Lecture Notes

Lecture Notes – Inheritance and Polymorphism

With the use of inheritance and polymorphism, programmers can reuse lot of existing code, thereby avoiding code duplication and saving on development time and effort.

Inheritance

Let’s look at an example of classes by implementing a Rectangle class.

```
class Rectangle {
  protected final float length ;
  private final float breadth ;
  public Rectangle ( float l, float b) {
      this . length = l;
      this . breadth = b;
  }
  public float area () {
      return this . length * this . breadth ;
  }
}
```

Figure 1: A rectangle class

The code in figure 1 can be tested as follows:

```
public class Geometry2 {
  public static void main(String[] a) {
      Rectangle r = new Rectangle(10, 20);
      System.out.println(" area = " + r.area());
  }
}
```

Next, let's also add a Square class to the code.

```
class Square {
  private final float length;

  public Square(float l) {
      this.length = l;
  }

  public float area() {
      return this.length * this.length;
  }
}
```

Figure 2: A square class

The code in figure 2 can be tested as follows:

```
public class Geometry2 {
  public static void main(String[] a) {
      Rectangle r = new Rectangle(10, 20);
      System.out.println(" area = " + r.area());
      Square s = new Square(200);
      System.out.println(" area = " + s.area());
  }
}
```

The Rectangle and Square classes _g. 1 and _g. 2 aren't similar by co-incidence. The fact is, a square is a rectangle with its length equal to its breadth. Unfortunately, the code here doesn't capture this fact. It would be nice if we could make this knowledge an explicit part of our code. Does Java allow us to do that? Yes, it does!

```
class Square extends Rectangle {
  private final float length;

  public Square(float l) {
      this.length = l;
  }

  public float area() {
      return this.length * this.length;
  }
}
```

Figure 3: Square class declared a sub-class of Rectangle

The code in figure 3 modifies that in figure 2 by declaring Square as a sub-class of Rectangle, by using the extends keyword as shown. This is good. But it doesn't do much functionally. However, following version of the Square class does the real magic!

```
class Square extends Rectangle {
  public Square(float l) {
      super(l, l);
  }
}
```

Figure 4: Square class declared a sub-class of Rectangle

Several lines from the Square have been reduced. Does this code even compile? Sure enough, it does! In particular, the call to s.area() from the main method compiles in spite of there being no area method anymore in the Square class. And if you run the code, it seems to work just as _ne as before, giving the same result. How could this happen?

The reason for this is: Square being the child of the Rectangle class, inherits all its properties. This is called inheritance, the most important feature of all object oriented programming languages. We say that:

- Square inherits from Rectangle.

- Square is the sub-class/child-class/sub-type/child-type/derived-class of Rectangle.
- Rectangle is the super-class/parent-class/super-type/parent-type of Square.

Thus, area method in Rectangle also becomes a property of Square. But this area method needs the length and breadth attributes of Rectangle to be set to the correct values (in this case, equal to the length attribute (now deleted) of the Square class). Where and how does this happen?

It happens in the constructor of Square class, when super(l, l) executes. To understand what this does observe figure 5

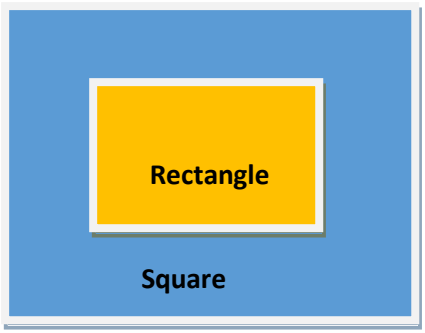

Figure 5: Square object with an object of Rectangle embedded within itself

Each object of a sub-class can be visualised as embedding within it an object of its super-class. Thus, an object of Square class has within it an object of Rectangle as shown in figure 5. During the construction of a sub-class object, the first step to complete is the construction of the embedded super-class object. The call to super method does precisely this. It calls the constructor of the super-class with the given arguments, in this case l and l (l being the parameter to Square's constructor).

