Session 07:

JAVA Inheritance and polymorphism

# Describe inheritance

# Explain the types of inheritance

# Explain super class and subclass

# Explain the use of super keyword

# Explain method overriding

# Describe Polymorphism

# Differentiate type of reference and type of objects

# Explain static and dynamic binding

# Explain virtual method invocation

# Explain the use of abstract keyword

# Session Overview

# Describe inheritance

**Inheritance in Java** is a mechanism in which one object acquires all the properties and behaviors of a parent object. It is an important part of OOPs

(Object Oriented programming system).

The idea behind inheritance in Java is that you can create new classes

that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of the parent class. Moreover, you can add new methods and fields in your current class also.

### **Terms used in Inheritance**

* **Class:** A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.
* **Sub Class/Child Class:** Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.
* **Super Class/Parent Class:** Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.
* **Reusability:** As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in the previous class.

### **The syntax of Java Inheritance**

**class** Subclass-name **extends** Superclass-name

{

   //methods and fields

}

#### The **extends keyword** indicates that you are making a new class that derives from an existing class. The meaning of "extends" is to increase the functionality.

### **Java Inheritance Example**



As displayed in the above figure, Programmer is the subclass and Employee is the superclass. The relationship between the two classes is **Programmer IS-A Employee**. It means that Programmer is a type of Employee.

# Explain the types of inheritance

On the basis of class, there can be three types of inheritance in java: single, multilevel and hierarchical.

In java programming, multiple and hybrid inheritance is supported through interface only. We will learn about interfaces later.



When one class inherits multiple classes, it is known as multiple inheritance. For Example:



## Single Inheritance Example

When a class inherits another class, it is known as a single inheritance. In the example given below, Dog class inherits the Animal class, so there is the single inheritance.

*File: TestInheritance.java*

**class** Animal{

**void** eat(){System.out.println("eating...");}

}

**class** Dog **extends** Animal{

**void** bark(){System.out.println("barking...");}

}

**class** TestInheritance{

**public** **static** **void** main(String args[]){

Dog d=**new** Dog();

d.bark();

d.eat();

}}

Output:

barking...

eating...

## Multilevel Inheritance Example

When there is a chain of inheritance, it is known as multilevel inheritance. As you can see in the example given below, BabyDog class inherits the Dog class which again inherits the Animal class, so there is a multilevel inheritance.

*File: TestInheritance2.java*

**class** Animal{

**void** eat(){System.out.println("eating...");}

}

**class** Dog **extends** Animal{

**void** bark(){System.out.println("barking...");}

}

**class** BabyDog **extends** Dog{

**void** weep(){System.out.println("weeping...");}

}

**class** TestInheritance2{

**public** **static** **void** main(String args[]){

BabyDog d=**new** BabyDog();

d.weep();

d.bark();

d.eat();

}}

## Hierarchical Inheritance Example

When two or more classes inherits a single class, it is known as hierarchical inheritance. In the example given below, Dog and Cat classes inherits the Animal class, so there is hierarchical inheritance.

*File: TestInheritance3.java*

**class** Animal{

**void** eat(){System.out.println("eating...");}

}

**class** Dog **extends** Animal{

**void** bark(){System.out.println("barking...");}

}

**class** Cat **extends** Animal{

**void** meow(){System.out.println("meowing...");}

}

**class** TestInheritance3{

**public** **static** **void** main(String args[]){

Cat c=**new** Cat();

c.meow();

c.eat();

//c.bark();//C.T.Error

}}

Output:

meowing...

eating...

## Why multiple inheritance is not supported in java?

* To reduce the complexity and simplify the language, multiple inheritance is not supported in java.
* Consider a scenario where A, B, and C are three classes. The C class inherits A and B classes. If A and B classes have the same method and you call it from child class object, there will be ambiguity to call the method of A or B class.
* Since compile-time errors are better than runtime errors, Java renders compile-time error if you inherit 2 classes. So whether you have same method or different, there will be compile time error.

**class** A{

**void** msg(){System.out.println("Hello");}

}

**class** B{

**void** msg(){System.out.println("Welcome");}

}

**class** C **extends** A,B{//suppose if it were

**public** **static** **void** main(String args[]){

   C obj=**new** C();

   obj.msg();//Now which msg() method would be invoked?

}

}

Compile Time Error

# Explain super class and subclass

* **Class:** A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.
* **Sub Class/Child Class:** Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.
* **Super Class/Parent Class:** Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.
* **Reusability:** As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in the previous class.

