Week – 08

INHERITANCE: Extending a Class

Inheritance in java is a mechanism in which one object acquires all the properties and behaviours of parent object.

Advantages of Inheritance

- **Minimizing duplicate code:** Key benefits of Inheritance include minimizing the identical code as it allows sharing of the common code among other subclasses.
- **Flexibility:** Inheritance makes the code flexible to change, as you will adjust only in one place, and the rest of the code will work smoothly.
- **Overriding:** With the help of Inheritance, you can override the methods of the base class.
- **Data Hiding:** The base class in Inheritance decides which data to be kept private, such that the derived class will not be able to alter it.

Types of Inheritance

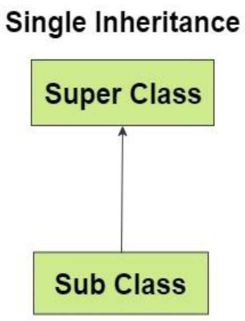
There are different forms of inheritance in java:

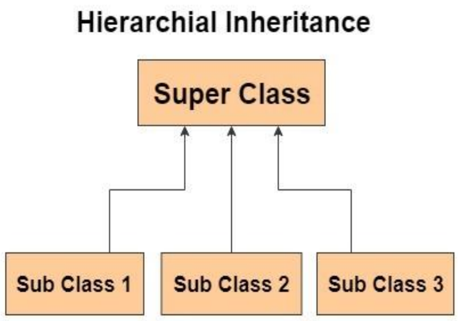
- Single Inheritance [only one super class]
- Multiple Inheritance [several super class]
- Multilevel Inheritance [Derived from derived class]
- Hierarchical Inheritance [one super class many subclass]

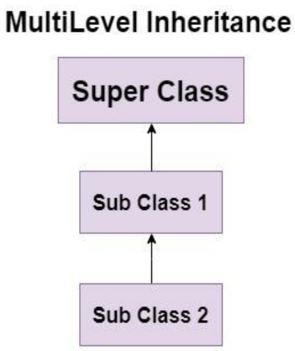
NOTE :Multiple inheritance is not supported in java, however the concept is implemented using a secondary inheritance path in the form of "interfaces"

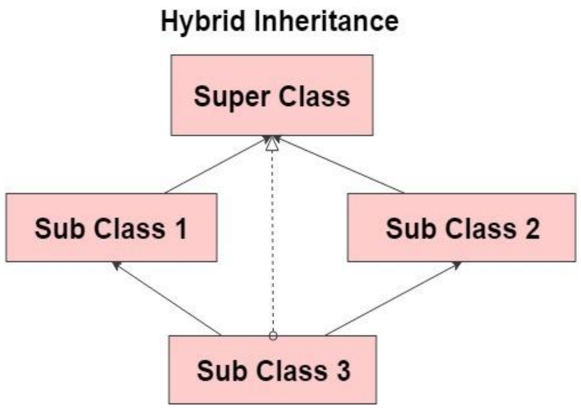
```
class Subclass-name extends Superclass-name
{
  Variable declaration;
  Method declaration;
}
```

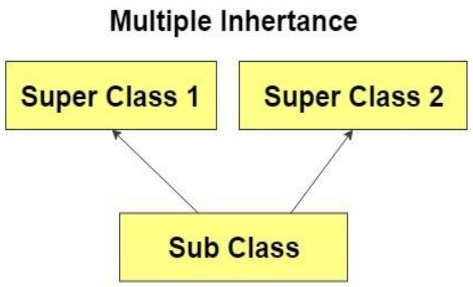
```
class A extends B {

    // Here Class A will inherit the features of Class B

  }
```











Example: Single Inheritance [simple inheritance]

```
class Parent
{
    public void pMethod()
    {
        System.out.println("Parent method");
    }
}

class Child extends Parent
{
    public void cMethod()
    {
        System.out.println("Child method");
    }
    public static void main(String[] args)
    {
        Child cobj = new Child();
        cobj.cMethod();   //method of Child class
        cobj.pMethod();   //method of Parent class
    }
}
```

Example: Multilevel Inheritance

```
class GrandParent
{
   public void gMethod() {  System.out.println("GrandParent method");    }
}
class Parent`extends GrandParent
{
   public void pMethod()  { gMethod();
      System.out.println("Parent method");     }
}
class Child extends Parent
{
   public void cMethod() { pMethod();
      System.out.println("Child method");     }
}
class MultilevelDemo
{
   public static void main(String[] args)
   {
      Child cobj = new Child();
      cobj.cMethod();   //method of Child class
      cobj.pMethod();   //method of Parent class
      cobj.gMethod();   //method of GrandParent class
   }
}
```

Example: Hierarchical Inheritance

```
class A
{
   public void aMethod() {  System.out.println("I am in A ");    }
}
class B`extends A
{
   public void bMethod()  { System.out.println("I am in B");     }
}
class C extends A
{
   public void cMethod() { System.out.println("I am in C");     }
}
class HierarchicalDemo
{
   public static void main(String[] args)
   {
      C cobj = new C();
      cobj.cMethod();
      cobj.aMethod();
      // cobj.bMethod();    gives error
   }
}
```

Open Closed principle (OCP)

Intent/Definition

Software entities like classes, modules, and functions should be open for extension but closed for modifications.

