Encapsulation & Inheritance

What is Encapsulation?

- Encapsulation is the technique of making the fields in a class private and providing access to the fields via public methods.
- If a field is declared private, it cannot be accessed by anyone outside the class, thereby hiding the fields within the class. For this reason, encapsulation is also referred to as data hiding.
- To achieve encapsulation in Java −
    1. Declare the variables of a class as private.
    2. Provide public setter and getter methods to modify and view the variables values.

Encapsulation Example:

```
public class EncapTest{
        private String name;
        private String idNum;
        private int age;
        public int getAge(){          return age;     }
        public String getName(){    return name;    }
        public String getIdNum(){    return idNum;    }
        public void setAge(int age){
                this.age = age;}
        public void setName(String name){
                this.name = name;}
        public void setIdNum(String idNum){
                this.idNum = idNum;}
}
```

Inheritance

- Inheritance is the process by which object of one class acquires the properties of another class.
- A class that is inherited is called a **superclass**. The class that does the inheriting is called a **subclass.**
- Therefore, a **subclass** is a specialized version of a **superclass**. It inherits all of the instance variables and methods defined by the **superclass** and add its own, unique elements.

- To inherit a class, you simply incorporate the definition of one class into another by using the **extends** keyword.
- The general form of class declaration that inherits a superclass is shown here.

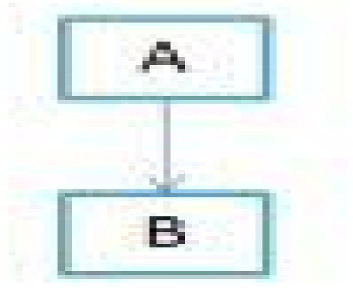
```
class subclass-name extends superclass-name
{
// body of class.


}
```

Types of inheritance

1. Single inheritance
2. Multilevel inheritance
3. Multiple inheritance
4. Hybrid inheritance
5. Hierarchical inheritance

Single Inheritance

- **Single inheritance:** When a class extends another class only then we call it a single inheritance.
- The below flow diagram shows that class B extends only one class which is A. Here A is a **parent class** of B and B would be a **child class** of A.



(a) Single Inheritance

Single Inheritance Example:

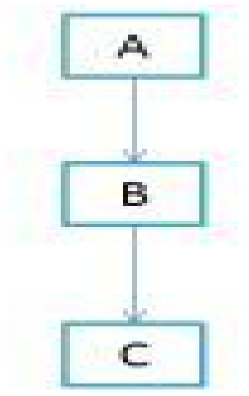
```
class A {
public void methodA() {
System.out.println("Base class method");   }
 }
class B extends A {
public void methodB() {
System.out.println("Child class method");
}
public class Inherent{
public static void main(String args[]) {
B obj = new B();
obj.methodA(); // calling super class method
obj.methodB(); // calling local method
}  }
```

Output is:

```
Base class method
Child class method
```

Multilevel Inheritance

- **Multilevel inheritance** refers to a mechanism in OO technology where one can inherit from a derived class, thereby making this derived class the base class for the new class.
- As you can see in below flow diagram C is subclass or child class of B and B is a child class of A.



(d) Multilevel Inheritance

Multilevel Inheritance Example:

```
class X {
public void methodX() {
System.out.println("Class X method");
 }
}
class Y extends X {
public void methodY() {
System.out.println("class Y method");
}
}
```

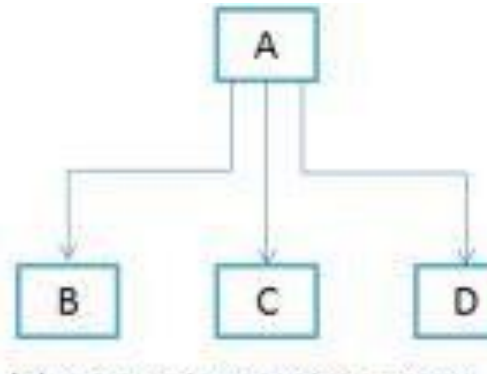
```
class Z extends Y {
public void methodZ() {
System.out.println("class Z method");
}
public static void main(String args[]) {
   Z obj = new Z();
  obj.methodX();      // calling grand parent class method
  obj.methodY();        // calling parent class method
  obj.methodZ();        // calling local method
     }
}
```

Output is:

```
Class X method
class Y method
class Z method
```

Hierarchical Inheritance

- In such kind of inheritance one class is inherited by many **sub classes**.
- In below example class B,C and D **inherits** the same class A. A is **parent class (or base class)** of B,C & D.



