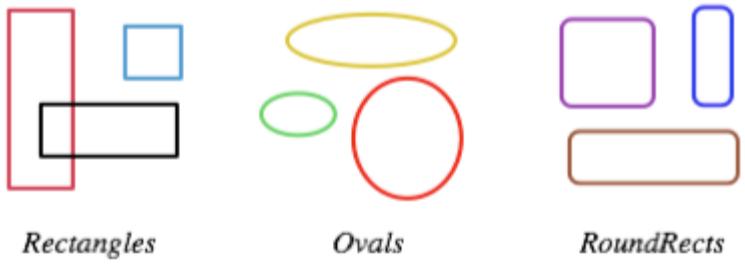
# POLYMORPHISM IN JAVA

As another example, consider a program that deals with shapes drawn on the screen. Let's say that the shapes include rectangles, ovals, and roundrects of various colors. (A "roundrect" is just a rectangle with rounded corners.)




Three classes, *Rectangle*, *Oval*, and *RoundRect*, could be used to represent the three types of shapes. These three classes would have a common superclass, *Shape*, to represent features that all three shapes have in common. The *Shape* class could include instance variables to represent the color, position, and size of a shape, and it could include instance methods for changing the color, position, and size. Changing the color, for example, might involve changing the value of an instance variable, and then redrawing the shape in its new color:

```
class Shape {

    Color color;    // Color of the shape.  (Recall that class
Color
                    // is defined in package java.awt.  Assume
                    // that this class has been imported.)

    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
```

```
    }

    void redraw() {
          // method for drawing the shape
       ? ? ?  // what commands should go here?
    }

    . . .          // more instance variables and methods

} // end of class Shape
```

Now, you might see a problem here with the method `redraw()`. The problem is that each different type of shape is drawn differently. The method `setColor()` can be called for any type of shape. How does the computer know which shape to draw when it executes the `redraw()`? Informally, we can answer the question like this: The computer executes `redraw()` by asking the shape to redraw **itself**. Every shape object knows what it has to do to redraw itself.

In practice, this means that each of the specific shape classes has its own `redraw()` method:

```
class Rectangle extends Shape {
   void redraw() {
      . . .  // commands for drawing a rectangle
   }
   . . . // possibly, more methods and variables
}

class Oval extends Shape {
   void redraw() {
      . . .  // commands for drawing an oval
   }
   . . . // possibly, more methods and variables
}
```

```
class RoundRect extends Shape {
   void redraw() {
      . . .  // commands for drawing a rounded rectangle
   }
   . . . // possibly, more methods and variables
}
```

If `oneShape` is a variable of type *Shape*, it could refer to an object of any of the types *Rectangle*, *Oval*, or *RoundRect*. As a program executes, and the value of `oneShape` changes, it could even refer to objects of different types at different times! Whenever the statement

```
oneShape.redraw();
```

is executed, the redraw method that is actually called is the one appropriate for the type of object to which `oneShape` actually refers. There may be no way of telling, from looking at the text of the program, what shape this statement will draw, since it depends on the value that `oneShape` happens to have when the program is executed. Even more is true. Suppose the statement is in a loop and gets executed many times. If the value of `oneShape` changes as the loop is executed, it is possible that the very same statement "`oneShape.redraw();`" will call different methods and draw different shapes as it is executed over and over. We say that the `redraw()` method is polymorphic. A method is polymorphic if the action performed by the method depends on the actual type of the object to which the method is applied. Polymorphism is one of the major distinguishing features of object-oriented programming.

Perhaps this becomes more understandable if we change our terminology a bit: In object-oriented programming, calling a method is often referred to as sending a message to an object. The object responds to the message by executing the appropriate method. The statement "`oneShape.redraw();`" is a message to the

object referred to by `oneShape`. Since that object knows what type of object it is, it knows how it should respond to the message. From this point of view, the computer always executes "`oneShape.redraw();`" in the same way: by sending a message. The response to the message depends, naturally, on who receives it. From this point of view, objects are active entities that send and receive messages, and polymorphism is a natural, even necessary, part of this view. Polymorphism just means that different objects can respond to the same message in different ways.

One of the most beautiful things about polymorphism is that it lets code that you write do things that you didn't even conceive of, at the time you wrote it. Suppose that I decide to add beveled rectangles to the types of shapes my program can deal with. A beveled rectangle has a triangle cut off each corner:

*BeveledRects*

To implement beveled rectangles, I can write a new subclass, *BeveledRect*, of class *Shape* and give it its own `redraw()` method. Automatically, code that I wrote previously -- such as the statement`oneShape.redraw()` -- can now suddenly start drawing beveled rectangles, even though the beveled rectangle class didn't exist when I wrote the statement!

In the statement "`oneShape.redraw();`", the `redraw` message is sent to the object `oneShape`. Look back at the method in the *Shape* class for changing the color of a shape:

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

A `redraw` message is sent here, but which object is it sent to? Well, the `setColor` method is itself a message that was sent to some object. The answer is that the `redraw` message is sent to that **same object**, the one that received the `setColor` message. If that object is a rectangle, then it contains a `redraw()` method for drawing rectangles, and that is the one that is executed. If the object is an oval, then it is the`redraw()` method from the *Oval* class. This is what you should expect, but it means that the "`redraw();`" statement in the `setColor()` method does **not** necessarily call the `redraw()` method in the *Shape*class! The `redraw()` method that is executed could be in any subclass of *Shape*. This is just another case of polymorphism.

Again, this is not a real surprise if you think about it in the right way. Remember that an instance method is always contained in an object. The class only contains the source code for the method. When a*Rectangle* object is created, it contains a `redraw()` method. The source code for that method is in the *Rectangle* class. The object also contains a `setColor()` method. Since the *Rectangle* class does not define a `setColor()` method, the **source code** for the rectangle's `setColor()` method comes from the superclass, *Shape*, but the **method itself** is in the object of type *Rectangle*. Even though the source codes for the two methods are in different classes, the methods themselves are part of the same object. When the rectangle's `setColor()` method is executed and calls `redraw()`, the `redraw()` method that is executed is the one in the same object.

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