This initialises the embedded Rectangle to have its length and breadth attributes both set to the argument passed to the constructor to Square. Thus, a subsequent call to s.area() in the main method calls the area method of the embedded Rectangle object. This, in turns returns the product of length and breadth attributes (which, remember, are equal to each other) thus giving us the correct area of the Square.

So, super is a new keyword we have learned, it used in the context of a sub-class, is essentially a reference to the embedded instance of the superclass.

Can we derive further classes from Square? Yes, and we present an example in figure 6.

```
class Point extends Square {
  public Point() {
      super(0);
  }
```

Figure 6: Point class declared as sub-class of Square

Again, a very rudimentary class, with hardly any code! It just creates the new type Point as a sub-type of Square, making point a special type of square with zero length, which is indeed a reasonable way to look at things.

The code in figure 6 can be tested with the following lines added to the main method:

```
Square p = new Point(
 System.out.println(" area = "+p.area());
```

This works just smoothly, giving us the expected results:

```
area = 0.0
```

Indeed, a point is a square (and hence a rectangle) with zero area. Pictorially, the scenario can be depicted as in figure 7.

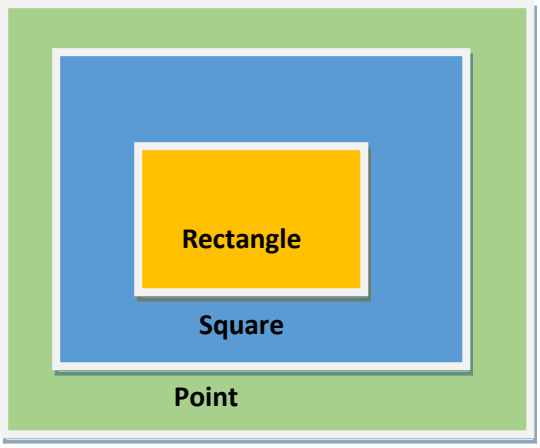

Figure 7: Point object with an object of Square embedded within itself

Protected Access Specifier

Any method from within Square class won't be able to access Rectangle's length attribute directly. This may be OK sometimes, but sometimes this may be too restrictive. For example, the following piece of code

```
public float circumference(){
   return 4.0 f*this.length;
   }
```

if added to the Square class would lead to a compilation error:

```
error : length has private access in Rectangle
return 4.0 f * this . length ;
```

In other words, the designer of the Rectangle class may want the implementers of its sub-classes to have direct access to length. One option would be to turn length into a public attribute. This would work, but this is an overkill, and too permissive. We would like to tune the visibility of length to just the level where the sub-classes have direct access to it, but it remains invisible to any other class in the program. For this we use the protected access specifier:

protected float length;

With this, it is possible to write code within Square class that directly refers to length. For instance, the line of code above, when added to Square class, will not give any error.

INHERITANCE AND POLYMORPHISM- I

To summarise:

- Inheritance is a powerful feature to avoid code duplication
- Whenever there is a common code, group them into super and subclasses.
- Use the extends keyword for inheriting features of a superclass into a subclass
- In order to restrict access of class members to only subclasses, use the protected modifier.

Polymorphism

Consider the modified main method shown in figure 8. The notable point here is that the variable r, which is of the type Rectangle is first initialised to a Rectangle, which is familiar. However, subsequently, we assign to it an instance of a Square, and then a Point. We print the area in each case

```
public static void main(String[]a){
   Rectangle r=new Rectangle(10,20);
   System.out.println(" area = "+r.area());
   r=new Square(10);
   System.out.println(" area = "+r.area());
   r=new Point();
   System.out.println(" area = "+r.area());
      }
```

Figure 8: Square class declared a sub-class of Rectangle

The output of running the code in figure 8 is as shown below:

area = 200.0

area = 100.0

area = 0.0

And look, the area gets printed correctly for all the three shapes.

At this point, a slight refinement of terminology. Here, rather than thinking of r as a variable of type Rectangle, it's more proper to think of it as a reference of the type Rectangle. This means that it can point to an object of the type Rectangle. What we observe further in the code in figure 8 is that it is allowed for r to point to any object whose type is a sub-type of Rectangle.

