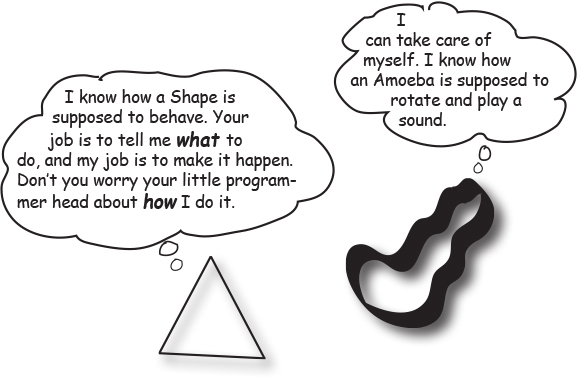
**LAURA:** Whatever. How can amoeba do something different if it “inherits” its functionality from the Shape class?

**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.

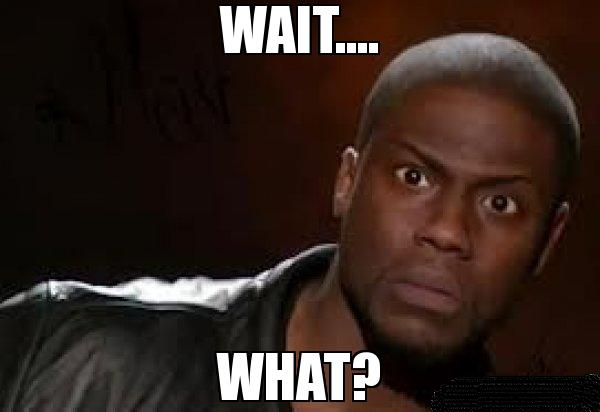


**LAURA:** How do you “tell” an Amoeba to do something? Don’t you have to call the procedure, sorry—*method,* and then tell it *which* thing to rotate?

**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 неведомый to all, the Project Manager had given the spec to *three* programmers. Amy completed the project faster since she got on with OO programming without arguing with her co-workers)

**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.”

-Jess, 22, foosball настольный футбол champion

“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, who hasn’t written a line of code in 5 years, just said that.”

-Daryl, 44, works for Chris

“Besides the chair?”

-Amy, 34, programmer

**Brain Power**

Images

**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**

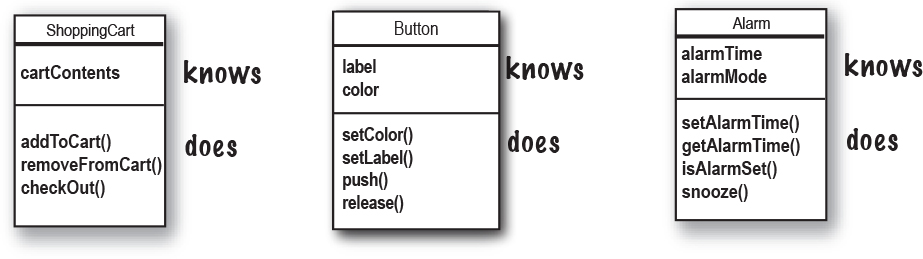


If you’re stuck on an exercise, try talking about it out loud. Speaking (and hearing) activates a different part of your brain. Although it works best if you have another person to discuss it with, pets work too. That’s how our dog learned polymorphism.

**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**

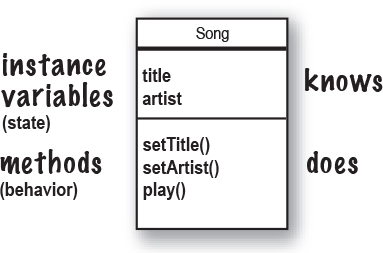


**Things an object knows about itself are called**

**instance variables**

**Things an object can do are called**

**methods**



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

**Think of instance as another way of saying object.**

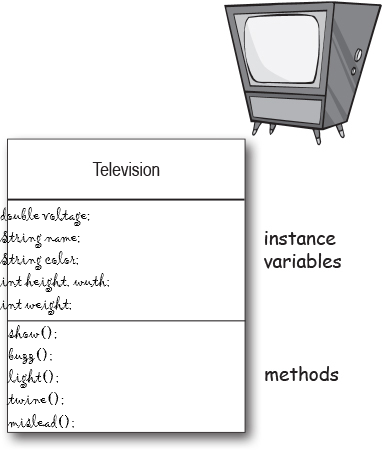
Things an object can ***do*** are called **methods**. When you design a class, you think about the data an object will need to know about itself, and you also design the methods that operate on that data. It’s common for an object to have methods that read or write the values of the instance variables.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