(c) Hierarchical Inheritance

Hierarchical Inheritance Example:

```
class A {
       public void methodA() {
          System.out.println("method of Class A");
   }   }
class B extends A {
       public void methodB() {
         System.out.println("method of Class B");
   }    }
class C extends A {
        public void methodC() {
        System.out.println("method of Class C");
}   }
```

```
class D extends A {
      public void methodD() {
           System.out.println("method of Class D");
   } }
class MyClass {
public static void main(String args[]) {
     B obj1 = new B();
    C obj2 = new C();
     D obj3 = new D();
    obj1.methodA();
    obj2.methodA();
    obj3.methodA();
}    }
```

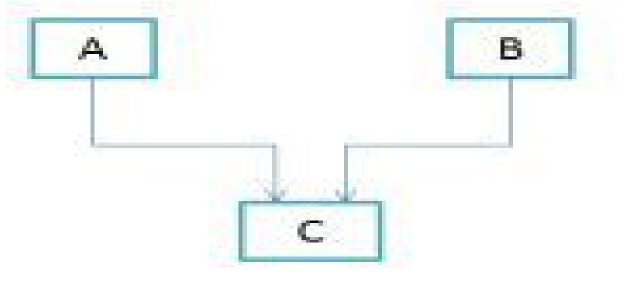
Output is:

```
method of Class A
method of Class A
method of Class A
```

Multiple Inheritance

- "**Multiple Inheritance**" refers to the concept of one class extending (Or inherits) more than one base class.
- The inheritance we learnt earlier had the concept of one base class or parent. The problem with "multiple inheritance" is that the derived class will have to manage the dependency on two base classes.

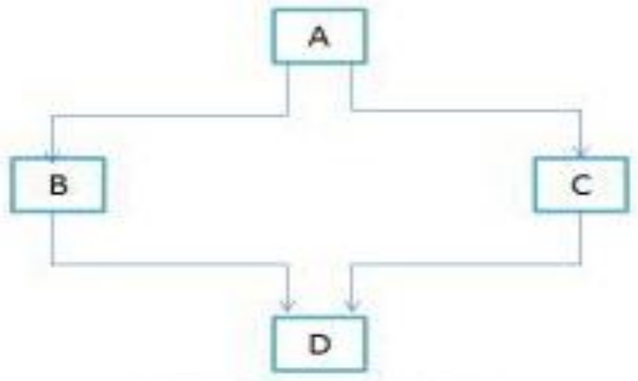
- Note 1: Multiple Inheritance is very rarely used in software projects. Using Multiple inheritance often leads to problems in the hierarchy. This results in unwanted complexity when further extending the class.
- Note 2: Most of the new OO languages like **Small Talk, Java, C#** do not support Multiple inheritance. Multiple Inheritance is supported in C++.



(b) Multiple Inheritance

Hybrid Inheritance

- In simple terms you can say that Hybrid inheritance is a combination of **Single** and **Multiple inheritance.**
- A typical flow diagram would look like below. A hybrid inheritance can be achieved in the java in a same way as multiple inheritance can be!!
- Using interfaces. yes you heard it right. By using **interfaces** you can have multiple as well as **hybrid inheritance** in Java.



(e) Hybrid Inheritance

Hybrid Inheritance Example: show error

```
public class A{
     public void methodA()     {
           System.out.println("Class A methodA");
      }
}
public class B extends A{
      public void methodA()      {
  System.out.println("Child class B is overriding
inherited  method A");
      }
      public void methodB()      {
           System.out.println("Class B methodB");
       }
}
```

```
public class C extends A{
      public void methodA()      {
        System.out.println("Child class C is overriding the
                                    methodA");
       }
      public void methodC()      {
          System.out.println("Class C methodC");
       }
}
public class D extends B, C{
      public void methodD()      {
          System.out.println("Class D methodD");
       }
       public static void main(String args[])      {
             D obj1= new D();
             obj1.methodD();
             obj1.methodA();
       }  }
```

