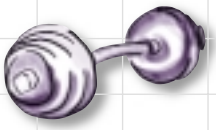
Your Brain on Java—A Learner's Guide

Head First Java



Pump neurons with the Brain Barbell workouts

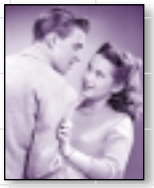
Gather your clues to solve a Five-Minute Java Mystery

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

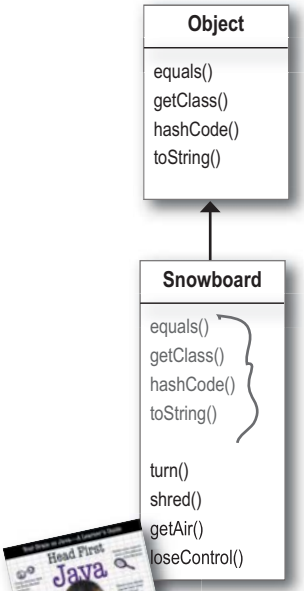
Kathy Sierra & Bert Bates

Serious Polymorphism

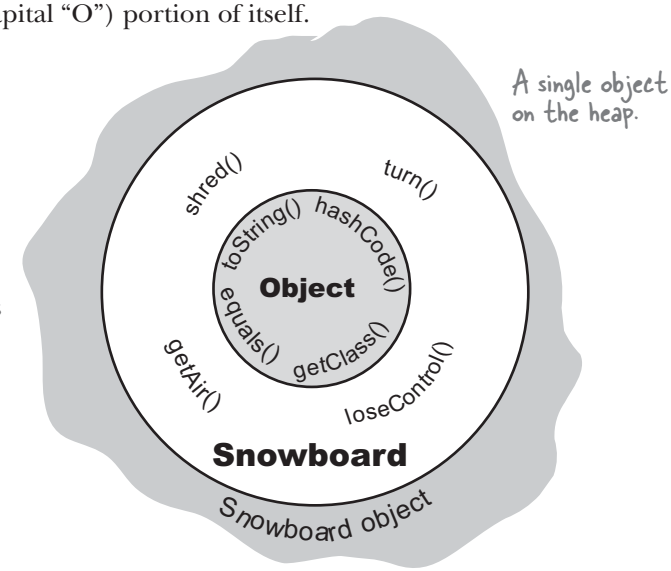
Inheritance is just the beginning. To exploit polymorphism, we need interfaces (and not the GUI kind). We need to go beyond simple inheritance to a level of flexibility and extensibility you can get only by designing and coding to interface specifications. Some of the coolest parts of Java wouldn't even be possible without interfaces, so even if you don't design with them yourself, you still have to use them. But you'll *want* to design with them. You'll *need* to design with them. ***You'll wonder how you ever lived without them***. What's an interface? It's a 100% abstract class. What's an abstract class? It's a class that can't be instantiated. What's that good for? You'll see in just a few moments. But if you think about the end of the last chapter, and how we used polymorphic arguments so that a single Vet method could take Animal subclasses of all types, well, that was just scratching the surface. Interfaces are the ***poly*** in polymorphism. The ***ab*** in abstract. The ***caffeine*** in Java.

Get in touch with your inner Object.

An object contains *everything* it inherits from each of its superclasses. That means *every* object—regardless of its actual class type—is *also* an instance of class Object.That means any object in Java can be treated not just as a Dog, Button, or Snowboard, but also as an Object. When you say **`new Snowboard()`**, you get a single object on the heap—a Snowboard obect—but that Snowboard wraps itself around an inner core representing the Object (capital "O") portion of itself.



Snowboard inherits methods from superclass Object, and adds four more.



There is only ONE object on the heap here. A Snowboard object. But it contains both the Snowboard class parts of itself and the Object class parts of itself.

‘Polymorphism’ means ‘many forms’.

You can treat a Snowboard as a Snowboard or an Object.

If a reference is like a remote control, the remote control takes on more and more buttons as you move down the inheritance tree. A remote control (reference) of type Object has only a few buttons—the buttons for the exposed methods of class Object. But a remote control of type Snowboard includes all the buttons from class Object, plus any new buttons (for new methods) of class Snowboard. The more specific the class, the more buttons it may have.

