A Trip to Objectville

> We're going to Objectville! We're leaving this dusty ol' procedural town for good. I'll send you a postcard.

I was told there would be objects. In chapter 1, we put all of our code in the main() method. That's not exactly object-oriented. In fact, that's not object-oriented *at all*. Well, we did *use* a few objects, like the String arrays for the Phrase-O-Matic, but we didn't actually develop any of our own object *types*. So now we've got to leave that procedural world behind, get the heck out of main(), and start making some objects of our own. We'll look at what makes object-oriented (OO) development in Java so much fun. We'll look at the difference between a *class* and an *object*. We'll look at how objects can give you a better life (at least the programming part of your life. Not much we can do about your fashion sense). Warning: once you get to Objectville, you might never go back. Send us a postcard.
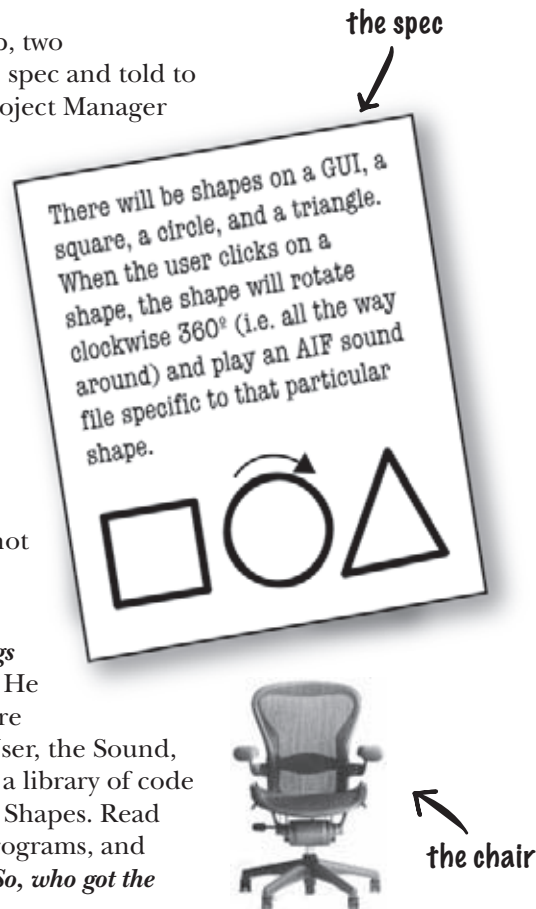
Chair Wars
(or How Objects Can Change Your Life)

*O*nce 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 one of those cool Aeron™ chairs all the Silicon Valley guys have. Larry, the procedural programmer, and Brad, the OO guy, both knew this would be a piece of cake.

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

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

the spec

There will be shapes on a GUI, a square, a circle, and a triangle. When the user clicks on a shape, the shape will rotate clockwise 360º (i.e. all the way around) and play an AIF sound file specific to that particular shape.
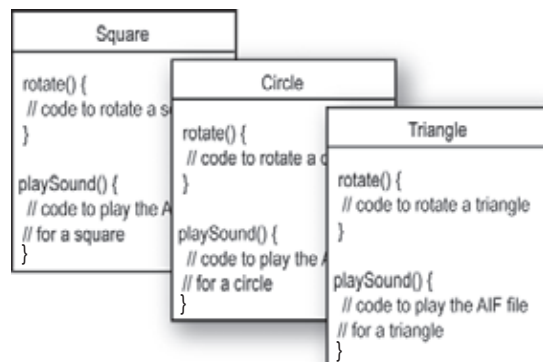
the chair

In Larry's cube

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

```
rotate(shapeNum) {
   // make the shape rotate 360°
}
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