- **Why it is not allowed?**
- Most of the times you will find the following explanation of above error – Multiple inheritance is not allowed in java so class D cannot extend two classes(B and C).
- **But do you know why it's not allowed?**
- Let's look at the above code once again, In the above program class B and C both are extending class A and they both have overridden the methodA(), which they can do as they have extended the class A.
- But since both have different version of methodA(), **compiler is confused** which one to call when there has been a call made to methodA() in child class D (child of both B and C, it's object is allowed to call their methods),
- This is a ambiguous situation and to avoid it, such kind of scenarios are not allowed in java. In C++ it's allowed.

Are superclass’s Constructor Inherited?

- No. They are not inherited.
- They are invoked explicitly or implicitly.
- Explicitly using the super keyword.

- A constructor is used to construct an instance of a class. Unlike properties and methods, a superclass's constructors are not inherited in the subclass.
- They can only be invoked from the subclasses' constructors, using the keyword super. If the keyword super is not explicitly used, the superclass's no-arg constructor is automatically invoked.

Using the Keyword super

The keyword super refers to the superclass of the class in which super appears. This keyword can be used in two ways:

- To call a superclass constructor
- To call a superclass method

Constructors and Inheritance

- The Constructor in the superclass is responsible for building the object of the superclass and the constructor of the subclass builds the object of subclass.
- When the subclass constructor is called during object creation, it by default invokes the default constructor of super-class. Hence, in inheritance the objects are constructed top-down.
- The superclass constructor can be called explicitly using the keyword super, but it should be first statement in a constructor.
- The keyword super always refers to the superclass immediately above of the calling class in the hierarchy. The use of multiple super keywords to access an ancestor class other than the direct parent is illegal.

Constructors and Inheritance Example:

```
class Shape {
      private int length;
      private int breadth;
    Shape() {
            length = 0;
            breadth = 0;
     System.out.println("Inside default constructor of Shape ");
  }  }
class Rectangle extends Shape {
      private String type;
     Rectangle() {
      super();          //Would have been invoked anyway
     type = null;
System.out.println("Inside default constructor of rectangle ");
  }   }
```

```
class ColoredRectangle extends Rectangle {
     private String color;

 ColoredRectangle() {
        super();        // //Would have been invoked
anyway
       color = null;
System.out.println("Inside default constructor of
                                        coloredRectangle");
   }
}
public class Test {
       public static void main(String args[]) {

ColoredRectangle CR = new ColoredRectangle();
}
}
```

Output is:

```
Inside default constructor of Shape
Inside default constructor of rectangle
Inside default constructor of coloredRectangle
```

```
class Shape {
      public int length;
      public int breadth;
       Shape(int len, int bdth) {
             length = len;
             breadth = bdth;
  System.out.println("Inside constructor of Shape ");
  System.out.println("length : " + length);
  System.out.println("breadth : " + breadth);
}  }
```

```
class Rectangle extends Shape {
      protected String type;
       Rectangle(String ty, int len, int bdth) {
          super(len, bdth);      //must needed
          type=ty;
       System.out.println("Inside constructor of rectangle ");
       System.out.println("length : " + length);
       System.out.println("breadth : " + breadth);
       System.out.println("type : " + type);
}}
```

```
public class Test {
      public static void main(String args[]) {

       ColoredRectangle CR2 = new

                   ColoredRectangle("Red", "Big", 5, 2);
  }
}
```

```
class ColoredRectangle extends Rectangle {
        private String color;

      ColoredRectangle(String c, String ty, int len, int bdth) {
          super(ty, len, bdth);         //must needed
          color=c;
   System.out.println("Inside constructor of coloredRectangle ");
   System.out.println("length : " + length);
   System.out.println("breadth : " + breadth);
   System.out.println("type : " + type);
   System.out.println("color : " + color);
} }
```

Output is :

```
Inside constructor of Shape
length : 5
breadth : 2

Inside constructor of rectangle
length : 5
breadth : 2
type : Big

Inside constructor of coloredRectangle
length : 5
breadth : 2
type : Big
color: Red
```