Programming via Java Defining classes

Our programs so far have used classes, like `Turtle` and `GOval`, which were written by other people. In writing larger programs, we often find that another class would be handy for our purposes, even though nobody has written such a class yet. Also, as we write larger programs, we will need some way to break a program into independent parts, so that the task of programming becomes more manageable. For both situations, the solution is to develop our own class.

Large Java programs are broken into *many* interrelated classes. So it is very important that we understand how to write our own classes and get them to interact. We study the fundamentals of this in this chapter.

14.1. A simple example

Just as we had a set of design questions to answer before defining a method, we also have several questions to answer before defining a class.

- What will the class do? Generally speaking, you should be able to summarize the purpose of the class within a brief sentence.
- What name will the class have? Conventionally, the every word in the class name is capitalized, as with *GraphicsProgram*.
- What existing class will the class extend? Every class must extend some other class. But sometimes no superclass seems appropriate, and in this case the right choice is `Object`.
- When instances of the class are created, what information will need to be specified for the new object? This will indicate the parameters to include for the class's constructor. Or, if we want to provide multiple ways of specifying the information for a new instance, then we would need multiple constructors.
- What methods will instances of the class be able to perform?

One class that would be handy for our programs would be the notion of a button — basically, a rectangular region with a word nested within it, which the user could click. We'll call it the `GButton` class. An example where this might be handy would be a program with a moving ball and two buttons to control whether the ball is moving, as illustrated in Figure 14.1.

Figure 14.1: Running `StopGoBall`.

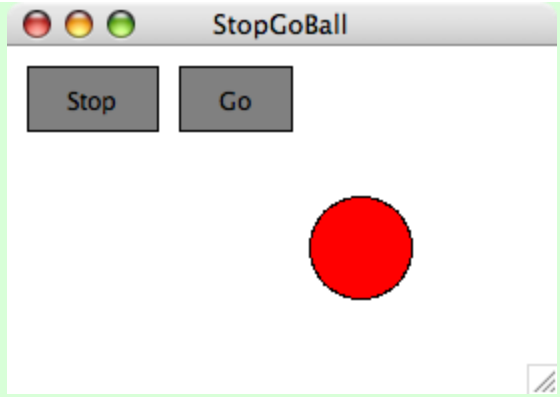


Figure 14.2 contains the `StopGoBall` program, which illustrates such a program using a hypothetical class `GButton`.

Figure 14.2: The `StopGoBall` program.

```
1  import acm.graphics.*;
2  import acm.program.*;
3  import java.awt.*;
4  import java.awt.event.*;
5
6  public class StopGoBall extends GraphicsProgram {
7      private boolean going;
8      private GButton stop;
9      private GButton go;
10
11     public void run() {
12         going = true;
13         stop = new GButton("Stop", 10, 10);
14         go = new GButton("Go", 10 + stop.getWidth() + 10, 10);
15
16         add(stop);
17         add(go);
18         addMouseListeners();
19
20         double dx = 2;
21         GOval ball = new GOval(75, 75, 50, 50);
22         ball.setFilled(true);
23         ball.setFillColor(Color.RED);
24         add(ball);
25
```

```
26          while(true) {
27              pause(20);
28              if(going) {
29                  if(ball.getX() < 0 || ball.getX() + 50 > getWidth()) {
30                      dx = -dx;
31                  }
32                  ball.move(dx, 0);
33              }
34          }
35      }
36
37      public void mouseClicked(MouseEvent e) {
38          if(stop.contains(e.getX(), e.getY())) {
39              going = false;
40          } else if(go.contains(e.getX(), e.getY())) {
41              going = true;
42          }
43      }
44  }
```

Imagining the `StopGoBall` program allows us to envision what methods we will need from a `GButton` class.

`GButton(String text, double x, double y)`

> (Constructor) Creates a button with its upper left corner at (`x`, `y`) and its label being `text`.

`boolean contains(double x, double y)`

> Returns **`true`** if the point (`x`, `y`) lies within this button.

`double getWidth()`

> Returns the width of this button.

That addresses the question of what methods the class needs. But what class should it extend? In thinking about this, we notice that a `GButton` is a combination of two`GObject`s — a rectangle and a label. We could thus think of a `GButton` as just a special type of `GCompound`, and so it is natural to make `GButton` be a subclass of `GCompound`. This will have the useful side effect that we won't have to define `contains` and `getWidth`: `GCompound` defines these

methods already, doing exactly what we want, and `GButton` will automatically inherit these methods if we extend it. Our implementation of `GButton` in Figure 14.3 takes advantage of this by defining the constructor only.