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

**Sharpen your pencil**

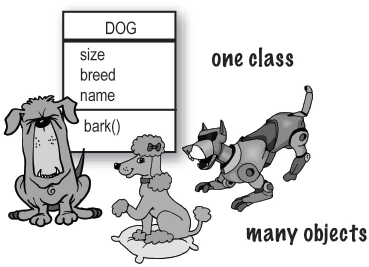
image

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

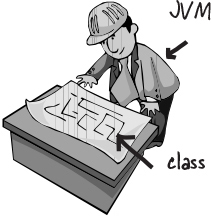


**What’s the difference between a class and an object?**

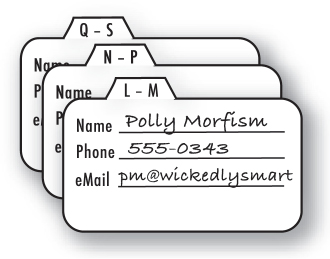
**A class is not an object. (but it’s used to construct them)**



**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 пример 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. Each one of these different buttons would be a button *object*.



**Look at it this way...**



**An object is like one entry in your contacts list.**

One analogy for classes and objects is your phone’s contact list. Each contact has the same blank fields (the instance variables). When you create a new contact, you are creating an instance (object), and the entries Записи you make for that contact represent its state.

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

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

**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.

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** Банджи class, we’ll need a **BungeeTestDrive** class as well. Only the *<SomeClassName>***TestDrive** class will have a main() method, and its sole purpose единственная цель is to create objects of your new class (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 потрясающе clear by the following examples. No, *really.*

**Note**

**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()

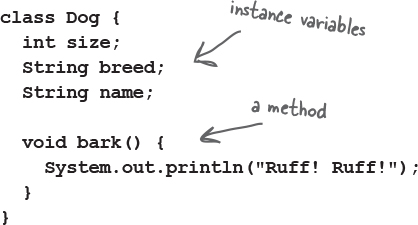
**d.bark();**

// set its size using the

// dot operator

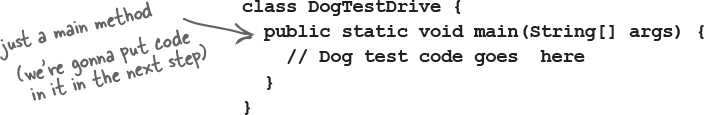
**d.size = 40;**

Images**Write your class**

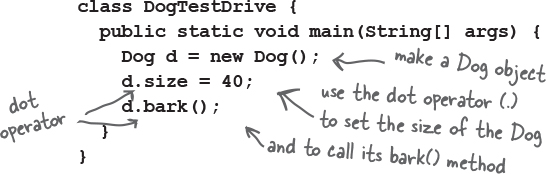




Images**Write a tester (TestDrive) class**



Images**In your tester, make an object and access the object’s variables and methods**



**Note**

If you already have some OO savvy понимание, you’ll know we’re not using encapsulation. We’ll get there in [Chapter 4](#_Chapter_4._Methods).

**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";**

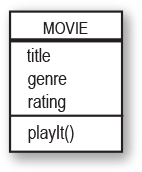
**three.rating = 127;**

**}**

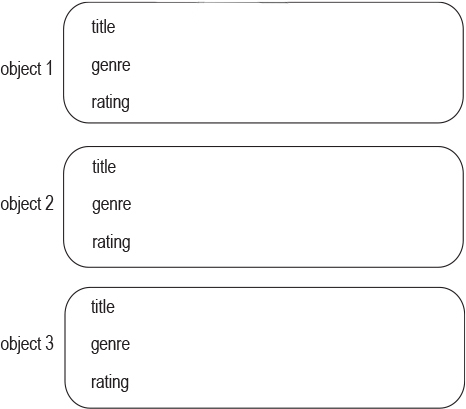
**}**

**Sharpen your pencil**





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 Вызывает (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:**

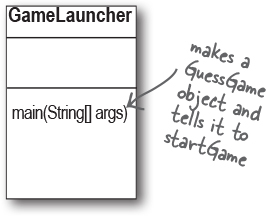
**to test your real class**

**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](#_Chapter_4._Methods) , 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](#_Chapter_6._Get) 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 инструментарий, so you’ll find this program a little clunky Неуклюжим 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**

