Inheritance and Polymorphism

A CLASS REPRESENTS A SET OF OBJECTS which share the same structure and behaviors. The class determines the structure of objects by specifying variables that are contained in each instance of the class, and it determines behavior by providing the instance methods that express the behavior of the objects. This is a powerful idea. However, something like this can be done in most programming languages. The central new idea in object-oriented programming—the idea that really distinguishes it from traditional programming—is to allow classes to express the similarities among objects that share **some**, but not all, of their structure and behavior. Such similarities can be expressed using inheritance and polymorphism.

Extending Existing Classes

The existing class can be extended to make a subclass. The syntax for this is

```
public class subclass-name extends existing-class-name {
   .
   .   // Changes and additions.
   .
}
```

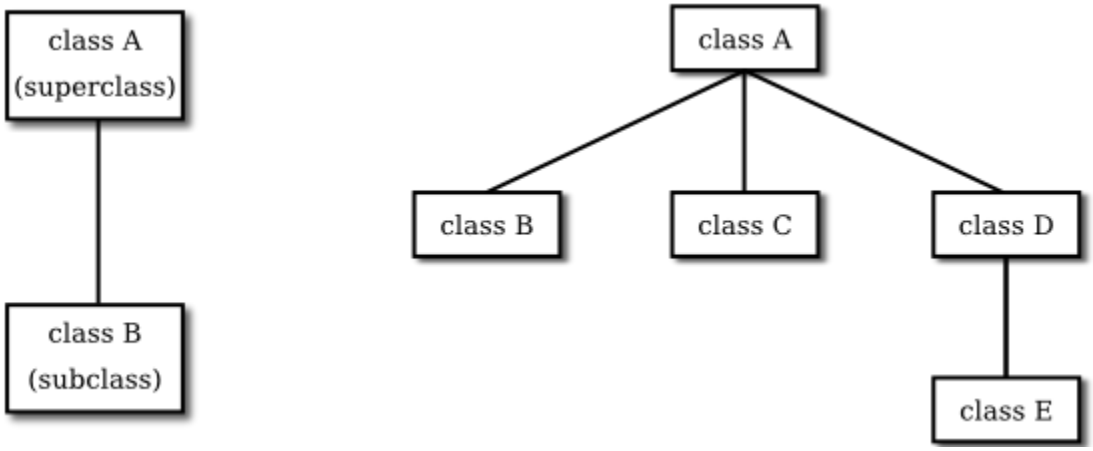
Extending existing classes is an easy way to build on previous work.

Access modifiers such as `public` and `private` are used to control access to members of a class. There is one more access modifier, protected, that comes into the picture when subclasses are taken into consideration. When `protected` is applied as an access modifier to a method or member variable in a class, that member can be used in subclasses—direct or indirect—of the class in which it is defined, but it cannot be used in non-subclasses. (There is an exception: A `protected` member can also be accessed by any class in the same package as the class that contains the protected member.)

When you declare a method or member variable to be `protected`, you are saying that it is part of the implementation of the class, rather than part of the public interface of the class. However, you are allowing subclasses to use and modify that part of the implementation.

Inheritance and Class Hierarchy

The term inheritance refers to the fact that one class can inherit part or all of its structure and behavior from another class. The class that does the inheriting is said to be a subclass of the class from which it inherits. If class B is a subclass of class A, we also say that class A is a superclass of class B. (Sometimes the terms derived class and base class are used instead of subclass and superclass; this is the common terminology in C++.) A subclass can add to the structure and behavior that it inherits. It can also replace or modify inherited behavior (though not inherited structure). The relationship between subclass and superclass is sometimes shown by a diagram in which the subclass is shown below, and connected to, its superclass, as shown on the left below:



In Java, to create a class named "B" as a subclass of a class named "A", you would write

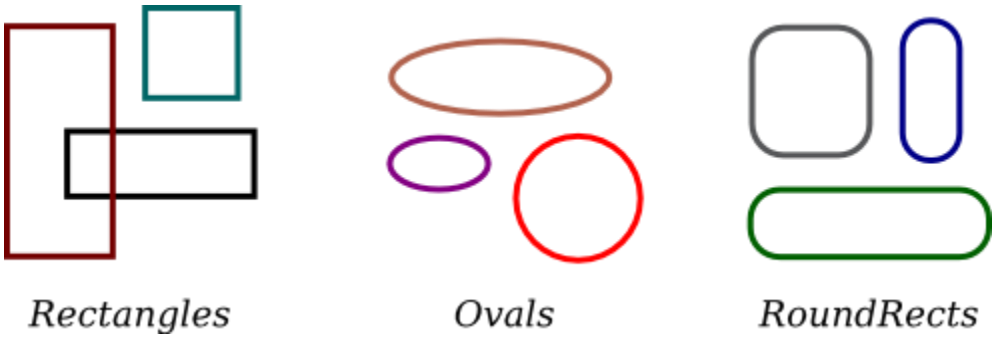
```
class B extends A {
    .
    .  // additions to, and modifications of,
    .  // stuff inherited from class A
    .
}
```

Several classes can be declared as subclasses of the same superclass. The subclasses, which might be referred to as "sibling classes," share some structures and behaviors—namely, the ones they inherit from their common superclass. The superclass expresses these shared structures and behaviors. In the diagram shown on the right above, classes B, C, and D are sibling classes. Inheritance can also extend over several "generations" of classes. This is shown in the diagram, where class E is a subclass of class D which is itself a subclass of class A. In this case, class E is considered to be a

subclass of class A, even though it is not a direct subclass. This whole set of classes forms a small class hierarchy.

Polymorphism

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 values of those properties. 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;

    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
}
```

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

```
someShape.redraw();
```

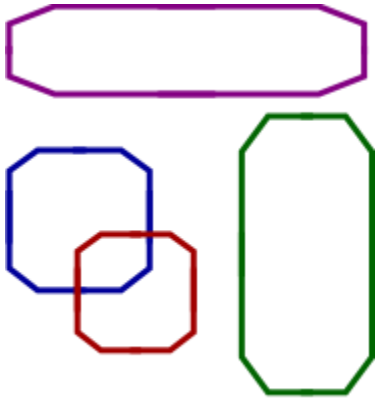
is executed, the redraw method that is actually called is the one appropriate for the type of object to which `someShape` actually refers. We say that the `redraw()` method is polymorphic. A method is polymorphic if the action performed by the method depends on the actual type of the object to which the method is applied at run time. Polymorphism is one of the major distinguishing features of object-oriented programming. This can be seen most vividly, perhaps, if

we have an array of shapes. Suppose that `shapelist` is a variable of type `Shape[]`, and that the array has already been created and filled with data. Some of the elements in the array might be *Rectangles*, some might be *Ovals*, and some might be *RoundRects*. We can draw all the shapes in the array by saying

```
for (int i = 0; i < shapelist.length; i++ ) {
    Shape shape = shapelist[i];
    shape.redraw();
}
```

As the computer goes through this loop, the statement `shape.redraw()` will sometimes draw a rectangle, sometimes an oval, and sometimes a roundrect, depending on the type of object to which array element number `i` refers.

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 `someShape.redraw()`—can now suddenly start drawing beveled rectangles, even though the beveled rectangle class didn't exist when I wrote the statement!