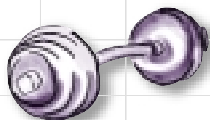
Your Brain on Java—A Learner's Guide

Head First Java



Pump neurons with the Brain Barbell workouts

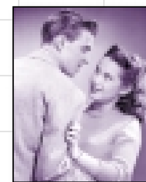
Gather your clues to solve a Five-Minute Java Mystery

See what makes the JVM tick and what ticks it off

Watch Java objects expose their inner secrets on Java Tabloid TV

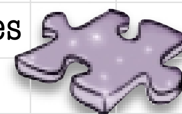
Whip up some code with Ready-Bake Java

Fool around in the Java Library



Learn why Lucy *really* keeps her variables private

Bend your mind around 42 Java puzzles



Kathy Sierra & Bert Bates

A trip to Objectville

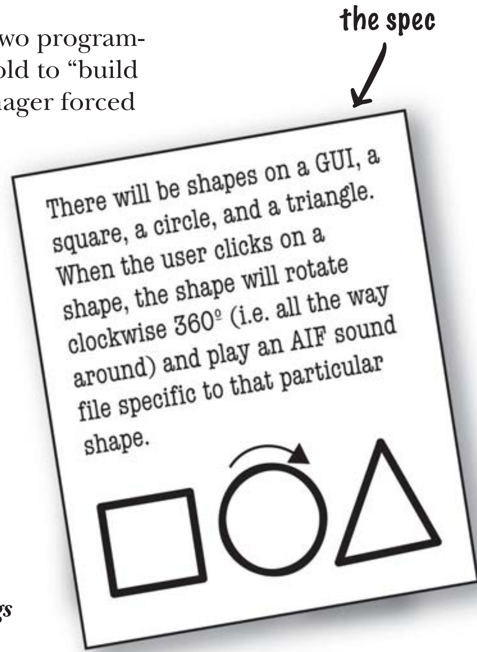
I was told there would be objects. In Chapter 1, we put all of our code in the main() method. That's not exactly object-oriented. In fact, that's not object-oriented *at all*. Well, we did *use* a few objects, like the String arrays for the Phrase-O-Matic, but we didn't actually develop any of our own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

Chair Wars

(or How Objects Can Change Your Life)

Once upon a time in a software shop, two programmers were given the same spec and told to "build it". The Really Annoying Project Manager forced the two coders to compete, and whoever delivered first got one of those cool Aeron™ chairs all the Silicon Valley guys had. Larry, the procedural programmer, and Brad, the OO guy, both knew this would be a piece of cake.

the spec



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 has 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 chair

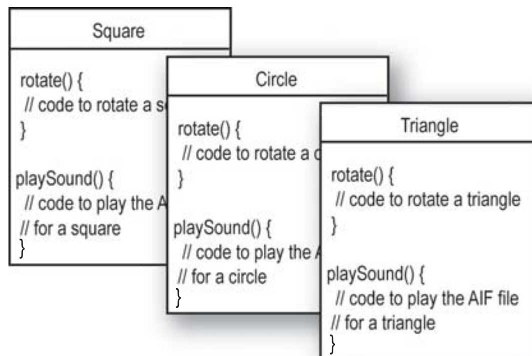
In Larry's cube

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

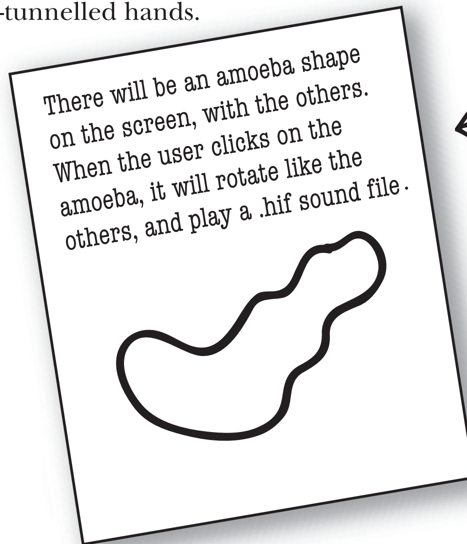


Larry thought he'd nailed it. He could almost feel the rolled steel of the Aeron beneath his...

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.



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

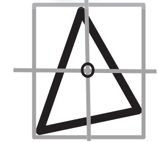
Brad smiled, sipped his margarita, and wrote one new class. Sometimes the thing he loved most about OO was that he didn't have to touch code he'd already tested and delivered. "Flexibility, extensibility,..." he mused, reflecting on the benefits of OO.

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 slid away when the Really Annoying Project Manager said (with that tone of disappointment), "Oh, no, that's not how the amoeba is supposed to rotate..."



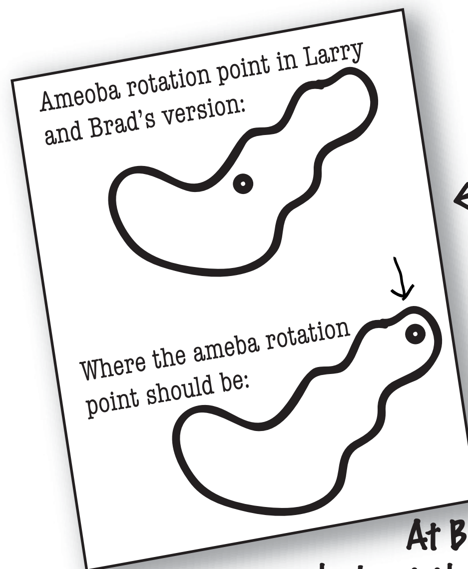
Turns out, both programmers had written their rotate code like this:

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 sourdough. "Although, hmmmm. I could just add another if/else to the rotate, and 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?"