```
Square
rotate() {
  // code to rotate a s
}

playSound() {
  // code to play the A
  // for a square
}
```

```
Circle
rotate() {
  // code to rotate a c
}

playSound() {
  // code to play the A
  // for a circle
}
```

```
Triangle
rotate() {
  // code to rotate a triangle
}

playSound() {
  // code to play the AIF file
  // for a triangle
}
```
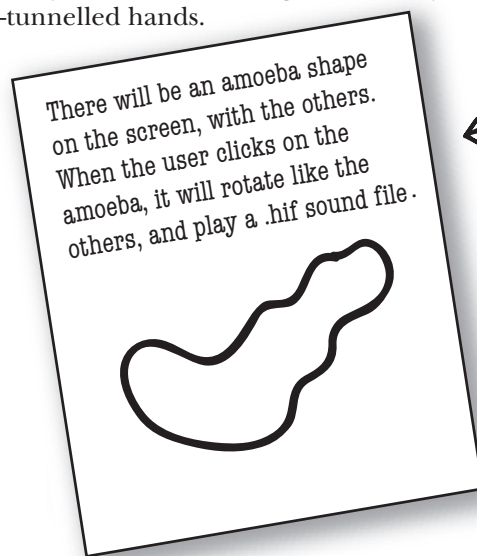
Larry thought he'd nailed it. He could almost feel the rolled steel of the Aeron beneath his...

But wait! There's been a spec change.

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

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

There will be an amoeba shape on the screen, with the others. When the user clicks on the amoeba, it will rotate like the others, and play a .hif sound file.

← **what got added to the spec**

Back in Larry's cube

The rotate procedure would still work; the code used a lookup table to match a shapeNum to an actual shape graphic. But **playSound would have to change.** And what the heck is a .hif file?

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

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

At Brad's laptop at the beach

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

Amoeba

rotate() {
  // code to rotate an amoeba
}

playSound() {
  // code to play the new
  // .hif file for an amoeba
}

Larry snuck in just moments ahead of Brad.

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

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

1) determine the rectangle that surrounds the shape

2) calculate the center of that rectangle, and rotate the shape around that point.

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

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

Ameoba rotation point in Larry and Brad's version:

Where the ameba rotation point should be:
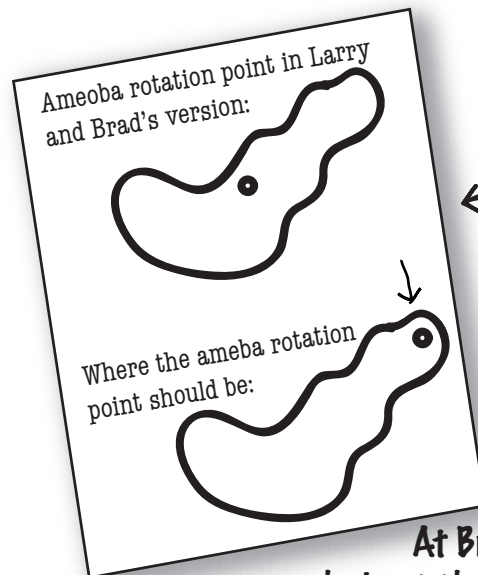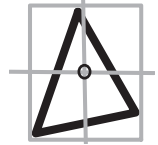
What the spec conveniently forgot to mention

Back in Larry's cube

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

