ABSTRACT CLASSES

Whenever a *Rectangle*, *Oval*, or *RoundRect* object has to draw itself, it is the `redraw()` method in the appropriate class that is executed. This leaves open the question, What does the `redraw()` method in the*Shape* class do? How should it be defined?

The answer may be surprising: We should leave it blank! The fact is that the class *Shape* represents the abstract idea of a shape, and there is no way to draw such a thing. Only particular, concrete shapes like rectangles and ovals can be drawn. So, why should there even be a `redraw()` method in the *Shape* class? Well, it has to be there, or it would be illegal to call it in the `setColor()` method of the *Shape* class, and it would be illegal to write "`oneShape.redraw();`". The compiler would complain that `oneShape` is a variable of type *Shape* and there's no `redraw()` method in the *Shape* class.

Nevertheless the version of `redraw()` in the *Shape* class itself will never actually be called. In fact, if you think about it, there can never be any reason to construct an actual object of type *Shape*! You can have**variables** of type *Shape*, but the objects they refer to will always belong to one of the subclasses of *Shape*. We say that *Shape* is an abstract class. An abstract class is one that is not used to construct objects, but only as a basis for making subclasses. An abstract class exists **only** to express the common properties of all its subclasses. A class that is not abstract is said to be concrete. You can create objects belonging to a concrete class, but not to an abstract class. A variable whose type is given by an abstract class can only refer to objects that belong to concrete subclasses of the abstract class.

Similarly, we say that the `redraw()` method in class *Shape* is an abstract method, since it is never meant to be called. In fact, there is nothing for it to do -- any actual redrawing is done by `redraw()` methods in the subclasses of *Shape*. The `redraw()` method in *Shape* has to be there. But it is there only to tell the computer that **all** `Shapes` understand the `redraw` message. As an abstract method, it exists merely to specify the common interface of all the actual, concrete versions of `redraw()` in the subclasses. There is no reason for the abstract `redraw()` in class *Shape* to contain any code at all.

Shape and its `redraw()` method are semantically abstract. You can also tell the computer, syntactically, that they are abstract by adding the modifier "`abstract`" to their definitions. For an abstract method, the block of code that gives the implementation of an ordinary method is replaced by a semicolon. An implementation must then be provided for the abstract method in any concrete subclass of the abstract class. Here's what the *Shape* class would look like as an abstract class:

```
public abstract class Shape {

    Color color;   // color of shape.

    void setColor(Color newColor) {
          // method to change the color of the shape
       color = newColor; // change value of instance variable
       redraw();  //  redraw  shape,  which  will  appear  in  new
color
    }

    abstract void redraw();
          // abstract method -- must be defined in
          // concrete subclasses

    . . .  // more instance variables and methods
```

```
} // end of class Shape
```

Once you have declared the class to be `abstract`, it becomes illegal to try to create actual objects of type *Shape*, and the computer will report a syntax error if you try to do so.

Note, by the way, that the *Vehicle* class discussed above would probably also be an abstract class. There is no way to own a vehicle as such -- the actual vehicle has to be a car or a truck or a motorcycle, or some other "concrete" type of vehicle.

Recall from Subsection 5.3.3 that a class that is not explicitly declared to be a subclass of some other class is automatically made a subclass of the standard class *Object*. That is, a class declaration with no "`extends`" part such as

```
public class myClass { . . .
```

is exactly equivalent to

```
public class myClass extends Object { . . .
```

This means that class *Object* is at the top of a huge class hierarchy that includes every other class. (Semantically, *Object* is an abstract class, in fact the most abstract class of all. Curiously, however, it is not declared to be `abstract` syntactically, which means that you can create objects of type *Object*. What you would do with them, however, I have no idea.)

Since every class is a subclass of *Object*, a variable of type *Object* can refer to any object whatsoever, of any type. Java has several standard data structures that are designed to hold *Objects*, but since every object is an instance of class *Object*, these data structures can actually hold any object whatsoever. One example is the

"ArrayList" data structure, which is defined by the class *ArrayList* in the package `java.util`. (*ArrayList* is discussed more fully in Section 7.3.) An *ArrayList* is simply a list of *Objects*. This class is very convenient, because an *ArrayList* can hold any number of objects, and it will grow, when necessary, as objects are added to it. Since the items in the list are of type `Object`, the list can actually hold objects of any type.

A program that wants to keep track of various *Shapes* that have been drawn on the screen can store those shapes in an *ArrayList*. Suppose that the *ArrayList* is named `listOfShapes`. A shape, such as `oneShape`, can be added to the end of the list by calling the instance method "`listOfShapes.add(oneShape);`". The shape can be removed from the list with the instance method "`listOfShapes.remove(oneShape);`". The number of shapes in the list is given by the function "`listOfShapes.size()`". And it is possible to retrieve the i-th object from the list with the function call "`listOfShapes.get(i)`". (Items in the list are numbered from 0 to `listOfShapes.size() - 1`.) However, note that this method returns an *Object*, not a *Shape*. (Of course, the people who wrote the `ArrayList` class didn't even know about *Shapes*, so the method they wrote could hardly have a return type of *Shape*!) Since you know that the items in the list are, in fact, *Shapes* and not just *Objects*, you can type-cast the *Object* returned by `listOfShapes.get(i)` to be a value of type *Shape*:

```
oneShape = (Shape)listOfShapes.get(i);
```

Let's say, for example, that you want to redraw all the shapes in the list. You could do this with a simple `for` loop, which is a lovely example of object-oriented programming and of polymorphism:

```
for (int i = 0; i < listOfShapes.size(); i++) {
```

```
   Shape s;  // i-th element of the list, considered as a Shape
   s = (Shape)listOfShapes.get(i);
   s.redraw();  // What is drawn here depends on what type of
shape s is!
}
```

The sample source code file *ShapeDraw.java* uses an abstract *Shape* class and an *ArrayList* to hold a list of shapes. The file defines an applet in which the user can add various shapes to a drawing area. Once a shape is in the drawing area, the user can use the mouse to drag it around.

You might want to look at this file, even though you won't be able to understand all of it at this time. Even the definitions of the shape classes are somewhat different from those that I have described in this section. (For example, the `draw()` method has a parameter of type *Graphics*. This parameter is required because of the way Java handles all drawing.) I'll return to similar examples in later chapters when you know more about GUI programming. However, it would still be worthwhile to look at the definition of the *Shape* class and its subclasses in the source code. You might also check how an `ArrayList` is used to hold the list of shapes.

If you click one of the buttons along the bottom of this applet, a shape will be added to the screen in the upper left corner of the applet. The color of the shape is given by the "pop-up menu" in the lower right. Once a shape is on the screen, you can drag it around with the mouse. A shape will maintain the same front-to-back order with respect to other shapes on the screen, even while you are dragging it. However, you can move a shape out in front of all the other shapes if you hold down the shift key as you click on it.

In the applet, the only time when the actual class of a shape is used is when that shape is added to the screen. Once the shape has been created, it is manipulated entirely as

an abstract shape. The routine that implements dragging, for example, works with variables of type *Shape* and makes no reference to any of its subclasses. As the shape is being dragged, the dragging routine just calls the shape's draw method each time the shape has to be drawn, so it doesn't have to know how to draw the shape or even what type of shape it is. The object is responsible for drawing itself. If I wanted to add a new type of shape to the program, I would define a new subclass of *Shape*, add another button to the applet, and program the button to add the correct type of shape to the screen. No other changes in the programming would be necessary.

Source : http://math.hws.edu/javanotes/c5/s5.html