Figure 14.3: The `GButton` class.

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

public class GButton extends GCompound {
    public GButton(String text, double x, double y) {
        GLabel label = new GLabel(text, 0, 0);
        label.setLocation(20, 10 + label.getAscent());

        GRect background = new GRect(0, 0, label.getWidth() + 40,
            label.getAscent() + 20);
        background.setFilled(true);
        background.setFillColor(Color.GRAY);

        add(background);   // add background first, so it is below label
        add(label);
        setLocation(x, y); // reposition entire GCompound to (x,y)
    }
}
```

Many professional Java programmers would argue that using `GCompound` as a superclass for `GButton` would be a flawed design: After all, we think of buttons as unit objects, not as compounds. Moreover, `GButton` will inherit several methods, such as `add`, that have no obvious relationship with buttons. Such programmers would agree that `GButton` should definitely be a subclass of `GObject`, since it is a graphical object, and perhaps of `GRect`, since it's basically a rectangle that happens to have a word on top of it. We haven't done it that way, though, because doing that would involve some more complex techniques that don't really pertain to our discussion.

The primary point here is that Java professionals have definite opinions about when properly to use subclasses. Most would agree that one thing you should definitely not do is to select a superclass based purely on whether you could inherit methods that happen to work as desired.

Because the two instance methods that we need for the `StopGoBall` program — `contains` and `getWidth` — are automatically inherited, our implementation has only to

define the constructor. The definition of a constructor closely resembles the definition of an instance method; the difference is that you write the class name before the parameters, whereas with an instance method you would list the return type and method name.

```
public <className>(<parmType> <parmName>...) {
    <bodyOfMethod>
}
```

The purpose of a constructor is to set up an object that is in the process of being created. In this case, a newly created `GButton` should be initialized to have a combination of a `GLabel` and a `GRect`, and it adds them both into the `GCompound` using the inherited `add` method. The ordering is a bit peculiar, because the program must first create the label so that the rectangle for the background can be sized to be just a bit larger than the label; but the label must be added afterwards so that it is drawn on top of the background.

Beginners, accustomed to always having a return type just after **`public`**, often mistakenly insert the word **`void`**. For example, a beginner might write:

```
public void GButton(String text, double x, double y) { // bad!!
```

Rather than reading this as a constructor, the compiler will understand it as the definition of an instance method, which happens to be named `GButton`, and which doesn't return anything. Thus, if we could create a `GButton` variable `stop`, we would be able to write `stop.GButton("str", 0, 0)` to invoke the instance method. This is not what the programmer means, but the compiler will not complain about it.

The compiler will still refuse to compile the program, but its explanation it gives will not describe the problem well: The compiler will point to the line in `StopGoBall` where the `GButton` is constructed, and it will indicate that `GButton` doesn't have a constructor. The compiler has of course misdiagnosed the problem completely.

The solution is to delete the word **`void`**, so that the compiler interprets it as a constructor definition.

14.2. Instance variables

Section 13.3 discussed instance variables, but it did not discuss one of their more important characteristics: Each instance of the class has its own version of the instance variable. Thus, if `GButton` had an instance variable named `label`, then every `GButton` will have its own variable named `label`. (In fact, `GButton` doesn't have any instance variables as defined in Figure 14.3. There is a `label` variable, but it is a local variable, not an instance variable,

since its declaration appears inside the constructor.) When an instance method refers to an instance variable, then the computer will access the variable associated with the particular object that is performing the method (i.e., **this**'s version of the instance variable).

Let's consider an example. Suppose that we want to modify our `GButton` class so that it provides an instance method for changing the text in the button's label. We'll call this method `setLabel`. Such a method could be useful for our bouncing ball program: We may want to combine the two buttons into a single button, which says Stopwhen the ball is moving and Go when the ball is stopped. If we had a `setLabel` method, then we could accommodate this by modifying `StopGoBall`'s `mouseClicked`method to the following instead. We're imagining that the single button is now referenced by an instance variable named `stopGo`.