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

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

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

Amoeba
int xPoint
int yPoint
rotate() {
// code to rotate an amoeba
// using amoeba's x and y
}
playSound() {
// code to play the new
// .hif file for an amoeba
}

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

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

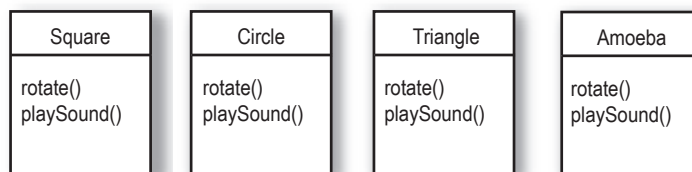**LARRY:** You've got duplicated code! The rotate procedure is in all four Shape things.

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

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

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

What Larry wanted ↰
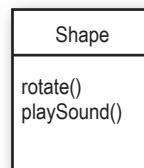**(figured the chair would impress her)**

①

Square
rotate()
playSound()

Circle
rotate()
playSound()

Triangle
rotate()
playSound()

Amoeba
rotate()
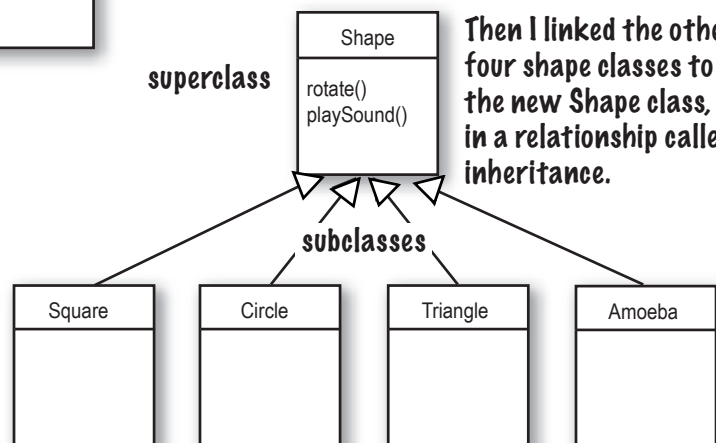playSound()

I looked at what all four classes have in common.

②

They're Shapes, and they all rotate and playSound. So I abstracted out the common features and put them into a new class called Shape.

Shape
rotate()
playSound()

③

Then I linked the other four shape classes to the new Shape class, in a relationship called inheritance.

superclass

Shape
rotate()
playSound()

subclasses

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.*

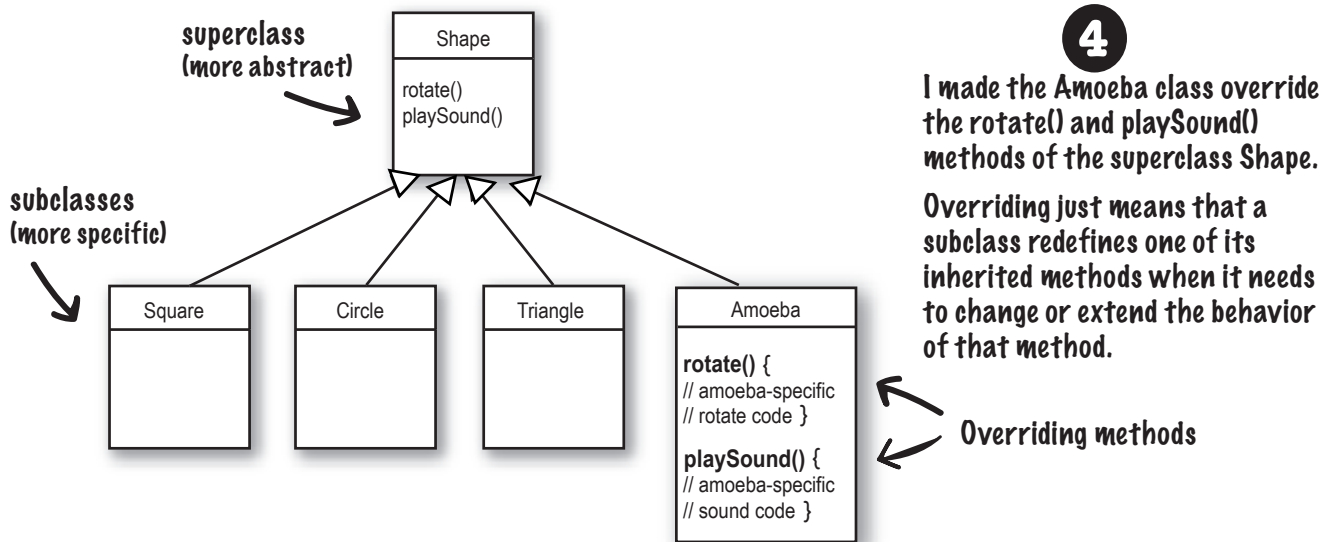| Square		Circle		Triangle		Amoeba

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?
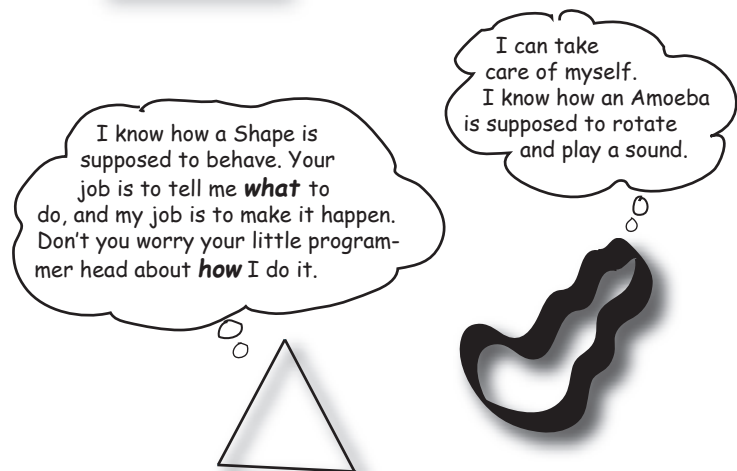
BRAD: Method.

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.

Override Now
Ask Me How

superclass (more abstract)

Shape

rotate()
playSound()

subclasses (more specific)

Square

Circle

Triangle

Amoeba

rotate() {
// amoeba-specific
// rotate code }

playSound() {
// amoeba-specific
// sound code }

4

I made the Amoeba class override the rotate() and playSound() methods of the superclass Shape.

