Inheritance I

Inheritance is an important pillar of OOP(Object Oriented Programming). It is the mechanism in Java by which one class is allow to inherit the features(fields and methods) of another class.

Important terminology:

- Super Class: The class whose features are inherited is known as super class(or a base class or a parent class).
- Sub Class: The class that inherits the other class is known as sub class(or a derived class, extended class, or child class). The subclass can add its own fields and methods in addition to the superclass fields and methods.

Inheritance II

- Reusability: Inheritance supports the concept of ”reusability”, i.e. when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class.

How to use inheritance in Java

The keyword used for inheritance is extends. Syntax :

```
class derived-class extends base-class
{
    //methods and fields
}
```

Inheritance III

Java Object Creation of Inherited Class:

✓ In inheritance, subclass acquires super class properties.

✓ An important point to note is, when subclass object is created, a separate object of super class object will not be created.

✓ Only a subclass object object is created that has super class variables.

✓ This situation is different from a normal assumption that a constructor call means an object of the class is created, so we can't blindly say that whenever a class constructor is executed, object of that class is created or not.

Inheritance IV

```
1  // A Java program to demonstrate that both super class and subclass constructors refer to same
       object
2  // super class
3  class Fruit
4  {
5      public Fruit()
6      {
7          System.out.println("Super class constructor");
8          System.out.println("Super class object hashcode :" +this.hashCode());
9          System.out.println(this.getClass().getName());
10     }
11 }
12 // sub class
13 class Apple extends Fruit
14 {
15     public Apple()
16     {
```

Inheritance V

```
17              System.out.println(”Subclass constructor invoked”);
18              System.out.println(”Sub class object hashcode :” +this.hashCode());
19              System.out.println(this.hashCode() + ”” +
20              super.hashCode());
21
22              System.out.println(this.getClass().getName() + ”  ” +super.getClass().getName());
23      }
24  }
25  // driver class
26  public class Test
27  {
28      public static void main(String[] args)
29      {
30          Apple myApple = new Apple();
31      }
32  }
```

Inheritance VI

✓ Example: In below example of inheritance, class Bicycle is a base class, class MountainBike is a derived class which extends Bicycle class and class Test is a driver class to run program.

```
1  //Java program to illustrate the concept of inheritance
2  // base class
3  class Bicycle
4  {
5      // the Bicycle class has two fields
6      public int gear;
7      public int speed;
8      // the Bicycle class has one constructor
9      public Bicycle(int gear, int speed)
10     {
11         this.gear = gear;
12         this.speed = speed;
```

Inheritance VII

```
13      }
14      // the Bicycle class has three methods
15      public void applyBrake(int decrement)
16      {
17          speed -= decrement;
18      }
19      public void speedUp(int increment)
20      {
21          speed += increment;
22      }
23      // toString() method to print info of Bicycle
24      public String toString()
25      {
26          return("No of gears are "+gear
27          +"\n"
28          + "speed of bicycle is "+speed);
29      }
```

Inheritance VIII

```
30 }
31 // derived class
32 class MountainBike extends Bicycle
33 {
34     // the MountainBike subclass adds one more field
35     public int seatHeight;
36     // the MountainBike subclass has one constructor
37     public MountainBike(int gear,int speed,
38     int startHeight)
39     {
40         // invoking base-class(Bicycle) constructor
41         super(gear, speed);
42         seatHeight = startHeight;
43     }
44     // the MountainBike subclass adds one more method
45     public void setHeight(int newValue)
46     {
47         seatHeight = newValue;
```

Inheritance IX

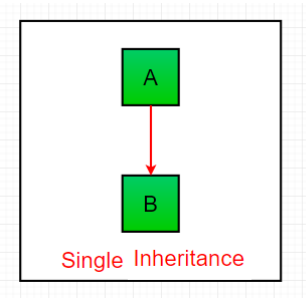
```
48      }
49      // overriding toString() method of Bicycle to print more info
50      @Override
51      public String toString()
52      {
53          return (super.toString()+
54          "\nseat height is "+seatHeight);
55      }
56  }
57  // driver class
58  public class Test
59  {
60      public static void main(String args[])
61      {
62          MountainBike mb = new MountainBike(3, 100, 25);
63          System.out.println(mb.toString());
64      }
65      Output:
```

```
66      No of gears are 3
67      speed of bicycle is 100
68      seat height is 25
```

Types of Inheritance in Java:

1. Single Inheritance : In single inheritance, subclasses inherit the features of one superclass. In image below, the class A serves as a base class for the derived class B.

