Programming via Java Subclasses

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.

11.1. 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`.)

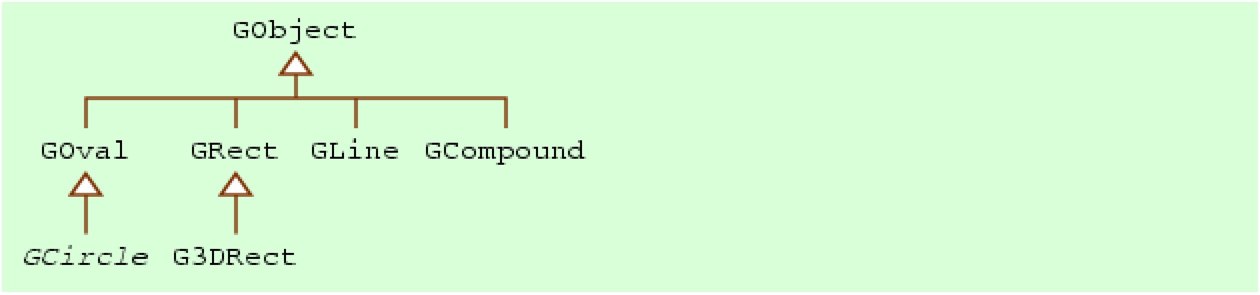
Subclasses are meant to be more specialized versions of the superclass — hence the word *subclass*, similar to the word *subset* from mathematics. Ovals are a subset of all shapes, so `GOval` is defined as a subclass of `GObject`. Being more specialized, it may make sense for the subclass to be perform 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`.)



11.2. 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.

11.3. Subclasses and variables

Any instance of a class *A* can also be treated as an instance of a superclass of *A*. Thus, if *B* is a superclass of *A*, then every *A* object can also be treated as a *B* objects. As a result, if a variable's type is a class *C*, then the variable can reference an object whose actual class is some subclass of *C*. For example, we could rewrite our `MovingBall`class so that `ball` is a `GObject` variable.

```
GObject ball;                       // Note that ball is a GObject
ball = new GOval(10, 10, 100, 100); // but we assign a GOval to it.
add(ball);

while(true) {
    pause(40);
    ball.move(1, 0);
}
```

The `ball` variable is declared to reference any `GObject`, including instances of subclasses of `GObject`. So in the second line, initializing `ball` to reference a `GOval` is acceptable. We could also make `ball` refer to a `GRect` if we liked, since `GRect` is also a subclass of `GObject`.

Later in the fragment, the line `ball.move(1, 0);` is acceptable, because the `move` method is defined in the `GObject` class. (The compiler knows that this method will be inherited by any subclass, so it's assured that even if `ball`'s actual class is some subclass, `move` will still be a legal method.) On the other hand, a program cannot invoke a method not defined for the class that the variable represents, as attempted below.

```
GObject ball;
ball = new GOval(75, 75, 50, 50);
ball.setFilled(true);         // Illegal!!
```

Should this fragment execute, `ball` will of course refer to a `GOval`, and one might think that invoking `setFilled` should be legal. But the compiler translates the program before execution, and the compiler looks only at the statement in question to determine whether a method invocation is legal. Looking at the `setFilled` invocation, the compiler knows only that `ball` references a `GObject` object; it doesn't look at the rest of the program to deduce the obvious fact that `ball` will actually be a `GOval`. Since not all `GObject` objects have a `setFilled` method, the compiler will reject the line and the program.

You ought to be wondering: Why would I ever want a variable anyway whose type is a superclass of the object that it will actually reference? After all, using the superclass simply restricts what we can do with the object. Most often, this reasoning is correct: The program might as well use the most specific type available. But there are some circumstances where using a more general type is quite useful. A notable example is the `add` method in the `GraphicsProgram` class.

```
void add(GObject shape)
```

> Adds `shape` for drawing within this window. If `shape`'s class is written properly, any modifications to `shape` will be reflected immediately in this window.

Using the above reasoning concerning variables, the `add` method's parameter `shape` can refer to an instance of any `GObject` subclass. Thus, this single `add` method can deal with inserting a `GOval`, a `GRect`, or any other subclass of `GObject`. For the person writing `GraphicsProgram`, writing `add` just once for all `GObject`s is much more convenient than requiring an `add` method for each possible category of shape. In fact, we can even define our own subclasses of `GObject` (as we'll do in Chapter 14), and the `add` method will automatically apply to them, too, with no need to modify `GraphicsProgram`.

11.4. Example: Psychedelic balls

To illustrate an instance where having a `GObject` variable is useful, let's consider the following goal: Suppose we want a window containing fifty circles, and we want them all to switch colors every quarter second.

The obvious way to do it, based on what we've seen so far, is to create fifty different variables, one for each circle. With each quarter second, we'd change the color for the circle to which each variable refers. This way is quite cumbersome, and it involves lots of duplication of code, which is always something we should strive to avoid, since duplication inhibits debugging and enhancing programs.

A much simpler alternative is to use the following two methods from the `GraphicsProgram` class.

`GObject getElement(int index)`

> Returns a `GObject` that has been added into the window. The objects are numbered consecutively starting from 0.

`int getElementCount()`

> Returns the number of `GObject`s that are currently in this window.

These methods are useful because they relieve us from the problem of having to remember all the shapes within our program: We can instead just ask the`GraphicsProgram` object to get each shape for us. Note that `getElement` returns a `GObject`; this is because a particular shape in the window can be of any subclass of`GObject`.

Figure 11.2 contains a program accomplishing our task using this technique.