# Explain the use of super keyword

### **Example**

Using super to call the superclass of Dog (subclass):

class Animal { // Superclass (parent)

public void animalSound() {

System.out.println("The animal makes a sound");

}

}

class Dog extends Animal { // Subclass (child)

public void animalSound() {

super.animalSound(); // Call the superclass method

System.out.println("The dog says: bow wow");

}

}

public class Main {

public static void main(String args[]) {

Animal myDog = new Dog(); // Create a Dog object

myDog.animalSound(); // Call the method on the Dog object

}

}

* The super keyword refers to superclass (parent) objects.
* It is used to call superclass methods, and to access the superclass constructor.
* The most common use of the super keyword is to eliminate the confusion between superclasses and subclasses that have methods with the same name.
* To understand the super keyword, you should have a basic understanding of Inheritance and Polymorphism.

Example:

* If your method overrides one of its superclass's methods, you can invoke the overridden method through the use of the keyword super. You can also use super to refer to a hidden field (although hiding fields is discouraged). Consider this class, Superclass:

public class Superclass {

public void printMethod() {

System.out.println("Printed in Superclass.");

}

}

Here is a subclass, called Subclass, that overrides printMethod():

public class Subclass extends Superclass {

// overrides printMethod in Superclass

public void printMethod() {

super.printMethod();

System.out.println("Printed in Subclass");

}

public static void main(String[] args) {

Subclass s = new Subclass();

s.printMethod();

}

}

Within Subclass, the simple name printMethod() refers to the one declared in Subclass, which overrides the one in Superclass. So, to refer to printMethod() inherited from Superclass, Subclass must use a qualified name, using super as shown. Compiling and executing Subclass prints the following:

Printed in Superclass.

Printed in Subclass

# Explain method overriding

### If subclass (child class) has the same method as declared in the parent class, it is known as **method overriding in Java**.

### In other words, If a subclass provides the specific implementation of the method that has been declared by one of its parent class, it is known as method overriding.

#### **Rules for Java Method Overriding**

1. The method must have the same name as in the parent class
2. The method must have the same parameter as in the parent class.
3. There must be an IS-A relationship (inheritance).

### **Understanding the problem without method overriding**

/Java Program to demonstrate why we need method overriding

//Here, we are calling the method of parent class with child

//class object.

//Creating a parent class

**class** Vehicle{

**void** run(){System.out.println("Vehicle is running");}

}

//Creating a child class

**class** Bike **extends** Vehicle{

**public** **static** **void** main(String args[]){

  //creating an instance of child class

  Bike obj = **new** Bike();

  //calling the method with child class instance

  obj.run();

  }

}

Output:

Hello Java Program for Beginners

Vehicle is running

### **Example of method overriding**

### In this example, we have defined the run method in the subclass as defined in the parent class but it has some specific implementation. The name and parameter of the method are the same, and there is IS-A relationship between the classes, so there is method overriding.

//Java Program to illustrate the use of Java Method Overriding

//Creating a parent class.

**class** Vehicle{

  //defining a method

**void** run(){System.out.println("Vehicle is running");}

}

//Creating a child class

**class** Bike2 **extends** Vehicle{

  //defining the same method as in the parent class

**void** run(){System.out.println("Bike is running safely");}

**public** **static** **void** main(String args[]){

  Bike2 obj = **new** Bike2();//creating object

  obj.run();//calling method

  }

1. }

Output:

Bike is running safely

### **A real example of Java Method Overriding**

### Consider a scenario where Bank is a class that provides functionality to get the rate of interest. However, the rate of interest varies according to banks. For example, SBI, ICICI and AXIS banks could provide 8%, 7%, and 9% rate of interest.



//Java Program to demonstrate the real scenario of Java Method Overriding

//where three classes are overriding the method of a parent class.

//Creating a parent class.

**class** Bank{

**int** getRateOfInterest(){**return** 0;}

}

//Creating child classes.