Overriding just means that a subclass redefines one of its inherited methods when it needs to change or extend the behavior of that method.

Overriding methods

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.**

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 oper- ate 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 POWER

Time to pump some neurons.

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

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

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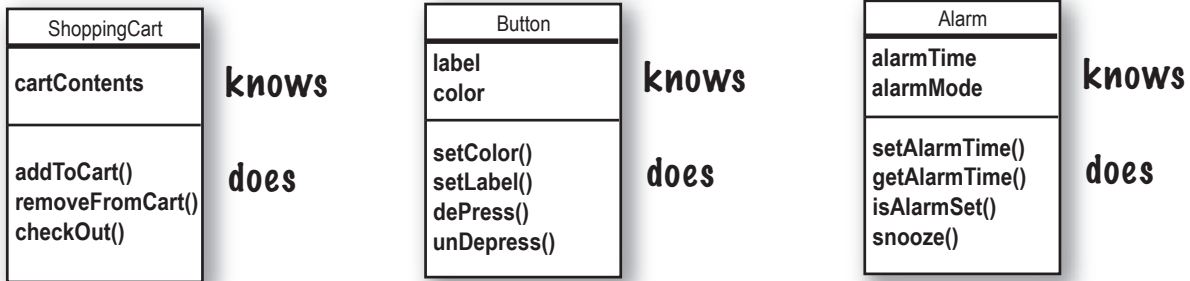
metacognitive tip

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

When you design a class, think about the objects that will be created from that class 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

instance
variables
(state)

methods
(behavior)

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

knows

does

Things an object can *do* are called

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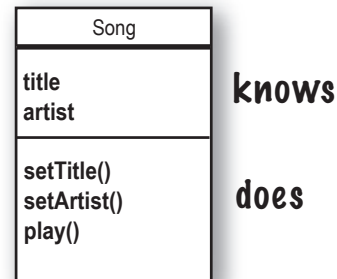
Things an object *knows* about itself are called **instance variables**. They represent an object's state (the data), and can have unique values for each object of that type.

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

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?

one class

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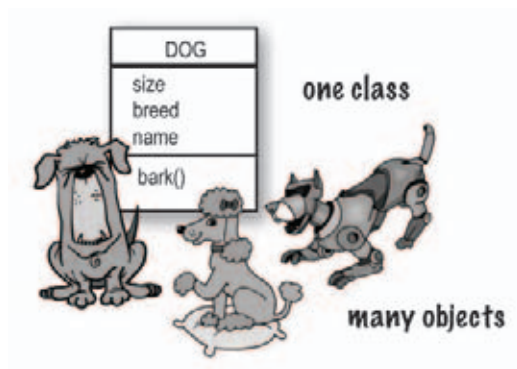
A class is not an object.
(but it's used to construct them)

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.

JVM

class
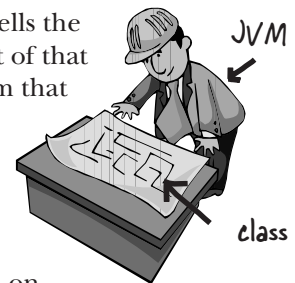
Look at it this way...

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.
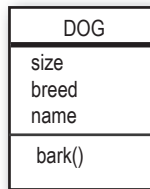

The Dot Operator (.)

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;

---

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**

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

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

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

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