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 Ameboas would have. He modified, tested, and delivered (wirelessly) the revised program during a single Bela Fleck song.

```
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.

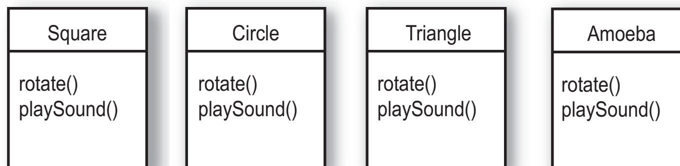
LARRY: Whatever. It's a stupid design. You have to maintain four different rotate "methods". How can that ever be good?

BRAD: Oh, I guess you didn't see the final design. Let me show you how OO **inheritance** works, Larry. Even *you* should be able to follow along.

What Larry wanted
(figured the chair would impress her)

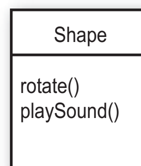
1

I looked at what all four classes have in common.



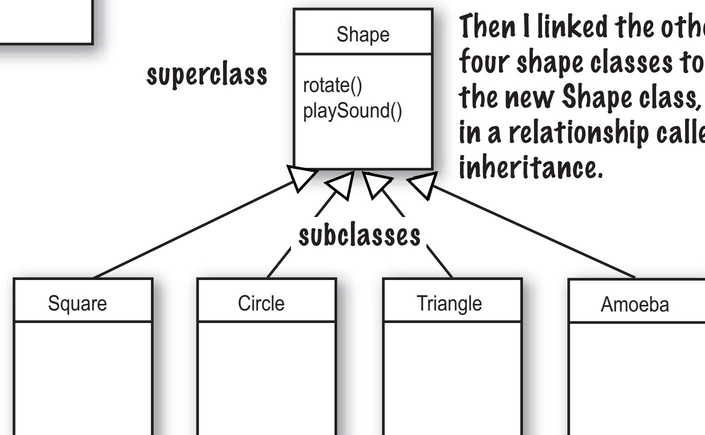
2

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.



3

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



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

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

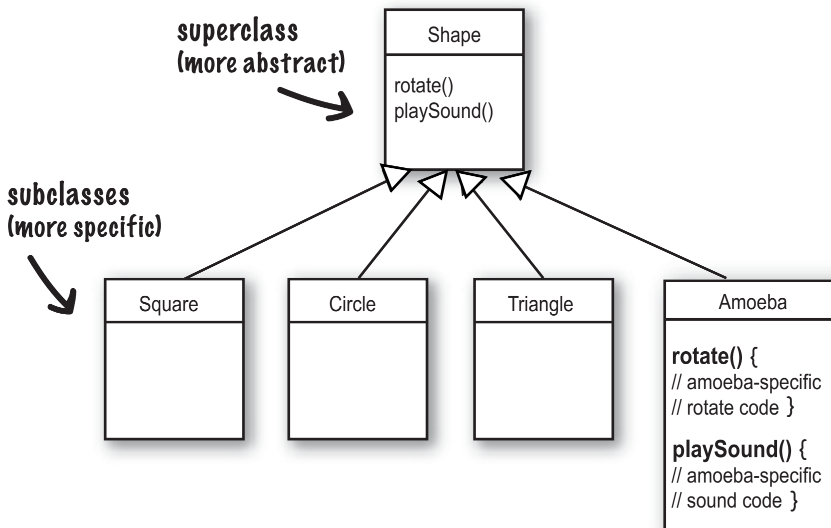
What about the Amoeba rotate()?

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

BRAD: Method.

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

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



4

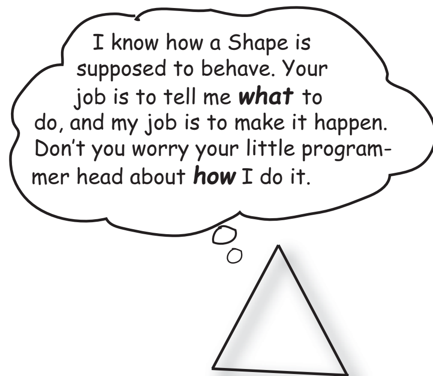
I made the Amoeba class override the rotate() and playSound() methods of the superclass Shape.

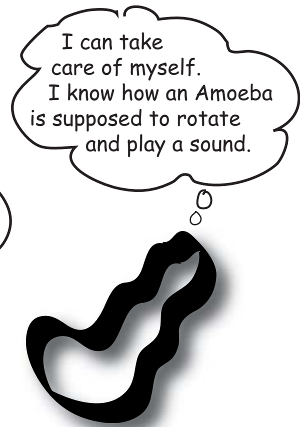
Overriding just means that a subclass redefines one of its inherited methods when it needs to change or extend the behavior of that method.

LARRY: How do you "tell" an Amoeba to do something? Don't you have to call the procedure, OK 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.





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

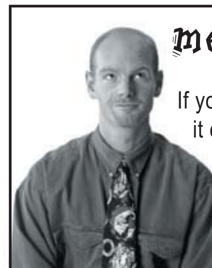
Brain Barbell

Time to pump some neurons.

You just read a story 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 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 it?



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.