Inheritance XI



Inheritance XII

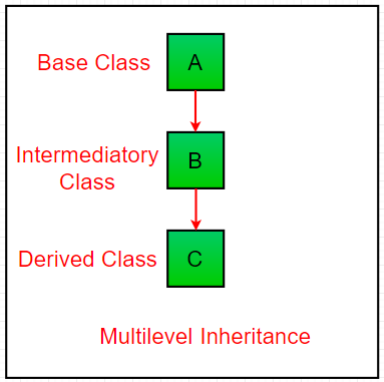
```java
//Java program to illustrate the concept of
    single inheritance
import java.util.*;
import java.lang.*;
import java.io.*;

class one
{
    public void print_MCA()
    {
        System.out.println("MCA");
    }
}
class two extends one
{
    public void print_for()
    {
        System.out.println("for");
    }
}
// Driver class
public class Main
{
    public static void main(String[] args)
    {
        two g = new two();
        g.print_MCA();
        g.print_for();
        g.print_MCA();
    }
}
```

Inheritance XIII

2. Multilevel Inheritance : In Multilevel Inheritance, a derived class will be inheriting a base class and as well as the derived class also act as the base class to other class. In below image, the class A serves as a base class for the derived class B, which in turn serves as a base class for the derived class C. In Java, a class cannot directly access the *grandparent's members*.

Inheritance XIV



Inheritance XV

```java
// Java program to illustrate the concept of
    Multilevel inheritance
import java.util.*;
import java.lang.*;
import java.io.*;
class one
{
    public void print_geek()
    {
        System.out.println("Geeks");
    }
}
class two extends one
{
    public void print_for()
    {
        System.out.println("for");
    }
}
class three extends two
{
    public void print_geek()
    {
        System.out.println("Geeks");
    }
}
// Drived class
public class Main
{
    public static void main(String[] args)
    {
        three g = new three();
        g.print_geek();
        g.print_for();
        g.print_geek();
    }
}
```

Examples on Inheritance I

```
1 class A
2 {
3       int methodOfA()
4       {
5             return (true ? null : 0);
6       }
7 }
```

Examples on Inheritance II

```
1  class ClassOne
2  {
3      static int i = 111;
4
5      int j = 222;
6
7      {
8          i = i++ - ++j;
9      }
10 }
11 class ClassTwo extends ClassOne
12 {
13     {
14         j = i-- + --j;
15     }
16 }
17 public class MainClass
18 {
19     public static void main(String[] args)
20     {
21         ClassTwo b = new ClassTwo();
22     }
23 }
```

Examples on Inheritance III

```
1  class A
2  {
3      static String s = ”AAA”;
4      static
5      {
6          s = s + ”BBB”;
7      }
8      {
9          s = ”AAABBB”;
10     }
11 }
12 class B extends A
13 {
14     {
15         System.out.println(s);
16     }
17     static
18     {
19         s = s + ”BBBAAA”;
20     }
21 }
22 public class MainClass
23 {
24     public static void main(String[] args)
25     {
26         B b = new B();
27     }
28 }
```

Examples on Inheritance IV

```
1  class A
2  {
3      {
4          System.out.println(1);
5      }
6  }
7  class B extends A
8  {
9      {
10         System.out.println(2);
11     }
12 }
```

```
13 class C extends B
14 {
15     {
16         System.out.println(3);
17     }
18 }
19 public class MainClass
20 {
21     public static void main(String[] args)
22     {
23         C c = new C();
24     }
25 }
```

Examples on Inheritance V

```
1  class A
2  {
3      int i = 10;
4  }
5  class B extends A
6  {
7      int i = 20;
8  }
9  public class MainClass
10 {
11     public static void main(String[] args)
12     {
13         A a = new B();
14         System.out.println(a.i);
15     }
16 }
```

Examples on Inheritance VI

```
1  class A
2  {
3      String s = "Class A";
4  }
5  class B extends A
6  {
7      String s = "Class B";
8      {
9          System.out.println(super.s);
10     }
11 }
12 class C extends B
13 {
14     String s = "Class C";
15     {
16         System.out.println(super.s);
17     }
18 }
19 public class MainClass
20 {
21     public static void main(String[] args)
22     {
23         C c = new C();
24         System.out.println(c.s);
25     }
26 }
```