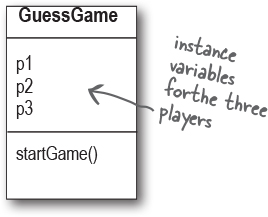


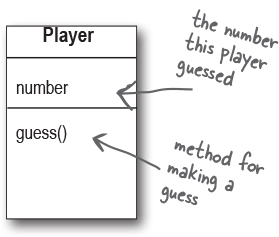
**Summary:**

The guessing game involves Включает 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**



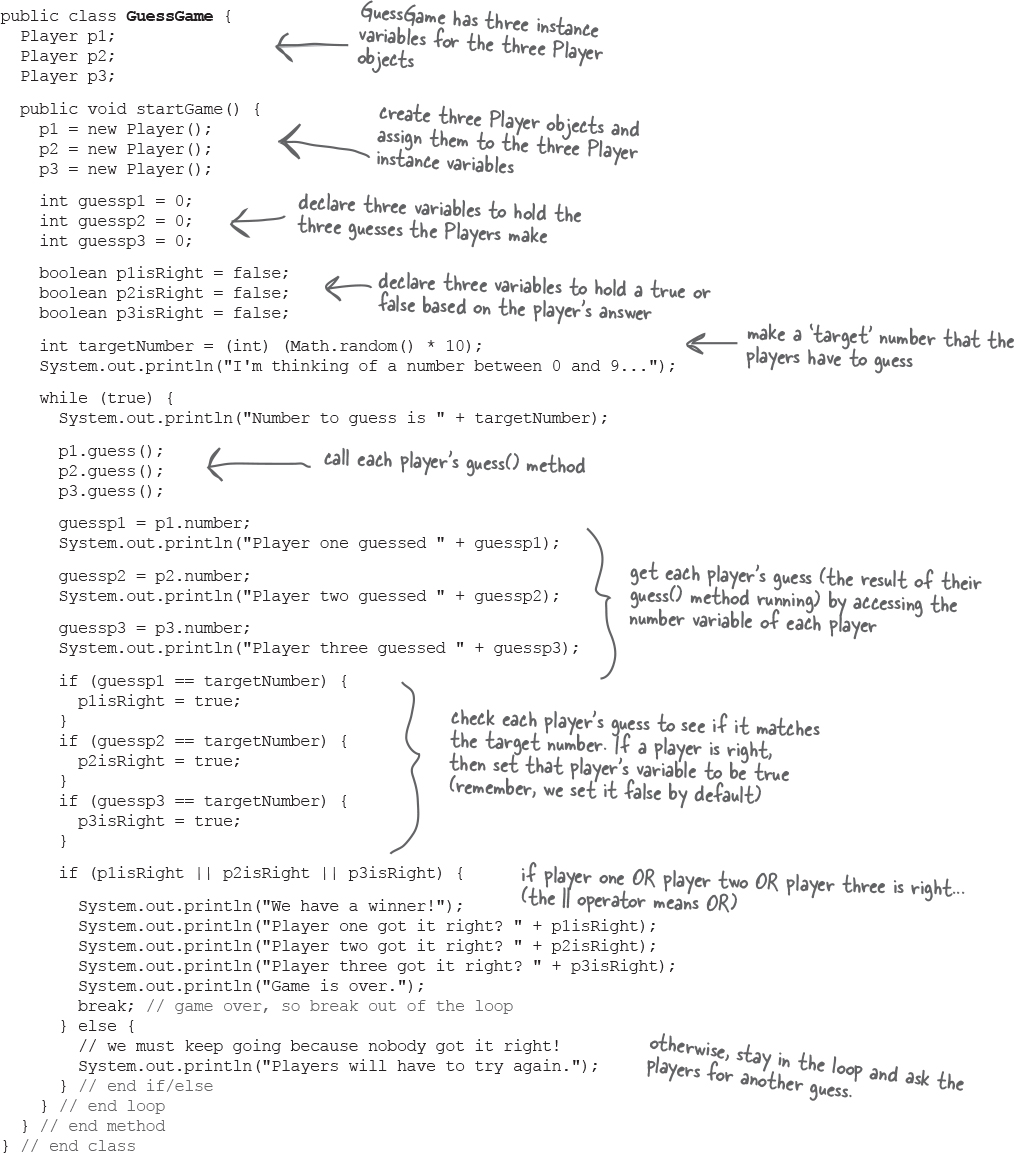


**The Logic:**

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 разыгрывается. 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.