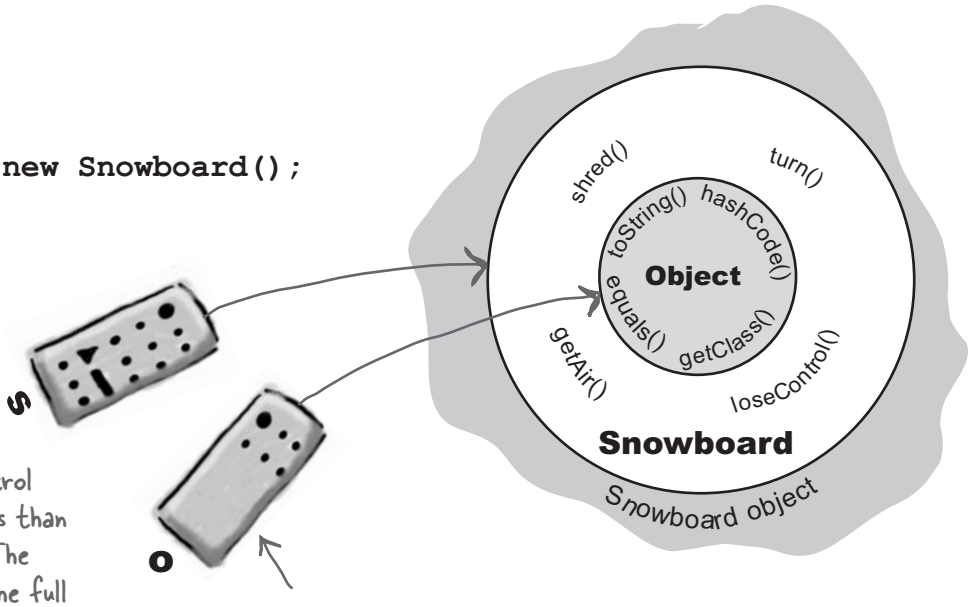
Of course that’s not always true; a subclass might not add any new methods, but simply override the methods of its superclass. The key point is that even if the *object* is of type Snowboard, an Object *reference* to it can’t see the Snowboard-specific methods.

When you put an object in an ArrayList, it forgets (temporarily) its true type, and thinks of itself as an Object.

When you get a reference from an ArrayList, the reference is always of type *Object*.

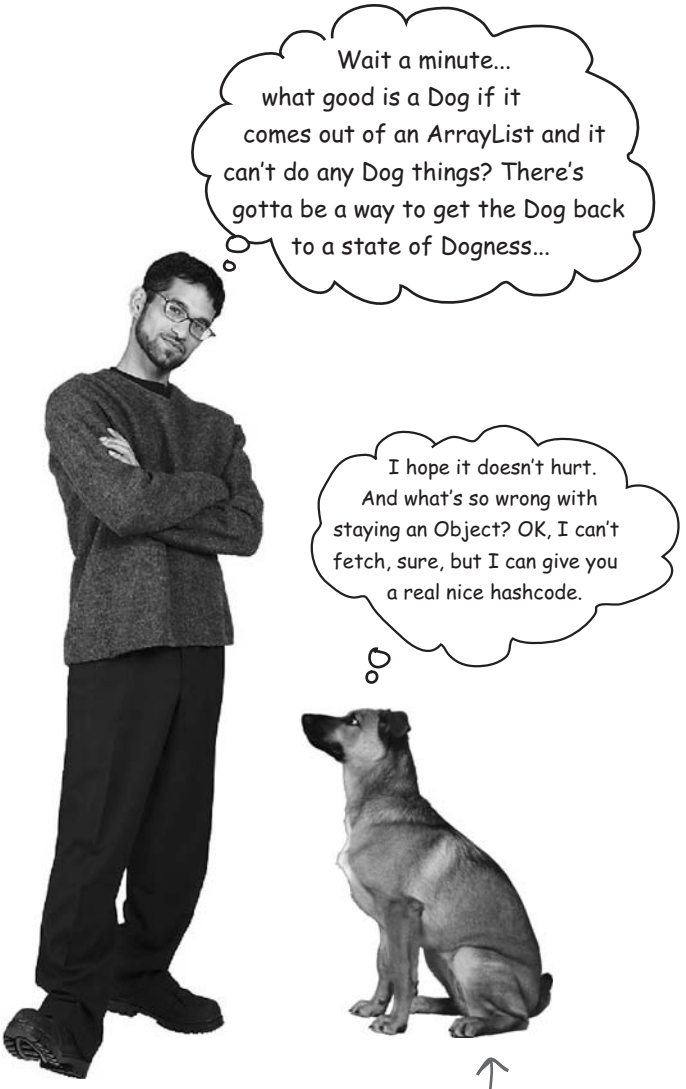
That means you get an *Object* remote control.

```
Snowboard s = new Snowboard();
Object o = s;
```