```java
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

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

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.

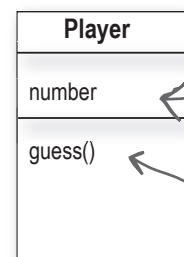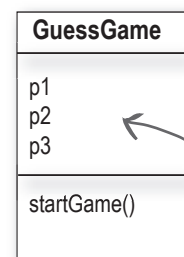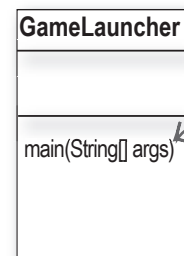## The Guessing Game

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.

GameLauncher

main(String[] args)

makes a GuessGame object and tells it to startGame

GuessGame

p1
p2
p3

startGame()

instance variables for the three players

Player

number

guess()

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 p1isRight = 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) {
          p1isRight = true;
        }
        if (guessp2 == targetNumber) {
          p2isRight = true;
        }
        if (guessp3 == targetNumber) {
          p3isRight = true;
        }

        if (p1isRight || p2isRight || p3isRight) {

          System.out.println("We have a winner!");
          System.out.println("Player one got it right? " + p1isRight);
          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

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

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


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

Java takes out the Garbage

Each time an object is created in Java, it goes into an area of memory known as **The Heap**.

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

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

```
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.

Make it Stick

Java is Pass by value

threads wait() notify()

Wash Cat

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**. They represent the *state* of an object.
- Things an object does are called **methods**. They represent the *behavior* of an object.
- When you create a class, you may also want to create a separate test class which you'll use to create objects of your new class type.
- A class can **inherit** instance variables and methods from a more abstract **superclass**.
- At runtime, a Java program is nothing more than objects 'talking' to other objects.

BE *the* compiler

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

A

```
class TapeDeck {

  boolean canRecord = false;

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

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

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

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

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

    }

  }
}
```

B

```
class DVDPlayer {

  boolean canRecord = false;

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

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

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

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

    }

  }
}
```

Code Magnets

Exercise

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.

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

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

```
boolean topHat = true;

boolean snare = true;
```

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

File Edit Window Help Dance

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

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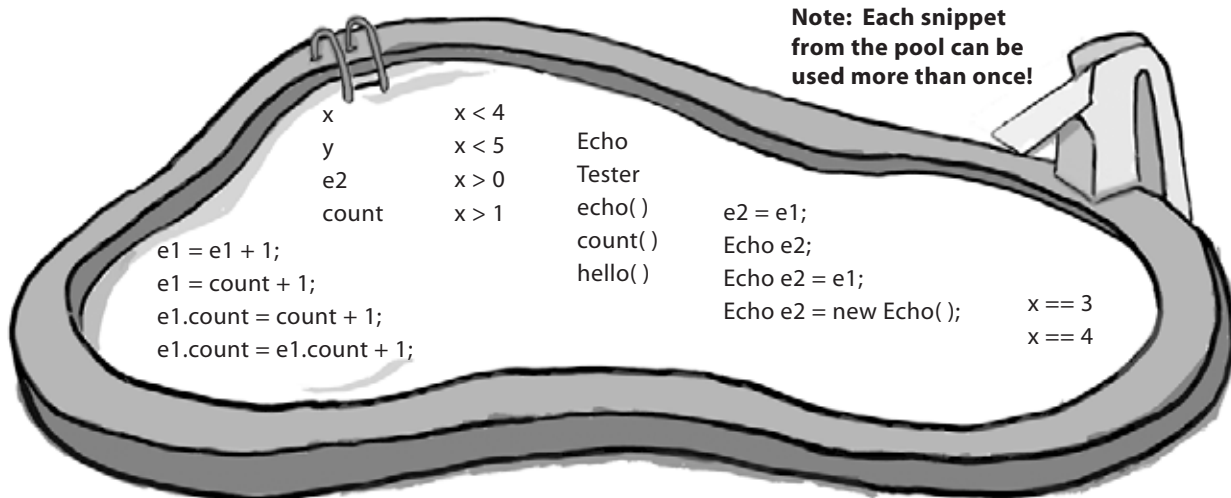
```
class _____ {
  int _____ = 0;
  void _____ {
    System.out.println("helloooo... ");
  }
}
```

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

```
x          x < 4        Echo
y          x < 5        Tester
e2         x > 0        echo( )
count      x > 1        count( )
                        hello( )      e2 = e1;
e1 = e1 + 1;                          Echo e2;
e1 = count + 1;                       Echo e2 = e1;
e1.count = count + 1;                 Echo e2 = new Echo( );    x == 3
e1.count = e1.count + 1;                                        x == 4
```

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

Exercise Solutions

Code Magnets:

```java
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
```java
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 !

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

B
```java
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 !

Puzzle Solutions

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... ");
  }
}
```

Who am I?

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

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

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