```
public void mouseClicked(MouseEvent e) {
    if(stopGo.contains(e.getX(), e.getY())) {
        if(going) {
            going = false;
            stopGo.setLabel("Go");
        } else {
            going = true;
            stopGo.setLabel("Stop");
        }
    }
}
```

For this to work, we will need to define a `setLabel` instance method in `GButton`. In order to write `setLabel`, we will need some way of referencing the label that the button contains. In our earlier version of `GButton` (Figure 14.3), `label` was a local variable available to the constructor only. We now need to reference that variable in`setLabel` also, so we will have to lift this variable declaration out of the constructor to become an instance variable instead. We'll do the same with `background`, since when the label is changed, the rectangle will need to be resized to match the size of the label. Figure 14.4 contains this modified class.

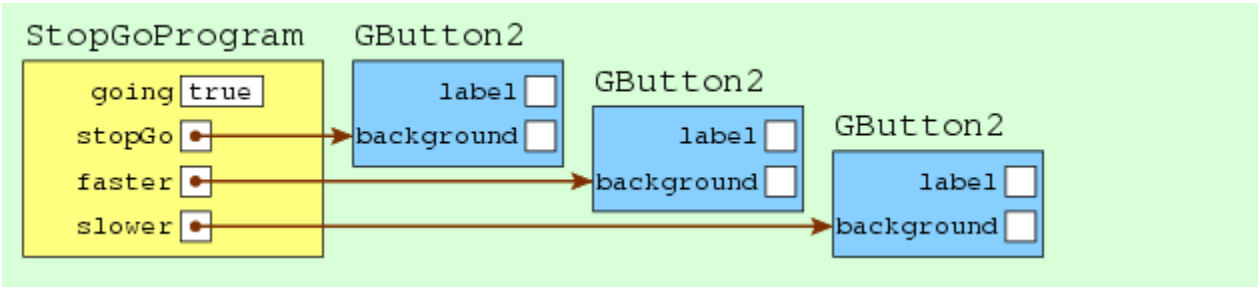
Figure 14.4: The modified `GButton` class.

```
1  import acm.graphics.*;
2  import java.awt.*;
3
4  public class GButton extends GCompound {
5      private GLabel label;
6      private GRect background;
```

```
7
8       public GButton(String text, double x, double y) {
9           label = new GLabel(text, 0, 0);
10          label.setLocation(20, 10 + label.getAscent());
11
12          background = new GRect(0, 0, label.getWidth() + 40,
13              label.getAscent() + 20);
14          background.setFilled(true);
15          background.setFillColor(Color.GRAY);
16
17          add(background);   // add background first, so it is below label
18          add(label);
19          setLocation(x, y); // reposition entire GCompound to (x,y)
20      }
21
22      public void setLabel(String text) {
23          label.setLabel(text);
24          background.setSize(label.getWidth() + 40, background.getHeight());
25      }
26  }
```

If we have a program containing several `GButton` objects, then each will have its own `label` and `background` variables. If `StopGoProgram` also had two more `GButton`s referenced by instance variables named `faster` and `slower`, then the computer will remember all these instance variables as diagrammed in Figure 14.5.

Figure 14.5: Multiple buttons as represented in computer's memory.



When a program invokes `setLabel` on a button, that `setLabel` method will end up modifying the `GLabel` and `GRect` variables contained in that button. Invoking `setLabel`on a different button will modify different `GLabel` and `GRect` variables.

14.3. Protection levels

Up until now, we've consistently labeled instance variables as **private** and instance methods as **public**. Actually, instance variables can be labeled **public** also. If we do this, then other classes can access those variables. For example, if `label` were a public instance variable, then `StopGoButton` could perform the following to change the label to read Stop.

```
stopGo.label.setLabel("Stop");
```

This line accesses the `label` instance variable within the `GButton` referenced by `stopGo`, and it tells that `GLabel` to change its text. (Of course, the background rectangle's size would not change, since `GLabel`'s `setLabel` method is a different method from `GButton`'s `setLabel` method, and the `GLabel` has no way of knowing about the background rectangle.)

A large majority of professional programmers agree that instance variables should never be **public**. The reason is that in the context of a large team project, having an instance variable be public loses control of the instance variable, since then other classes might work around shortcomings in the methods by manipulating the variables in unexpected ways. The possibility above, where the label changes but not the background, is an example. Thus, instance variables should always be declared **private**.

Methods can be declared **public** or **private**. A method would be declared **private** if it is only intended to be used within that class, as a helper for other methods. Our `createBalloon` method from Section 12.1 is an example of such a method: We wanted the method so that other methods in the same class could use it, but we have no reason to allow methods in other classes to access `createMethod`. It would make more sense to define it as a **private** instance method.


Though programs with dozens of classes may need other choices, beginners' programs should abide by the following rules: Always declare instance variables as **private**; and always declare methods as **public** or **private**, as appropriate.


You should always specify whether a class member, such as an instance variable, constructor, or instance method, is **public** or **private**. This is for stylistic reasons. In fact, the Java compiler will allow you to declare a class member without specifying whether it is **public** or **private**. If you do this, the class member will be mostly public but not quite. If you had a program spanning multiple packages, then the variable would be accessible only within classes in the same package. We're not concerned with such multi-package programs in this book, though, so don't worry about this detail: Simply remember always to declare each class member as **public** or **private**.

14.4. Another example: Fractions

Let's consider another example: Let's suppose we want to write a program that computes fractions. Of course, we could use **`double`**s, but a **`double`** doesn't represent the number involved precisely, and the correct denominator for a particular **`double`** isn't immediately available. Thus we'd like to introduce a new type — the `Fraction`type — to represent a fraction properly.

Introducing a class has some disadvantages relative to just using **`double`**s, though. We'll have to use **`new`** to create new `Fraction` objects, rather than type numbers directly. An even bigger pain is that Java doesn't allow the standard operators, like '`+`' or '`*`', to manipulate objects. (The built-in `String` class, which permits '`+`' to be applied, is a special exception to this rule.) Instead, we'll have to use instance methods. Thus, to compute ½ + ¼, we'll end up having to use an instance method to request an addition. We'll call the method `add`, and it will take a fraction as a parameter, add it to the fraction on which the method was invoked, and return the resulting fraction. To express the computation ½ + ¼, then, we would use the following rather cumbersome Java code.