Examples on Inheritance VII

```
1  class A
2  {
3      static
4      {
5          System.out.println("THIRD");
6      }
7  }
8
9  class B extends A
10 {
11     static
12     {
13         System.out.println("SECOND");
14     }
15 }
16 class C extends B
17 {
18     static
19     {
20         System.out.println("FIRST");
21     }
22 }
23 public class MainClass
24 {
25     public static void main(String[] args)
26     {
27         C c = new C();
28     }
29 }
```

Examples on Inheritance VIII

```
1  class X
2  {
3      static void staticMethod()
4      {
5          System.out.println("Class X");
6      }
7  }
8  class Y extends X
9  {
10     static void staticMethod()
11     {
12         System.out.println("Class Y");
13     }
14 }
15 public class MainClass
16 {
17     public static void main(String[] args)
18     {
19         Y.staticMethod();
20     }
21 }
```

Examples on Inheritance IX

```
1  public class A
2  {
3      public A()
4      {
5          System.out.println(1);
6          super();
7          System.out.println(2);
8      }
9  }
10 //Constructor calling statements ( super() or this() ), if written, must be the first statements in the
       constructor.
```

Examples on Inheritance X

```
1  public class A
2  {
3      public A()
4      {
5          super();
6          this(10);
7      }
8      public A(int i)
9      {
10         System.out.println(i);
11     }
12 }
13 //A constructor can have either super() or this() but not both.
```

Examples on Inheritance XI

```
class M
{
    static
    {
        System.out.println('A');
    }
    {
        System.out.println('B');
    }
    public M()
    {
        System.out.println('C');
    }
}
class N extends M
{
    static
    {
        System.out.println('D');
    }
    {
        System.out.println('E');
    }
    public N()
    {
        System.out.println('F');
    }
}
public class MainClass
{
    public static void main(String[] args)
    {
        N n = new N();
    }
}
```

Encapsulations I

✓ Encapsulation is one of the four fundamental OOP concepts. The other three are inheritance, polymorphism, and abstraction.

✓ Encapsulation in Java is a process of wrapping code and data together into a single unit, for example, a capsule which is mixed of several medicines.

encapsulation in java

✓ *We can create a fully encapsulated class in Java by making all the data members of the class PRIVATE. Now we can use setter and getter methods to set and get the data in it*.

Advantage of Encapsulation in Java

✓ By providing only a setter or getter method, you can make the class **read-only or write-only**. In other words, you can skip the getter or setter methods.

Encapsulations II

✓ It provides you the control over the data. Suppose you want to set the value of id which should be greater than 100 only, you can write the logic inside the setter method. You can write the logic not to store the negative numbers in the setter methods.

✓ It is a way to achieve data hiding in Java because other class will not be able to access the data through the private data members.

✓ In Java, encapsulation helps us to keep related fields and methods together, which makes our code cleaner and easy to read.

✓ It helps to control the values of our data fields.

✓ The encapsulate class is easy to test. So, it is better for unit testing.

Encapsulations III

```
1  class Area
2  {
3          // fields to calculate area
4          int length;
5          int breadth;
6          // constructor to initialize values
7          Area(int length, int breadth)
8          {
9              this.length = length;
10             this.breadth = breadth;
11
12         }
13         // method to calculate area
14         public void getArea()
15         {
16             int area = length * breadth;
17             System.out.println("Area: " + area);
18
19         }
20 }
21 class Main
22 {
23         public static void main(String[] args)
24         {
25             // create object of Area
26             // pass value of length and breadth
27             Area rectangle = new Area(5, 6);
28             rectangle.getArea();
29
30         }
31 }
```

Encapsulations IV

✓ The getter and setter methods provide read-only or write-only access to our class fields.

For example,

```
1 getName()  // provides read-only access
2 setName() // provides write-only access
```

Encapsulations V

Data hiding using the private specifier:

```
1  class Person
2  {
3      // private field
4      private int age;
5      // getter method
6      public int getAge()
7      {
8          return age;
9      }
10     // setter method
11     public void setAge(int age)
12     {
13         this.age = age;
14     }
15 }
16 class Main
17 {
18     public static void main(String[] args)
19     {
20         // create an object of Person
21         Person p1 = new Person();
22         // change age using setter
23         p1.setAge(24);
24         // access age using getter
25         System.out.println("My age is " + p1.
       getAge());
26     }
27 }
```

Encapsulations VI

Read-Only class

```
1  //A Java class which has only getter methods.
2  public class Student
3  {
4      //private data member
5      private String college=”AKG”;
6      //getter method for college
7      public String getCollege()
8      {
9          return college;
10     }
11 }
```

Write-Only class

```
1  //A Java class which has only setter methods.
2  public class Student
3  {
4      //private data member
5      private String college;
6      //getter method for college
7      public void setCollege(String college)
8      {
9          this.college=college;
10     }
11 }
```

Polymorphism I

✓ Polymorphism in Java is a concept by which we can perform a single action in different ways.