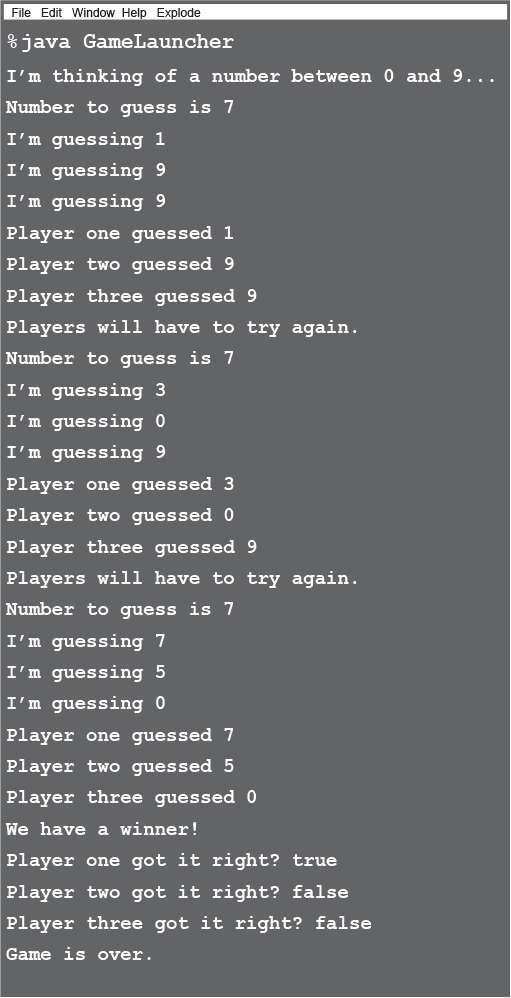
**Running the Guessing Game**

**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 восстанавливать 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 Нехватка памяти, 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](#_Chapter_10._Numbers) 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.

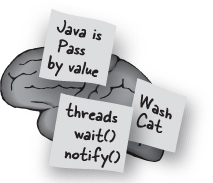
**Q: What** ***is*** **a Java program? What do you actually** ***deliver*?**

**A:** A Java program is a pile куча 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 programs that let you bundle your classes with a JVM and create a folder or file you can share however you want (e.g. via the internet). 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 **J**ava **AR**chive – *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.

**Make it Stick**



**A class is like a recipe. Objects are like cookies.**



**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** переменные экземпляра. 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**



**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 StreamingSong {**

**String title;**

**String artist;**

**int duration;**

**void play() {**

**System.out.println("Playing song");**

**}**

**void printDetails() {**

**System.out.println("This is " + title +**

**" by " + artist);**

**}**

**}**

**class StreamingSongTestDrive {**

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

**song.artist = "The Beatles";**

**song.title = "Come Together";**

**song.play();**

**song.printDetails();**

**}**

**}**

**B**

**class Episode {**

**int seriesNumber;**

**int episodeNumber;**

**void skipIntro() {**

**System.out.println("Skipping intro...");**

**}**

**void skipToNext() {**

**System.out.println("Loading next episode...");**

**}**

**}**

**class EpisodeTestDrive {**

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

**Episode episode = new Episode();**

**episode.seriesNumber = 4;**

**episode.play();**

**episode.skipIntro();**

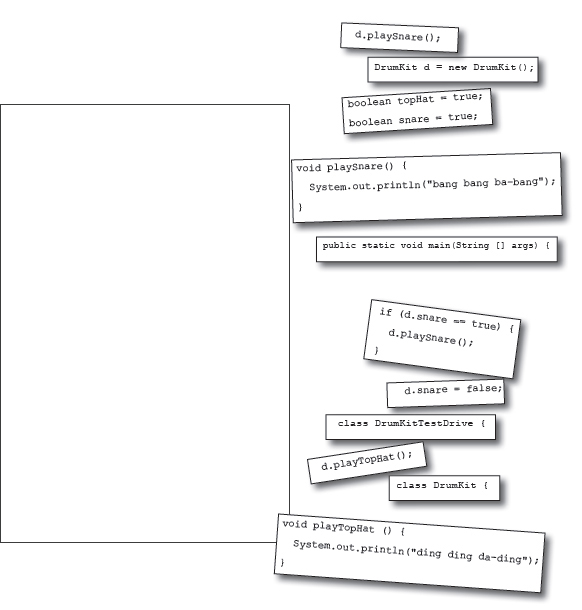
**}**

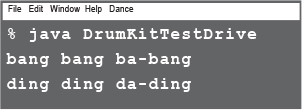
**}**

**Code Magnets**



A Java program is all scrambled зашифрованный up on the fridge. Can you reconstruct the code snippets Фрагменты 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**