Figure 11.2: The `PsychedelicBalls` program.

```java
import acm.program.*;
import acm.graphics.*;
import java.awt.*;

public class PsychedelicBalls extends GraphicsProgram {
    public void run() {
        // Add all the circles into the window
        Color cur = Color.BLUE; // set up the colors to switch between
        Color prev = Color.RED;
        for(int row = 0; row < 5; row++) {
            for(int col = 0; col < 10; col++) {
                double x = (col + 0.5) * getWidth() / 10.0;
                double y = (row + 0.5) * getHeight() / 5.0;
                GOval ball = new GOval(x - 25, y - 25, 50, 50);
                ball.setFilled(true);
                ball.setColor(cur);
                add(ball);
            }
        }

21         // Now repeatedly change out the colors
22         while(true) {
23             pause(250); // wait a quarter second
24 
25             Color temp = cur; // swap colors for this frame
26             cur = prev;
27             prev = temp;
28 
29             for(int i = 0; i < getElementCount(); i++) {
30                 GObject shape = getElement(i);
31                 shape.setColor(cur);
32             }
33         }
34     }
35 }
```

The program has two major sections. The first, lines 7 to 19, runs very quickly at the beginning to add 5 rows of 10 balls each into the window. The computer will spend the remainder of its time in the second part, lines 7 to 19, where it repeatedly waits for a quarter second, swaps into the color for the next frame, and then iterates through all the shapes in the window. The important part for us is line 30, which retrieves each individual ball using the `getElement` method.

Note line 31. It invokes `setColor` on each circle, rather than `setFillColor` as we've been using in programs until now. It would actually be illegal to invoke `setFillColor`here, since `shape` is a `GObject` variable, and `setFillColor` isn't defined in the `GObject` class. However, `GObject` *does* define `setColor`, so the program is legal as written. (The difference between `setColor` and `setFillColor` is that `setColor` also changes the color of the shape's boundary. However, if `setFillColor` has been applied to the shape, then `setColor` does not affect the shape's interior.)

11.5. Casting

But what if we *want* to use `setFillColor` in line 31 of Figure 11.2? It seems like this should be legal: After all, we already know that all the shapes in the window are`GOval` objects, and `GOval` has a `setFillColor` method.

In fact, Java does provide a way to accomplish this, using casting. This technique casts an object into the mold of a subclass, so that you can then invoke the desired method on it. To tell the computer to cast an object, you enclose the name of the class to which it should be casted in parentheses before the variable name. For example, we will replace the line `shape.setColor(cur);` with the following instead.

```
GOval ball = (GOval) shape; // the cast is on the right side of '='
ball.setFillColor(cur);
```

With each shape then, the cast will force `shape` into being treated as a `GOval`. Our program can assign this into a `GOval` variable such as `ball`, and then we can apply`setFillColor` to this `GOval` variable.

In fact, since the `ball` variable is used only once, we can combine the lines together. However, casting is a lower order of precedence than the period operator (the method application operator), just as addition is a lower order of precedence than multiplication in the order of operations. Thus, if we combine the lines, we'll need to enclose the cast in an extra set of parentheses.

```
((GOval) shape).setFillColor(cur);
```

If wrote `(GOval) shape.setFillColor(cur);`, omitting the parentheses, then the compiler would interpret this as if we had written `(GOval) (shape.setFillColor(cur));`. That is, the compiler would think we mean for the computer to invoke `setFillColor` on `shape` first and then to cast whatever`setFillColor` returns into a `GOval` object. The compiler will reject the line, because `shape`, as a `GObject`, does not provide a `setFillColor` method.

In fact, we can abbreviate the loop body even further, because the `shape` variable is also used only once. Thus, the entire loop body can fit into one line.

```
((GOval) getElement(i)).setFillColor(cur);
```

For any of these alternative formulations, it's`worth wondering: What if one of the shapes in the window isn't a `GOval`? For example, what would happen if we happened to have a `GRect` somewhere in the picture? In this case, the program would still compile properly. But when the program executes, the computer would soon get to the line where `getElement` returns the `GRect`, and it is supposed to cast that `GRect` into a `GOval`. At this point, the computer will terminate the program with an exception called a `ClassCastException`.

11.6. The `instanceof` operator

Sometimes you want to be able to test whether a variable references an object of a particular class. Returning to our `PsychedelicBalls` example of Figure 11.2, suppose our picture also included some `GRect`s, which we did *not* want to flicker between colors. To accomplish this, it would be helpful if, as we iterate through all the shapes of the window in lines 29 to 32, we could test which of the shapes are `GOval`s and which are `GRect`s.

The solution is the **`instanceof`** operator. On the left side of this operator is some object, and on its right side is the name of some class. Upon reaching the operator, the computer determines whether the left-side object is an instance of the right-side class, and it substitutes **`true`** or **`false`** accordingly. We can thus place this operator inside an **`if`** statement to determine whether to attempt the cast and invocation of `setFillColor`.

```
GObject shape = getElement(i);
if(shape instanceof GOval) {
    ((GOval) shape).setFillColor(cur);
}
```

With this modification, the `setFillColor` line will be attempted only for the `GOval` objects. This will avoid the potential for the `ClassCastException` that would otherwise occur if the cast within the line were attempted for `GRect`s or `GLine`s.

Of course, if the left-side object is an instance of some subclass (or subsubclass, ...) of the right-side class, it would still be considered an instance of the right-side class. Thus, if the window contained an instance of our hypothetical `GCircle` class, it too would flicker between colors.

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