In object-oriented programming, the open/closed principle states "software entities (classes, modules, functions, etc.) should be open for extension but closed for modification" that is, such an entity can allow its behaviour to be extended without modifying its source code.

Rules of Thumb

- Open to an extension - you should design your classes so that new functionality can be added as new requirements are generated.
- Closed for modification - Once you have developed a class you should never modify it, except to correct bugs.
- Design and code should be done in a way that new functionality should be added with minimum or no changes in the existing code
- When needs to extend functionality - avoid tight coupling, don't use if-else/switch-case logic, do code refactoring as required.
- Techniques to achieve - Inheritance, Polymorphism, Generics

Benefit

The benefit of this Object-oriented design principle is, which prevents someone from changing already tried and tested code.

Example of Open/Closed Principle in Java

Let’s say we need to calculate areas of various shapes.

```
class Rectangle
{
 public double length;
 public double width;
}

 class Circle
{
       public double radius;
}
 class AreaCalculator
{
        public double calculateRectangleArea(Rectangle rectangle)
         {
          return rectangle.length *rectangle.width;
         }
          public double calculateCircleArea(Circle circle)
          {
              return (3.147*circle.radius*circle.radius);

        }
}
public class NoOCPDemo {
public static void main(String args[])
{
      AreaCalculator a1 = new AreaCalculator();
      Rectangle r = new Rectangle();
      r.length=10;
      r.width=20;
      Circle c= new Circle();
      c.radius=10;
      double r1= a1.calculateRectangleArea(r);
      double c1= a1.calculateCircleArea(c);
    System.out.println("----------OUTPUT--------");
    System.out.println("Area of Rectangle="+r1);
    System.out.println("Area of Circle="+c1);
}
}
```

```
----------OUTPUT--------
Area of Rectangle=200.0
Area of Circle=314.7
```

However, note that there were flaws in the way we designed our solution above
Let's say we have a new shape pentagon next. In that case we will again end up modifying **AreaCalculator** class. As the types of shapes grows this becomes messier as **AreaCalculator** keeps on changing and any consumers of this class will have to keep on updating their libraries which contain **AreaCalculator.**
Also, **note that this design is not extensible**, i.e. what if complicated shapes keep coming, **AreaCalculator** will need to keep on adding their computation logic in newer methods. We are not really expanding the scope of shapes;

Modification of above design to comply with Open/Closed Principle

```
interface Shape
{
  public double calculateArea();
}

class Rect implements Shape
{
  double length;
  double width;
  public double calculateArea(){
    return length * width;
  }
}

class Cir implements Shape
{
  public double radius;
  public double calculateArea(){
    return (3.147*radius*radius);
  }
}
class Square implements Shape
{
      double l;
      public double calculateArea() {
             return (l*l);
             }
}
```

```
 class AreaCal
  {
         public double calculateShapeArea(Shape shape){
           return shape.calculateArea();
         }
       }

public class OCPDemo {
       public static void main(String args[])
       {
             AreaCal a1 = new AreaCal();
             Rect r = new Rect();
             r.length=10;
             r.width=20;
             Cir c= new Cir();
             c.radius=10;
             Square s = new Square();
             s.l=30;
             double r1= a1.calculateShapeArea(r);
             double c1= a1.calculateShapeArea(c);
             double s1=a1.calculateShapeArea(s);
           System.out.println("----------OUTPUT--------");
           System.out.println("Area of Rectangle="+r1);
           System.out.println("Area of Circle="+c1);
           System.out.println("Area of Square="+s1);
       }
}
```

```
----------OUTPUT--------
Area of Rectangle=200.0
Area of Circle=314.7
Area of Square=900.0
```

The design is now correct as per Open Closed Principle due to the following reasons –

- **The design is open for extension** as more shapes can be added without modifying the existing code. We just need to create a new class for the new shape and implement the **calculateArea()** method with a formula specific to that new shape.
- **This design is also closed for modification. AreaCal** class is complete w.r.t area calculations. It now caters to all the shapes which exists now, as well as to those that may be created later.

Thus the above program Satisfies OCP Principle.