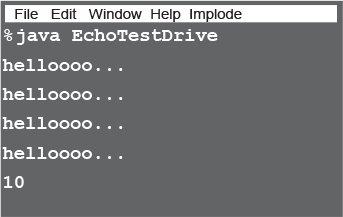




Your ***job*** is to take code snippets Фрагменты from the pool and place them into the blank lines in the code. You **may** use the same snippet more than once, and you won’t need to use all the snippets. Your ***goal*** is to make classes that will compile and run and produce the output listed below.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

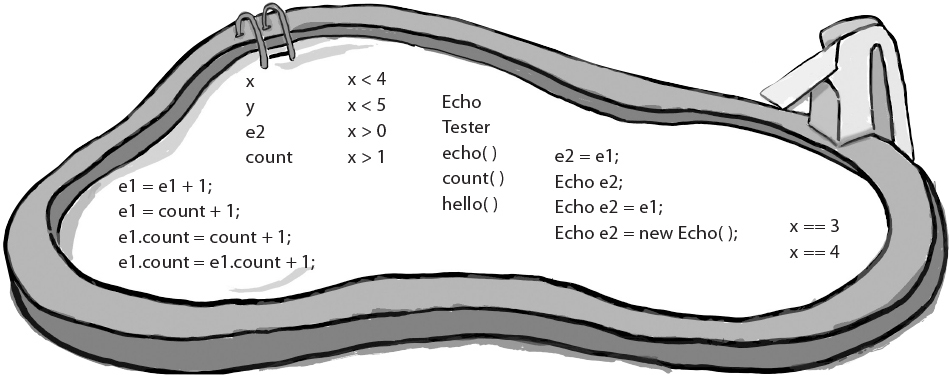
System.out.println("helloooo... ");

}

}

**Note**

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



**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



**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();

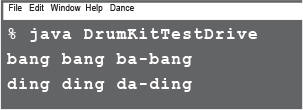
if (d.snare == true) {

d.playSnare();

}

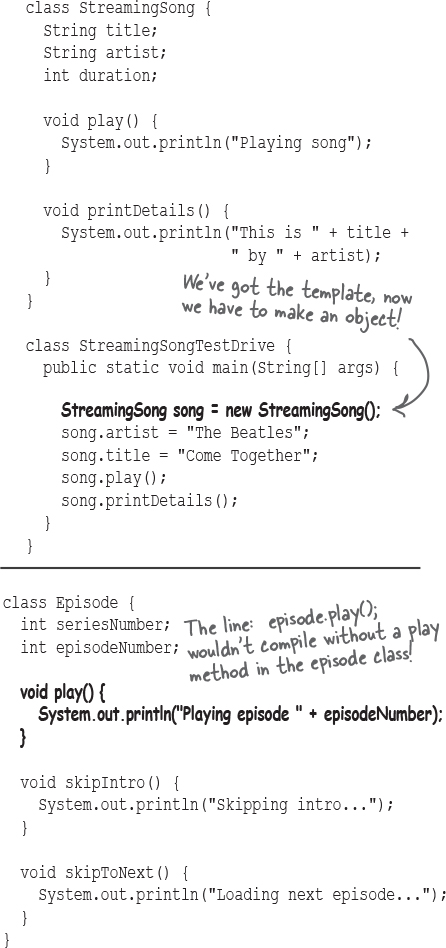
}

}



**Be the Compiler:**

**A**



**B**

class EpisodeTestDrive {

public static void main(String[] args) {

Episode episode = new Episode();

episode.seriesNumber = 4;

episode.play();

episode.skipIntro();

}

}

**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** {

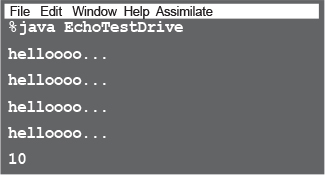
int **count** = 0;

void **hello()** {

System.out.println("helloooo... ");

}

}



**Who am I?**

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

**Note**

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.