In fact, references in Java are called polymorphic. And this property of a language which implements polymorphic references in the above sense is called polymorphism, more precisely, dynamic polymorphism. We discuss the meaning of this term a little later. But let's try to appreciate what this feature can do for us. Consider the modified driver code shown in figure9.

The main calls another method printRectangles. As argument, it passes an array of Rectangles constructed out of three Rectangles: r (indeed a Rectangle), s (actually, a Square) and p (which is in fact a Point).

Firstly, note that Java allows us to construct an array of Rectangles, wherein the elements can be objects of any sub-class of Rectangle. Secondly, passing this array to printRectangles gives us just the expected output: the areas of all the Rectangles in the array getting printed.

What does this mean? This means that printRectangles method couldn't care less what the precise type of the objects in the rarray array are. The Java type system assures that they all are instances of Rectangle or one of its sub-classes. In fact, there's no need for Square and Point classes to even exist at the time of implementing printRectangles. Even if these classes are implemented afterwards, much after the time printRectangles is implemented, everything here is guaranteed to work perfectly.

```
public static void main(String[]a){
   Rectangle r=new Rectangle(10,20);
   Square s=new Square(200);
   Square p=new Point();
   Rectangle[]rarray={r,s,p};
   printRectangles(rarray);
    }
public static void printRectangles(Rectangle[]rarray){
    for(Rectangle rec:rarray){
        System.out.println(" area = "+rec.area());
      }
     }
```

Figure 9: printRectangles method prints an array of Rectangles

The above idea is not new, but has existed for a long time in engineering. Wherever there is a system with components that interact and interoperate, engineers go about designing them by defining what we call interfaces. Consider the USB port, the VGA, power audio ports of your computer. As long as a VGA cord following the specifications of VGA is inserted into your computer's VGA port, it is kind of guaranteed to work. It doesn't matter who manufactured the VGA cord. Similarly, the Android OS can be installed on any Android compatible device. It could be any of hundreds of phone brands, it could be a tablet, a PC, a TV or anything else. Are these devices identical? No. But they follow the interfaces specified by the creators of Android OS. Internally, each one of them may have many variations, but Android doesn't concern itself with them.

Similarly, the super-class (here, Rectangle) is kind of an interface which the printRectangles method accepts. The inheritance rules of Java guarantee that all sub-classes of Rectangle adhere to its interface, i.e. if area method is called on them, it will be available. And therefore, printRectangle is able to work with any array of Rectangles, even when it may actually contain objects of other types. All that's needed is those other types must be sub-types of Rectangle. Java's type-system makes sure that requirement is fulfilled: an attempt to populate a Rectangle[] array with an object of a type which isn't a sub-type of Rectangle will fail at compile-time.

In a short while, we will have a bit more to say about interfaces, which are a very important concept in Java and OOP in general.

Method Overriding

What we have learned so far about inheritance is good to create sub-classes which are specialisations of their super-classes. In other words, they are the same as their super-classes, but for some additional constraints. This is useful, but not useful enough. Often there are situations when we wish to modify our super-classes as per need. I will present here a simple example.

Let's add a method printName in the rectangle class:

```
public void printName(){
    System.out.println(" I 'm a rectangle . ");
      }
```

Now, let's call this function from the main for all the Rectangles we have created there.

```
Rectangle[]rectangles={r,s,p};
   for(Rectangle rec:rectangles){
       rec.printName();
      }
```

This will produce the following output:

I'm a rectangle .

I'm a rectangle .

I'm a rectangle .

This output is technically correct, but not interesting. It would be nice if we could print the correct name as per the sub-class. For example, for a Square, the message should be "I'm a square.". Is it possible to have this output?

Given the fact that in the context of main, each shape is being accessed through a Rectangle type reference, this looks unlikely. Nevertheless, let's go ahead and implement the methods that we would have liked to be called to print the correct shape names.

In Square class, we add:

```
public void printName(){
   System.out.println(" I 'm a square . ");
      }
```

In Point class, we add:

```
public void printName(){
      System.out.println("I'm a point .");
      }
```

Let us compile the code and run it:

I'm a rectangle .

