1 Class Inheritance

One of the most important aspects of object-oriented programming is the ability to derive classes from existing classes. When this is done, the derived class *inherit*s data members and methods of the parent class, and can replace (*override*) methods of the parent class if there is need.

The way to extend an existing class is to use the `extends` keyword in the class declaration line, as follows:

```
public class Point3D extends Point {
    // Class definition goes here.
}
```

The new class, Point3D, has all the functionality of the class `Point` externally. Plus, it has the additional functionality added in its own class definition.

2 Access Modifier Keywords

A data member or method in a class can be declared with an access modifier to determine what classes can have access to that particular data member or method. From most restricted to unrestricted, these are as follows:

`private`: Only the methods of this class can access this member.
`protected`: Only this class and subclasses can access this member.
`<no modifier>`: Methods of classes in same package can access this member.
`public`: Any method of any class can access this member.

3 Overriding Methods

It is possible to replace methods defined in a base class in the derived class. To do this, just define a method with the same signature (name and parameter list) as the base class. One restriction is that the access modifier for the overriding method must be less restrictive than that for the overridden method.

4 Calling Methods and Constructors of the Base Class

Within the definition of the derived class, it is possible to call the methods and constructors of the base class. This is done by using the keyword `super`.

It is a good idea to call the superclass constructor in the derived class constructor. The restriction is that the call must be the first thing in the derived class constructor. The call looks like:

```
super();  // Call the base class constructor.
```

Of course, this is for a constructor that takes no arguments. For other constructors, you need to supply the proper argument list.

Normally, you can call methods of the base class directly, but if you have overriden them in the base class, you need to prepend `super.` in order to refer to the methods of the parent class rather than the derived class.

5 Polymorphism

Polymorphism refers to the ability of taking on different shapes. In the context of object-oriented programming, it refers to the ability of a base class type variable being able to refer to objects of any derived classes, and method calls will automatically select and call the most specific method for that object. This may look difficult conceptually, but should be clear when we have examples.

6 Exercise

Write a new class, called `TestQuestionFrame`, which extends `java.awt.Frame`. The constructor should take one argument, `String question`, and when displayed, should show a frame with the question, and a checkbox group with choices A to E in them, and a button labeled "OK" in it.

7 Abstract Classes and more on Inheritance

Since this is a very important part of object-oriented programming, it is only fair that we spend more time on the subject of class derivation and inheritance.

We will do a concrete example, and expand (and modify) our set of geometry classes. Suppose we wish to write some sort of drawing/graphics program. In it, we will need various classes for the various geometric shapes that can be drawn. Now, we need a base class for all the geometric shapes - let us call it `Shape`. By this, we mean any kind of geometric shape which has a well defined perimeter and area. From this, we plan to derive our various geometric shapes, like triangle, rectangle, circle. Our first attempt at writing this base class will be the following:

```
public class Shape {

    public double getArea() {
        return 0.0;
    }

    public double getPerimeter() {
        return 0.0;
    }

}
```

Now, what did we do here? We defined our *base class*, and put into it two methods that return things every geometric shape (that we are interested in, anyway) should have. Note that both methods are quite dumb, and just return 0.0. This is because this is an abstract class, and does not refer to any real geometric object, and therefore there is no way of calculating these things.

Now it is time for us to derive our geometric shapes from this. Let us start with a rectangle (actually, a special sort of rectangle, one that has sides parallel to the x and y axes). Here is our new `Rectangle` class (I assume we have the usual `Point` class):

```
public class Rectangle extends Shape {
```

```
    private Point topLeftCorner;
    private double width;
    private double height;

    public Rectangle() {
        topLeftCorner = new Point(0.0, 0.0);
        width = 1.0;
        height = 1.0;
    }

    public Rectangle(Point topLeftCorner, double width, double height) {
        this.topLeftCorner = topLeftCorner;
        this.width = width;
        this.height = height;
    }

    public double getArea() {
        return width*height;
    }

    public double getPerimeter() {
        return 2.0*(width+height);
    }

    public double getWidth() {
        return width;
    }

    public double getHeight() {
        return height;
    }
}
```

Let us look at the `Rectangle` class. It has three member variables. One is a `Point`, which is the top left corner of the rectangle. The other two member variables are the width and height of the rectangle. It has two constructors, one default constructor, which constructs a unit square of which the top left corner is at the origin. The second one allows the construction of any arbitrary rectangle.

Then there are two methods, `getArea()` and `getPerimeter()`, which have the same signature as the methods in the base class. This is exactly how one *overrides* a method. When one calls the `getArea()` method of a `Rectangle`, this method is the one that is going to be called, and not the one in the base class. So, the base class method has been overriden by the derived class method. Of course, the same applies to the `getPerimeter()` method.

Note that the `Rectangle` class has also methods specific to itself, namely the `getHeight()` and `getWidth()` methods.

Let us derive another class, the `Circle` from the `Shape` class as follows:

```
public class Circle extends Shape {
    private Point center;
```

```
    private double radius;

    public Circle() {
        center = new Point(0.0, 0.0);
        radius = 1.0;
    }

    public Circle(Point center, double radius) {
        this.center = center;
        this.radius = radius;
    }

    public double getArea() {
        return Math.PI*radius*radius;
    }

    public double getPerimeter() {
        return 2.0*Math.PI*radius;
    }

    public double getRadius() {
        return radius;
    }

}
```

Note the similarities between `Circle` and `Rectangle`. Both have methods that override the base class methods, and both have methods specific to themselves.