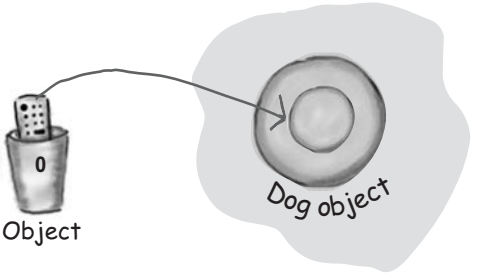


The Snowboard remote control (reference) has more buttons than an Object remote control. The snowboard remote can see the full Snowboardness of the Snowboard object. It can access all the methods in Snowboard, including both the inherited Object methods and the methods from class Snowboard.

The Object reference can see only the Object parts of the Snowboard object. It can access only the methods of class Object. It has fewer buttons than the Snowboard remote control.



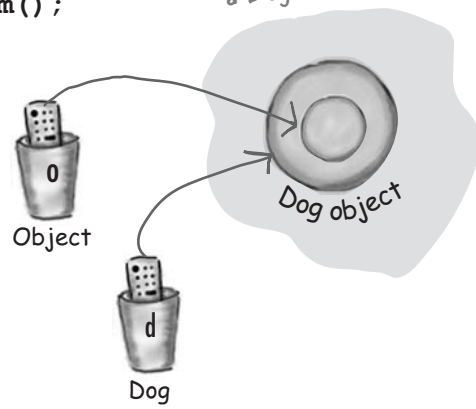
Casting an object reference back to its *real* type.



It's really still a Dog *object*, but if you want to call Dog-specific methods, you need a *reference* declared as type Dog. If you're *sure** the object is really a Dog, you can make a new Dog reference to it by copying the Object reference, and forcing that copy to go into a Dog reference varaiable, using a cast (Dog). You can use the new *Dog* reference to call *Dog* methods.

```
Object o = al.get(index);
Dog d = (Dog) o;
d.roam();
```

cast the Object back to a Dog we know is there.



*If you're *not* sure it's a Dog, you can use the **instanceof** operator to check. Because if you're wrong, you'll get a ClassCastException and come to a grinding halt.

```
if (d instanceof Dog) {
   Dog d = (Dog) o;
}
```

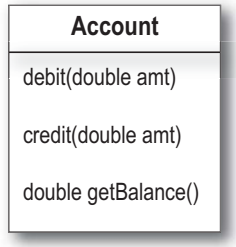
So now you've seen how much Java cares about the methods in the class of the reference variable.

You can call a method on an object *only* if the class of the reference variable has that method.

Think of the public methods in your class as your contract, your promise to the outside world about the things you can do.

When you write a class, you almost always *expose* some of the methods to code outside the class. To *expose* a method means you make a method *accessible*, usually by marking it public.

Imagine this scenario: you're writing code for a small business accounting program. A custom application for "Simon's Surf Shop". The good re-user that you are, you found an Account class that appears to meet your needs perfectly, according to its documentation, anyway. Each account instance represents an individual customer's account with the store. So there you are minding your own business invoking the *credit()* and *debit()* methods on an account object when you realize you need to get a balance on an account. No problem—there's a *getBalance()* method that should do nicely.



Except... when you invoke the *getBalance()* method, the whole thing blows up at runtime. Forget the documentation, the class does not have that method. Yikes!

But that won't happen to you, because everytime you use the dot operator on a reference (a.doStuff()), the compiler looks at the *reference* type (the type 'a' was declared to be) and checks that class to guarantee it has the method, and that it indeed takes the argument you're passing and returns the kind of value you're expecting to get back.

Just remember that it checks the class of the *reference* variable, not the class of the actual *object* at the other end of the reference.