I'm a square .

I'm a point .

As you can see, the program works as expected.For each object, the version of printName as

defined in the sub-class was called. Indeed, that's what happened. Even though printDetails is called from the context of main, with a reference to the Rectangle class, the implementations in the sub-classes are called. In fact, it's quite allowed to implement the main method in a separate source file, and compile it even before the sub-classes like Square and Point are written. These can be written and added to the program later, and yet, everything would work seamlessly. This feature is realised with a mechanism called **dynamic dispatch.**

To make things further interesting, let's implement a method named printDetails in the Rectangle class along with the main method as shown in figre 10.

```
public void printDetails(){
      this.printName();
      System.out.println("... and my area is "+this.area());
      }
```

Figure 10: printDetails method prints the details of the object.

Note that printDetails has a call to this.printName. If this method is called on a sub-class of Rectangle using a Rectangle reference, which version of printName would be called? Rectangle's or the sub-class's? To test, we also modify the driver code as follows:

```
Rectangle r=new Rectangle(10,20);
      Square s=new Square(200);
      Square p=new Point();
      Rectangle[]rectangles={r,s,p};
      for(Rectangle rec:rectangles){
      rec.printDetails();
      }
      }
```

When we compile and run the modified code, we get the following output:

```
I'm a rectangle .

... and my area is 200.0

I'm a square .

... and my area is 40000.0

I'm a point .

... and my area is 0.0
```

As you can observe, the correct versions of printName gets called from within Rectangle.printDetails.

This is one of the most powerful features of object-oriented programming and should be mastered well by an object-oriented programmer. Let's say, in a class C1 a method m1 calls another m2 in the same class, which has implementations in sub-class C2. Now, if a reference r points to an instance of C1 and a call r:m1 is called. Internally, r:m1 will call C2:m2. Because of this, we say that C2:m2 overrides C1:m2.

Method Overloading

Another type of polymorphism is known as static or compile time polymorphism. This is implemented with the help of method overloading. Whenever methods with similar name are defined in the same class or an immediate subclass, they can be made to perform differently.

Each version of a method in this case, must have a different argument list. The argument list can be made different either by changing the number of arguments or the type of arguments. In the code given below, the area() method based on the argument passed to it during run time, will invoke the corresponding version of the method. Thus, using the same method, you can calculate area for both rectangle and a square

```
class Shape {
  void area(double length) {
      System.out.println("Area of Square is: " + Math.pow(length, 2));
  }

  void area(double length, double breadth) {
      System.out.println("Area of Rectangle is: " + (length * breadth));
  }
```

Figure 11: area( ) method overloaded for square and rectangle

Abstract Classes

Let's add another class Circle into our family of classes as shown in figure 12.

```
class Circle {
  public static final float PI = 3.141
  private float radius;
  f;

  public Circle(float r) {
      radius = r;
  }
}
```

Figure 12: Circle class.

If we try to instantiate a Circle in the main and try to refer to it using an Rectangle reference, this will lead to a compilation error: the types simply don't match.

The first solution is very simple. Define a class Shape as the superclass of both Circle and Rectangle. Let's do it.

```
class Shape {
  public printName() {
      "I'm a shape .";
  }

  public float area() {
      return 0;
  }

  public void printDetails() {
      this.printName();
      System.out.println("... and my area is " + this.area());
  }
}

class Circle extends Shape {
...

  class Rectangle extends Shape {
```

Figure 13: Shape class

In Shape, we provide a default definition of the area method and printName methods. Note that these implementations don't make much sense. The printName method doesn't provide adequate information about the Shape, and area method would simply give a wrong result for all but Shapes with zero area. Nevertheless, they are needed. Otherwise, the code will not compile.

The main methods gets modified as follows to accommodate the above

```
public static void main(String[] a) {
          Rectangle r = new Rectangle(10, 20);
          Square s = new Square(200);
          Square p = new Point();
          Circle c = new Circle(10);
          Shape[] shapes = {r, s, p, c};
          for (Shape sh : shapes) {
              sh.printDetails();
          }
      }
```

When we compile and run the above code, we get the following output:

I'm a rectangle .

