Chapter 8 Interfaces & Subclasses

8.1 Interfaces & Subclasses

In our previous examples, the type of a variable has always been the same as the type of the object named by that variable:

```
Color frameColor;.... frameColor = new Color(255, 150, 0);
Font headingFont;.... headingFont = new Font("Serif", Font.BOLD, 24);
AButton bigger;..... bigger = new AButton("Bigger", 300, 50);
```

It is common to have a list of similar objects that are different enough from each other that some should be described by one class and some by another. For example, we would like to write:

```
list[0] = new Oval(10, 20, 100, 50);
list[1] = new Box(150, 80, 100, 50);
list[2] = new Oval(120, 80, 100, 50);
```

There are two features of Java intended for this type of situation, *interfaces* and *subclasses*.

- Interfaces are the simplest to understand and use.
- In addition to providing a solution to the mixed list problem, subclasses allow objects to be described partly by one class and partly by another and are correspondingly more complex.

8.2 Interfaces

An *interface* lists a set of features that a class must have to be used in some particular situation. Generally, an *interface* is a list of the signatures of *public* methods. For example:

```
interface AShape
   {
   public void paint(Graphics g);
   public boolean contains(int x, int y);
   public void setLocation(int left, int top);
   public void setSize(int width, int height);
   public void setColor(Color c);
   }
```

The *interface AShape* declares that an *AShape* must have five methods that are called as indicated. The *interface* does not say how these methods do what they do. The program fragment below puts three *AShapes* into an array. *Ovals* and *Boxes* must be *AShapes*. The definition of "being an *AShape*" is that the first line of the class contains *implements AShape* and the class contains methods such as described in the *interface AShape*.

```
AShape list[];
.......
list = new AShape[3];
.......
list[0] = new Oval(10, 20, 100, 50);
list[0].setColor(Color.blue);
list[1] = new Box(10, 80, 100, 50);
list[1].setColor(Color.red);
list[2] = new Oval(120, 80, 100, 50);
```

Each of the two classes below is an *AShape* because each declares that it *implements AShape* and each has appropriate methods.

```
class Box implements AShape
   {
   private int left, top, width, height;
   private Color theColor;

   public Box(int l, int t, int w, int h)
      {
      left = l; top = t; width = w; height = h;
      theColor = Color.black;
      }

   public void paint(Graphics g)
      {
      g.setColor(theColor);
      g.fillRect(left, top, width, height);
      }

   public boolean contains(int x, int y)
      {
      return left<x && x<left+width && top<y && y<top+height;
      }

   public void setLocation(int l, int t)
      {
      left = l; top = t;
      }

   public void setSize(int h, int w)
      {
      width = w; height = h;
      if (width < 5) width = 5;
      if (height < 5) height = 5;
      }

   public void setColor(Color c)
      {
      theColor = c;
      }
   }
```

The class *Oval* is identical to *Box* except for the changes indicated.

```
class ~~Box~~Oval implements AShape
   {
   ...
   public ~~Box~~Oval(int l, int t, int w, int h)
   ...
   public void paint(Graphics g)
      ...
      g.fill~~Rect~~Oval(left, top, width, height);
      ...

   public boolean contains(int x, int y)
      {
      ~~return left<x && x<left+width && top<y && y<top+height;~~
      double centerX = (left + 0.5*width), centerY = (top + 0.5*height);
      double xSquared = (x-centerX)*(x-centerX);
      double ySquared = (y-centerY)*(y-centerY);
      double ratio = width/height;
      return xSquared + ratio*ratio*ySquared < width*width/4.0;
      }
   ...
   }
```

The program that follows uses the *interface AShape* and the classes *Box* and *Oval* to create a program that puts a mixture of ovals and rectangles on the screen. To make an oval, click with the mouse. To make a rectangle, hold down the control key while clicking with the mouse.

```
public class ShapeProgram extends EventPanel
    {
    private AShape list[];
    private int howMany;

    public ShapeProgram()
        {
        list = new AShape[100];
        howMany = 0;
        }

    public void paint(Graphics g)
        {
        for (int h=0; h<howMany; h++)
            list[h].paint(g);
        }

    public void mousePressed(MouseEvent e)
        {
        int x = e.getX();
        int y = e.getY();
        if (howMany < 100)
            {
            if (e.isControlDown())
                list[howMany] = new Box(x-50, y-25, 100, 50);
            else
                list[howMany] = new Oval(x-50, y-25, 100, 50);
            howMany++;
            repaint();
            }
        }
    }
```

Exercise — 8.2

1. Use an *interface* to allow the user to put ovals and rectangles among the stars in one or more programs of chapter 5. Your *interface* must have a *morePoints* and a *fewerPoints* method. So do the *Oval* and *Box* classes. The *morePoints* and *fewerPoints* methods of the *Oval* and *Box* classes need not do anything, just exist.

8.3 Subclasses – classes that extend others

To extend a class write *extends* and the name of a class at the beginning of the class definition.

```
class Peacock extends Bird
   {
```

All but one class in Java is an extension of another. The lone exception is the class *Object*, which all other classes extend. When you declare a class and 'don't extend another class', Java acts as though you had typed *extends Object*.

