SUBCLASSES IN JAVA

Every class in Java is built from another Java class. The new class is called a subclass of the other class from which it is built. A subclass inherits all the instance methods from its superclass. The notion of being a subclass is transitive: If class *A* is a subclass of *B*, and *B* is a subclass of *C*, then *A* is also considered a subclass of *C*. And if *C* is a subclass of *D*, then so is *A* a subclass of *D*.

The subclass concept has important implications in Java, and we explore the concept in this chapter.

Fundamentals of subclasses

The `GOval` class is a good example of a subclass: It is a subclass of the `GObject` class. The `GObject` class represents abstract objects that might appear inside a graphics window, and a `GOval` object is a particular shape that will appear.

Since all such objects will have a position in the window, `GObject` defines several methods regarding the object's position, including `getX`, `getY`, and `move`. The `GOval` class, as a subclass of `GObject`, inherits all of these methods, as well as adding some of its own, like `setFilled`. The `GLine` class, also a subclass of `GObject`, inherits `GObject`'s methods too, and it adds different methods, like `setStartPoint` and `setEndPoint` for moving the line's endpoints. (The `GLine` class does not have a method called `setFilled`.)

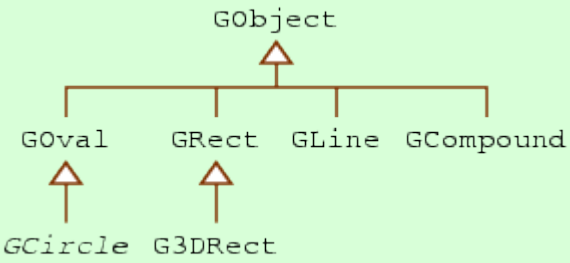
Subclasses are meant to be more specialized versions of the superclass — hence the word *subclass*, similar to the word *subset* from mathematics. Ovals are a subset of all shapes, so `GOval` is defined as a subclass of `GObject`. Being more specialized, it may make sense for the subclass to be perform specialized methods that don't apply to the more general superclass. The method `setFilled` doesn't make sense for `GObject`, because for some shapes (such as lines), the notion of being filled is senseless. But for ovals, it does make sense, and so `GOval` defines a `setFilled` method.

Often we say that the new class extends the other class, since it contains all of the instance methods of the superclass, plus possibly more that are defined just for the subclass. In fact, our programs have included exactly this word *extends*: We've started the program with words like `public class...extends GraphicsProgram`.)

If we wanted a `GCircle` class for representing circles, then a good designer would define it as a subclass of `GOval`, since circles are simply a special type of oval. This class might add some additional methods, such as `getRadius`, that don't make as much sense in the context of ovals. (The designers of the `acm.graphics` package didn't feel that circles were sufficiently interesting to include a special class for them, though.) By the way, `GCircle` would automatically be a subclass of `GObject`, too, since every circle is an oval, and every oval is a shape.

Much like a family tree, classes can be arranged into a diagram called an inheritance hierarchy, showing they relate to each other. Figure 11.1 illustrates an inheritance hierarchy for several classes in the `acm.graphics` package, plus our hypothetical `GCircle` class.

Figure 11.1: An inheritance hierarchy. (The italicized `GCircle` is not in `acm.graphics`.)



The Object class

This chapter began: Every class in Java is built from another Java class. This leads to an obvious question: Does this mean that there are infinitely many classes, each extending the next one?

Actually, the sentence wasn't entirely true. There is one special class which does not extend any other class: `Object`, found in the `java.lang` package. When defining a class that doesn't seem sensibly to extend any other class (as often happens), a developer would define it to extend the `Object` class alone. The `GObject` class is an example of a class whose only superclass is `Object`.

The `Object` class does include a few methods. Because `Object` is the ultimate parent of all other classes, all other classes inherit these methods. In this book, we'll examine two of these methods, `equals` and `toString`.

```
boolean equals(Object other)
```

> Returns `true` if this object is identical to `other`. By default, the two objects are regarded as equal only if they are the same object. For some classes (notably `String`), `equals`compares the data within the object to check whether the objects represent the same concept, even if they are two separate objects.

```
String toString()
```

> Returns a string representation of this object's value. By default, this string is basically nonsense, but for some subclasses the method returns a meaningful string representation.

These instance methods can be applied to any object in a program, because every object is a member of some class, and that class must lie somewhere below `Object` class in the inheritance hierarchy, so it will inherit the `Object` instance methods. Thus, writing `ball.toString()` would be legal, no matter what sort of object `ball` references.

Source : http://www.toves.org/books/java/ch11-subclass/index.html