```
Fraction half = new Fraction(1, 2);
Fraction third = new Fraction(1, 3);
Fraction result = half.add(third);
```

As we begin to consider writing the `Fraction` class, a useful question with which to start is to ask what each individual fraction would need to know in order to understand its identity. In this case, each `Fraction` object is determined by two integers — a numerator and a denominator. These two integers will therefore be the instance variables for the `Fraction` class.

Figure 14.6 contains the definition of a `Fraction` class. Notice that this class isn't defined to extend any other class, since there doesn't really seem to be an appropriate superclass. When the superclass is left unspecified, then Java will make `Object` be the superclass. (We could explicitly write **`extends`** `Object`, which achieves the same thing.)

Figure 14.6: The `Fraction` class.

```
1  public class Fraction {
2      private int num;
3      private int den;
4
5      public Fraction(int value) {
```

```
6           num = value;
7           den = 1;
8       }
9
10      public Fraction(int numerator, int denominator) {
11          num = numerator;
12          den = denominator;
13      }
14
15      public Fraction multiply(Fraction other) {
16          return new Fraction(this.num * other.num, this.den * other.den);
17      }
18
19      public Fraction add(Fraction other) {
20          int n = this.num * other.den + other.num * this.den;
21          return new Fraction(n, this.den * other.den);
22      }
23
24      public String toString() {
25          return num + "/" + den;
26      }
27  }
```

The `Fraction` class defines two constructors and three instance methods. The two constructors allow us either to construct a `Fraction` corresponding to an integer, or to construct a `Fraction` corresponding to a known numerator and denominator. The three instance methods allow us to ask a fraction to add itself to another fraction, to multiply itself by another fraction, or to compute a string representation. (The `toString` method in fact replaces the `toString` method that `Fraction` has inherited from the `Object` class. We'll revisit this in a later chapter.)

An alternative way to write the first constructor would be:

```
public Fraction(int value) {
    this(value, 1);
}
```

When the first line of a constructor looks like `this(...)`, Java invokes another constructor with the indicated parameters. Thus, if we create a fraction with `new Fraction(2)`, which invokes the one-argument constructor, the computer would see the `this(...)` line at that constructor's beginning and promptly forward the invocation to the two-argument

constructor, with `1` being the second argument. This is a fairly minor feature of Java, but later we'll see a slightly different form (substituting **`super`** for **`this`**), which will be fairly important.

Another thing that you'll notice is that the `multiply` and `add` methods talk about **`this`**`.num` and **`this`**`.den`. This is a long-winded alternative to writing `num` and `den`. We could delete **`this`**`.` every time it occurs, and the class would work identically. But for the aesthetic principle of symmetry, my personal preference is to opt for the long-winded version when a method accessing the same instance variables in a different object of the same class.

That brings us to another interesting point to observe about this class. Even though `num` and `den` are private, we are still allowed to reference their values on `Fraction`objects other than **`this`**. Thus, the references to `other.num` and `other.den` are legal. The word private prevents accessing those variables only from within other classes. In this case, `other` is a `Fraction`, and we are writing a `Fraction` instance method, so writing `other.num` is legitimate. The rationale underlying this design in Java is thatprivateness is to insulate other classes from being able to access internal design decisions, not to insulate a class's design from itself.

Source: http://www.toves.org/books/java/ch14-makeclass/index.html