The term subclass is used for a class that extends another. Subclass is derived from the mathematical term subset. The class *Peacock* (above) *extends Bird*, creating a *Peacock* as a fancied up *Bird*. We consider an object created from the class *Peacock* to be both a *Peacock* and a *Bird*. *Peacock* is said to be a subclass of *Bird*, emphasizing that the set of all *Peacock* objects is a subset of the set of all *Bird* objects.

A class that has been extended (*Bird*) is called the superclass (of *Peacock*).

The terms base class (for superclass) and extending class (for subclass) are also used and expresses a central aspect of the relationship — the subclass extends the capabilities of the superclass.

When one class extends another, it inherits methods, variables, interfaces and superclasses. The new class can also override an inherited method — replacing it with its own. This is done by providing the subclass with a method having the same signature as the method to be overridden.

- If a subclass has a method with the same signature as a method of the superclass, the new method will override (be used instead of) the method of the superclass.
- If a subclass has no method with a signature like a given method of its superclass, objects created from the subclass will use the method of the superclass.

In previous program examples, we extended *EventPanel*, which is itself an extended version of the class *Panel*. We extended *EventPanel* so we could use methods that existed in *Panel* (*repaint*, *setBackground*). We also supplied methods that did something to override do-nothing methods that existed in *Panel* (*paint*) and *EventPanel* (*keyPressed*, *mousePressed*).

Extending a class brings up the question of accessing the variables and calling the methods of the superclass. It is often appropriate to do this.

- In general, methods in the superclass can be accessed if they have been declared *public*. Accessor and mutator methods are placed in the superclass so that variables can be accessed by the subclass.
- Methods in a subclass can access a method overridden by that subclass (rather than the one in the subclass that has done the overriding). To do this write *super.* in front of the name of the method when you call it. For example, to access an overridden paint method write *super.paint(g)*.
- To access a constructor of the class you have extended, you write *super* in front of the parameters for the constructor. This is only done as the first command in a constructor. A example appears in the class *Oval* below.

An extended class

The following example creates the class *Oval* of the previous example. It will work as written if accessor methods are included in the class *Box*.

```
class Oval extends Box
   {
   public Oval(int l, int t, int w, int h)
      {
      super(l, t, w, h);       // The constructor of the superclass (Box) initializes the variables.
      }                        // The constructor for the superclass must be used.

   public void paint(Graphics g)
      {
      g.setColor(getTheColor());              // These get… methods are in the superclass.
      g.fillOval(getLeft(), getTop(), getWidth(), getHeight());
      }

   public boolean contains(int x, int y)
      {
      int width = getWidth(), height = getHeight();
      if (height == 0)
         return false;
      double centerX = (getLeft() + 0.5*width);     // Uses the Pythagorean theorem with
      double centerY = (getTop() + 0.5*height);     // a fudge factor (ratio) for the oval.
      double xSquared = (x-centerX)*(x-centerX);
      double ySquared = (y-centerY)*(y-centerY);
      double ratio = width / height;
      return xSquared + ratio*ySquared < height/4.0;
      }
   }
```

The above is the complete class *Oval* (given that *Box* is available with accessor methods *getTheColor*, *getTop* etc.). Compare this to the description in section 8.2 of how to modify *Box* to obtain *Oval*. Objects created with the class *Oval* as written above can be used in a list of *AShapes*. They inherit the property of implementing *AShape*.

Extending a class is like Implementing an interface

In one very important way, the relationship between superclass and subclass is the same as the relationship between interface and implementing class: an implementing class has all of the methods of the interface — a subclass has all of the methods of the superclass.

Extending a class may override methods, supplying a substitute. Extending never hides a method without providing a replacement. In fact, overriding a *public* method with a *private* one is not permitted, just so that this will be true.

Java recognizes this similarity by giving a class that extends another similar rights to those of a class that implements an interface. A variable whose type is a class can be used to refer to an object belonging to a subclass:

```
class Oval extends Box
   ...
   Box b;
   ...
   b = new Box();   or   b = new Oval();
```

Either of the assignments above is acceptable. We can do this because *Oval* has methods to match those of *Box*. *b.paint(g)*, *b.contains(x, y)*… make sense in either case.

An example that uses overridden methods

This class adds to the overridden methods *setColor* and *paint* rather than completely replacing them:

```
class OvalOnBox extends Box
    {
    private Color ovalColor;

    public OvalOnBox(int l, int t, int w, int h)
        {
        super(l, t, w, h);              // The superclass constructor must come first.
        ovalColor = Color.darkGray;     // The variables of the subclass are then initialized.
        }

    public void setColor(Color c)
        {
        if (!c.equals(c.darker()))      // put a darker color behind (if different)
            {
            super.setColor(c.darker());
            ovalColor = c;
            }
        else if (!c.equals(c.brighter()))       // else put a brighter color in front (if different)
            {
            super.setColor(c);
            ovalColor = c.brighter();
            }
        else                            // black is unchanged by both brighter and darker.
            {
            super.setColor(c);
            ovalColor = Color.darkGray;
            }
        }

    public void paint(Graphics g)
        {
        super.paint(g);
        g.setColor(ovalColor);
        g.fillOval(getLeft(), getTop(), getWidth(), getHeight());
        }
    }
```