✓ Polymorphism is derived from 2 Greek words: poly and morphs. The word ”poly” means many and ”morphs” means forms. So polymorphism means many forms.

✓ There are two types of polymorphism in Java: compile-time polymorphism and runtime polymorphism. We can perform polymorphism in java by method overloading and method overriding.

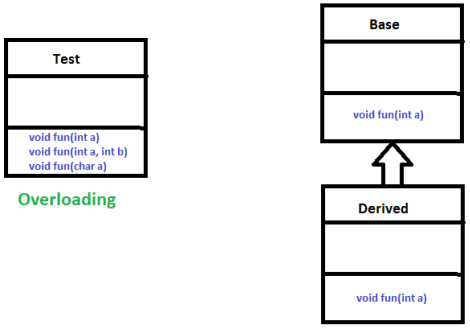
✓ If you overload a static method in Java, it is the example of compile time polymorphism. Here, we will focus on runtime polymorphism in java.

Polymorphism II

Compile-time polymorphism:

✓ It is also known as static polymorphism. This type of polymorphism is achieved by function overloading or operator overloading. But Java doesn't support the Operator Overloading.

Polymorphism III



✓ Method Overloading: When there are multiple functions with same name but different parameters then these functions are said to be overloaded. Functions can be overloaded by change in number of arguments or/and change in type of arguments.

Polymorphism IV

Example 1: By using different types of arguments

```
1  // Java program for Method overloading
2  class MultiplyFun
3  {
4      // Method with 2 parameter
5      static int Multiply(int a, int b)
6      {
7          return a * b;
8      }
9      // Method with the same name but 2 double
       parameter
10     static double Multiply(double a, double b)
11     {
12         return a * b;
13     }
14 }
15 class Main
16 {
17     public static void main(String[] args)
18     {
19         System.out.println(MultiplyFun.
       Multiply(2, 4));
20
21         System.out.println(MultiplyFun.
       Multiply(5.5, 6.3));
22     }
23 }
```

Polymorphism V

Example 2: By using different numbers of arguments

```
1  // Java program for Method overloading
2  class MultiplyFun
3  {
4      // Method with 2 parameter
5      static int Multiply(int a, int b)
6      {
7          return a * b;
8      }
9      // Method with the same name but 3
       parameter
10     static int Multiply(int a, int b, int c)
11     {
12         return a * b * c;
13     }
14 }
15 class Main
16 {
17     public static void main(String[] args)
18     {
19         System.out.println(MultiplyFun.
       Multiply(2, 4));
20
21         System.out.println(MultiplyFun.
       Multiply(2, 7, 3));
22     }
23 }
```

Polymorphism VI

Runtime Polymorphism in Java

✓ Runtime polymorphism or Dynamic Method Dispatch is a process in which a call to an **overridden method** is resolved at runtime rather than compile-time.

✓ In this process, an overridden method is called through the reference variable of a superclass.

✓ The determination of the method to be called is based on the object being referred to by the reference variable.

✓ Understand the upcasting before Runtime Polymorphism.

If the reference variable of Parent class refers to the object of Child class, it is known as upcasting. For example:

Polymorphism VII

```
1  class Bike
2  {
3          void run()
4          {
5              System.out.println("running");
6          }
7  }
8  class Splendor extends Bike
9  {
10         void run()
11         {
12             System.out.println("running safely
               with 60km");
13         }
14
15          public static void main(String args[])
16         {
17             Bike b = new Splendor(); //upcasting
18             b.run();
19
20         }
21 }
```

Polymorphism VIII

Java Runtime Polymorphism Example:

Polymorphism IX

```
1  // Java program for Method overriding
2  class Parent
3  {
4      void Print()
5      {
6          System.out.println("parent class");
7      }
8  }
9  class subclass1 extends Parent
10 {
11     void Print()
12     {
13         System.out.println("subclass1");
14     }
15 }
16 class subclass2 extends Parent
17 {
18     void Print()
19     {
20         System.out.println("subclass2");
21     }
22 }
23 class TestPolymorphism3
24 {
25     public static void main(String[] args)
26     {
27         Parent a;
28         a = new subclass1();
29         a.Print();
30         a = new subclass2();
31         a.Print();
32     }
33 }
```

Polymorphism X

✓ Example:

```
1  class Shape
2  {
3      void draw()
4      {
5          System.out.println(”drawing...”);
6      }
7  }
8  class Rectangle extends Shape
9  {
10     void draw()
11     {
12         System.out.println(”drawing rectangle...”);
13     }
14 }
```

Polymorphism XI

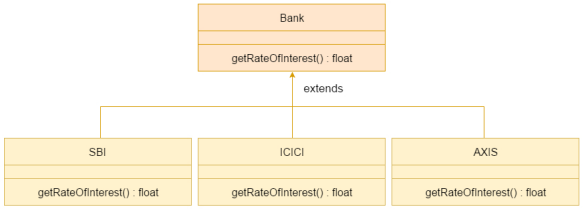
```
15 class Circle extends Shape
16 {
17     void draw()
18     {
19         System.out.println(”drawing circle...”);
20     }
21 }
22 class Triangle extends Shape
23 {
24     void draw()
25     {
26         System.out.println(”drawing triangle...”);
27     }
28 }
29 class TestPolymorphism2
30 {
31     public static void main(String args[])
```

Polymorphism XII

```
32      {
33          Shape s;
34          s=new Rectangle();
35          s.draw();
36          s=new Circle();
37          s.draw();
38          s=new Triangle();
39          s.draw();
40      }
41  }
```

✓ Consider a scenario where Bank is a class that provides a method to get the rate of interest. However, the rate of interest may differ according to banks. For example, SBI, ICICI, and AXIS banks are providing 8.4%, 7.3%, and 9.7% rate of interest.

Polymorphism XIII




```
1 class Bank
2 {
3     float getRateOfInterest()
4     {
5         return 0;
6     }
7 }
8 class SBI extends Bank
```

Polymorphism XIV

```
9  {
10     float getRateOfInterest()
11     {
12         return 8.4f;
13     }
14 }
15 class ICICI extends Bank
16 {
17     float getRateOfInterest()
18     {
19         return 7.3f;
20     }
21 }
22 class AXIS extends Bank
23 {
24     float getRateOfInterest()
25     {
```

Polymorphism XV

```
26              return 9.7f;
27          }
28  }
29  class TestPolymorphism
30  {
31          public static void main(String args[])
32          {
33              Bank b;
34              b=new SBI();
35              System.out.println("SBI Rate of Interest: "+b.getRateOfInterest());
36              b=new ICICI();
37              System.out.println("ICICI Rate of Interest: "+b.getRateOfInterest());
38              b=new AXIS();
39              System.out.println("AXIS Rate of Interest: "+b.getRateOfInterest());
40          }
41  }
```

Polymorphism XVI

Java Runtime Polymorphism with Multilevel Inheritance:

Polymorphism XVII

```
1  class Animal
2  {
3      void eat()
4      {
5          System.out.println("eating");
6      }
7  }
8  class Dog extends Animal
9  {
10     void eat()
11     {
12         System.out.println("eating fruits");
13     }
14 }
15 class BabyDog extends Dog
16 {
17     void eat()
18     {
19         System.out.println("drinking milk");
20     }
21     public static void main(String args[])
22     {
23         Animal a1,a2,a3;
24         a1=new Animal();
25         a2=new Dog();
26         a3=new BabyDog();
27         a1.eat();
28         a2.eat();
29         a3.eat();
30     }
31 }
```

Polymorphism XVIII

```
1  class Animal
2  {
3      void eat()
4      {
5          System.out.println("animal is eating...");
6      }
7  }
8  class Dog extends Animal
9  {
10     void eat()
11     {
12         System.out.println("dog is eating...");
13     }
14 }
15 public class BabyDog1 extends Dog
16 {
17     public static void main(String args[])
18     {
19         Animal a=new BabyDog1();
20         a.eat();
21     }
22
23 }
```

✓ Since, BabyDog1 is not overriding the eat() method, so eat() method of Dog class is invoked.