This is a good point to introduce `abstract` classes, since the concept has already been well-formed. Note that `Shape` is not a concrete class – an object of type `Shape` does not really make much sense. A "shape" is really an abstraction we use to refer to a lot of different things. We had to define dummy methods which returned puppet values. That is really a nuisance. So, for the purpose of defining classes that are just for the purpose of providing a base class for other classes, and never be instantiated itself, Java provides the keyword `abstract`. Here is how we can rewrite the `Shape` class taking advantage of the `abstract` keyword:

```
public abstract class Shape {

    public abstract double getArea();
    public abstract double getPerimeter();

}
```

Now that looks much better. On the declaration line, the keyword `abstract` informs the compiler that this is an abstract class – which means no objects of this class can be instantiated, i.e., you can not use the `new` operator with this class directly. This also tells the compiler that it is allright for this class to declare `abstract` methods. In other words, any class which decalres `abstract` methods must be declared `abstract` itself.

8 Making use of Polymorphism

Polymorphism, as we have mentioned before, means taking different shapes. Here, the `abstract` class `Shape` (no pun intended) has multiple subclasses, which can be considered a different form of `Shape`.

The first step towards polymorphism is the ability of a base class variable to refer to an object of a derived class. For example, the following is a valid, legal code fragment:

```
Shape shape;
shape = new Rectangle();
```

This is possible exactly because `Rectangle` is a subclass of `Shape`. Now, there are things you can not do using the variable `shape` here. For example, you can not call any of the methods specific to the `Rectangle` class. Explicitly, the following code fragment will not compile:

```
int w = shape.getWidth(); // Wrong!
```

This is because the `Shape` class knows nothing about a `getWidth()` method. What is *does* know about, however, is very useful, and forms the heart of polymorphism. When you call a method through a base class variable, and that method is overriden in the actual object (which *must* be the case in an abstract base class) the overriding method (i.e. the method of the derived class) will be called. Here is an example demonstrating this:

```
public class TestingPolymorphism {
    public static void main(String[] args) {
        Shape shape1;
        Shape shape2;

        shape1 = new Circle(new Point(1.0, 1.0), 5.0);
        shape2 = new Rectangle(new Point(2.0, 2.0), 5.0, 4.0);

        System.out.println("Area of shape1: " + shape1.getArea());
        System.out.println("Area of shape2: " + shape2.getArea());
    }
}
```

In this short program, and the accompanying three objects, we can see polymorphism at work. Both `shape1` and `shape2` are variables of type `Shape`. However, they are assigned objects of different classes, which are both subclasses of `Shape`. At the final step of the demonstration, the `getArea()` method is called for each object. However, the `getArea()` method called depends on the actual object, for each object the proper overriding method is called. So, we get the *correct* values for the area of a circle and the area of a rectangle.

9 Casting with Classes

Casting, which we have already seen and used for basic types, works with classes as well. You can always assign a subclass object to a parent class variable, and no casting is required in that case.

Speaking more symbolically, it is always possible to assign an object to more general classes, i.e. parent classes. No cast is needed in that case. The opposite is also possible, you can also cast an object into a variable of a subclass. However, for that to work, there are two conditions. First, an explicit cast is required. Second, the object should really be of that type. If the cast does not make sense, it will produce an exception. Here is a demonstration:

```
public class TestingPolymorphism {
    public static void main(String[] args) {
        Shape shape1;
        Shape shape2;

        shape1 = new Circle(new Point(1.0, 1.0), 5.0);
        shape2 = new Rectangle(new Point(2.0, 2.0), 5.0, 4.0);

        System.out.println("Area of shape1: " + shape1.getArea());
        System.out.println("Area of shape2: " + shape2.getArea());

        Circle circle1 = (Circle)shape1;
        Circle circle2 = (Circle)shape2;

    }
}
```

The first cast is perfectly legal, since `shape1` does refer to a `Circle`, and it is being cast to a `Circle`. But, the second cast is not legal. The actual object is a `Rectangle`, and we are trying to cast it to a `Circle`. However, note that the program will compile just fine – the error in this case is going to be a *runtime* error. Run the program, and see what happens!

So, is there a way of testing for this runtime error? The answer is, of course, yes. There is an operator for checking whether a cast will work or not. This operator is `instanceof`. The form of the operator is as follows:

```
<variable> instanceof <class>
```

The result is a `boolean`. The result is `true` if and only if the object referenced to by `<variable>` can be cast into a variable of type `<class>`.

Perhaps it is best to see this by example. This final example demonstrates the use of the `instanceof` operator:

```
public class TestingPolymorphism {
    public static void main(String[] args) {
        Shape shape1;

        shape1 = new Circle(new Point(1.0, 1.0), 5.0);

        System.out.println("Area of shape1: " + shape1.getArea());

        if (shape1 instanceof Circle) {
            Circle c = (Circle)shape1;
            System.out.println("Radius: " + c.getRadius());
```

```
        }

        if (shape1 instanceof Rectangle) {
            Rectangle r = (Rectangle)shape1;
            System.out.println("Width: " + r.getWidth());
        }
    }
}
```