... and my area is 200.0

I'm a square .

... and my area is 40000.0

I'm a point .

... and my area is 0.0

I'm a shape .

... and my area is 0.0

Note that the outputs corresponding to the Circle c are wrong. In case of the Circle, it was the Shape's implementation of printName and area that gets called. This is not merely undesirable, but completely wrong.

On looking at the problem a little more closely, we realise the following:

- We have forgotten to implement printName and area methods in Circle class.
- The default implementation of printName and area methods provided in Shape class are really unnecessary, and are doing more harm than good. They have been put there just to satisfy the compiler (which is a very bad reason to implement a method). Not surprisingly, they are becoming the source of a bug which could be quite hard to detect.

What's the nature of this bug? This bug happens whenever there are default implementation of methods provided in the super-class which are necessarily supposed to implemented in the sub-classes, and somehow the implementer of the sub-class forgets to provide one. What we really want is:

1. No enforcement to provide useless dummy implementations of methods in the super-classes just to appease the compiler.
2. If we forget to implement such methods in the sub-classes, the compiler should alert us of our mistake.

Can we design our code to get the above? Yes, we can. By using abstract classes. To full condition 1 above, we make the following change to the Shape class:

```
abstract class Shape {
    public abstract void printName();
    public abstract float area();
    public void printDetails() {
         this.printName();
         System.out.println("... and my area is " + this.area());
       }
    }
```

Figure 14: Abstract Shape class

The modified code in figure 14 declares Shape as an abstract class: a class with one or more abstract methods. Abstract methods are methods which have a declaration in the class, but have not been implemented. In figure 14,printName and area methods are abstract methods of Shape.

A very important characteristic of abstract classes is that they can't be instantiated directly. That is, an attempt to have something like **Shape s = new Shape()** in the program would not be accepted by the compiler. The only way to instantiate abstract classes is through their concrete (which are not abstract themselves) sub-classes.

Let us try to compile the above code. Given below is the output:

error : Circle is not abstract and does not override

abstract method area () in Shape

class Circle extends Shape {

^

This says that Circle, which is not declared abstract doesn't provide necessary implementation for the abstract methods of its super-class Shape. This takes care of condition 2 above: the compiler has pointed us out our mistake of having forgotten to provide implementations for printName and area methods in Circle class.

To correct this, we add the implementations of printName and area methods in the Circle class as shown in figure 15.

```
class Circle extends Shape {
   private float radius;
   public static final float PI = 3.141
      public Circle(float r) {
         radius = r;
          }

      public void printName() {
         System.out.println("I'm a circle .");
          }

       public float area() {
          return PI * radius * radius;
          }
}
```

Figure 15: Circle class with printName and area methods implemented

Now the code compiles and on running it, we get the following output:

I'm a rectangle .

... and my area is 200.0

I'm a square .

... and my area is 40000.0

I'm a point .

... and my area is 0.0

I'm a circle .

... and my area is 314.1

Note that the errors in the last two lines of the output have now been corrected.

Interfaces

A possible design of the Shape class would to make it completely abstract, i.e. when all its methods are abstract, as shown in figure 16.

```
abstract class Shape {
   public void printName();
   public float area();
      }
```

Figure 16: A completely abstract Shape class.

Java presents another syntactic way to define such completely abstract classes. They are called interfaces, as shown in figure 17.

```
interface Shape {
  void printName();
  float area();
      }
class Circle implements Shape {
...
      }

class Rectangle implements Shape {
...
      }
```

Figure 17: Shape as an interface.

Interfaces are more significant than just being able to build completely abstract classes. An interface is not extended, but implemented, by its sub-classes, as shown in the modified code for Circle and Rectangle.

INHERITANCE AND POLYMORPHISM- II

To summarise:

- Polymorphism is an important feature of object oriented programming, implemented with the help of method overriding and method overloading
- Static Polymorphism is implemented with the help of method overloading
- Dynamic Polymorphism is implemented with the help of method overriding
- Abstract classes and Interfaces are helpful in cases, when we want to leave the implementation details for individual classes that inherit or implement them.