**class** SBI **extends** Bank{

**int** getRateOfInterest(){**return** 8;}

}

**class** ICICI **extends** Bank{

**int** getRateOfInterest(){**return** 7;}

}

**class** AXIS **extends** Bank{

**int** getRateOfInterest(){**return** 9;}

}

//Test class to create objects and call the methods

**class** Test2{

**public** **static** **void** main(String args[]){

SBI s=**new** SBI();

ICICI i=**new** ICICI();

AXIS a=**new** AXIS();

System.out.println("SBI Rate of Interest: "+s.getRateOfInterest());

System.out.println("ICICI Rate of Interest: "+i.getRateOfInterest());

System.out.println("AXIS Rate of Interest: "+a.getRateOfInterest());

}

}

Output:

SBI Rate of Interest: 8

ICICI Rate of Interest: 7

AXIS Rate of Interest: 9

# Describe Polymorphism

### 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.

### Let's first understand the upcasting before Runtime Polymorphism.

### **Upcasting**

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



**class** A{}

**class** B **extends** A{}

A a=**new** B();//upcasting

* For upcasting, we can use the reference variable of class type or an interface type. For Example:

**interface** I{}

**class** A{}

**class** B **extends** A **implements** I{}

* Here, the relationship of B class would be:

B IS-A A

B IS-A I

B IS-A Object

### Since Object is the root class of all classes in Java, so we can write B IS-A Object.

### **Example of Java Runtime Polymorphism**

### In this example, we are creating two classes Bike and Splendor. Splendor class extends Bike class and overrides its run() method. We are calling the run method by the reference variable of Parent class. Since it refers to the subclass object and subclass method overrides the Parent class method, the subclass method is invoked at runtime.

### Since method invocation is determined by the JVM not compiler, it is known as runtime polymorphism.

**class** Bike{

**void** run(){System.out.println("running");}

}

**class** Splendor **extends** Bike{

**void** run(){System.out.println("running safely with 60km");}

**public** **static** **void** main(String args[]){

    Bike b = **new** Splendor();//upcasting

    b.run();

  }

}

Output:

running safely with 60km.

## Java Runtime Polymorphism Example: Bank

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.



#### **Note: This example is also given in method overriding but there was no upcasting.**

**class** Bank{

**float** getRateOfInterest(){**return** 0;}

}

**class** SBI **extends** Bank{

**float** getRateOfInterest(){**return** 8.4f;}

}

**class** ICICI **extends** Bank{

**float** getRateOfInterest(){**return** 7.3f;}

}

**class** AXIS **extends** Bank{

**float** getRateOfInterest(){**return** 9.7f;}

}

**class** TestPolymorphism{

**public** **static** **void** main(String args[]){

Bank b;

b=**new** SBI();

System.out.println("SBI Rate of Interest: "+b.getRateOfInterest());

b=**new** ICICI();

System.out.println("ICICI Rate of Interest: "+b.getRateOfInterest());

b=**new** AXIS();

System.out.println("AXIS Rate of Interest: "+b.getRateOfInterest());

}

}

Output:

SBI Rate of Interest: 8.4

ICICI Rate of Interest: 7.3

AXIS Rate of Interest: 9.7

## Java Runtime Polymorphism Example: Shape

**class** Shape{

**void** draw(){System.out.println("drawing...");}

}

**class** Rectangle **extends** Shape{

**void** draw(){System.out.println("drawing rectangle...");}

}

**class** Circle **extends** Shape{

**void** draw(){System.out.println("drawing circle...");}

}

**class** Triangle **extends** Shape{

**void** draw(){System.out.println("drawing triangle...");}

}

**class** TestPolymorphism2{

**public** **static** **void** main(String args[]){

Shape s;

s=**new** Rectangle();

s.draw();

s=**new** Circle();

s.draw();

s=**new** Triangle();

s.draw();

}

}

Output:

drawing rectangle...

drawing circle...

drawing triangle...

## Java Runtime Polymorphism Example: Animal

**class** Animal{

**void** eat(){System.out.println("eating...");}

}

**class** Dog **extends** Animal{

**void** eat(){System.out.println("eating bread...");}

}

**class** Cat **extends** Animal{

**void** eat(){System.out.println("eating rat...");}

}

**class** Lion **extends** Animal{

**void** eat(){System.out.println("eating meat...");}

}

**class** TestPolymorphism3{

**public** **static** **void** main(String[] args){

Animal a;

a=**new** Dog();

a.eat();

a=**new** Cat();

a.eat();

a=**new** Lion();

a.eat();

}}

Output:

eating bread...

eating rat...

eating meat...

## Java Runtime Polymorphism with Data Member

### A method is overridden, not the data members, so runtime polymorphism can't be achieved by data members.

### In the example given below, both the classes have a data member speedlimit. We are accessing the data member by the reference variable of Parent class which refers to the subclass object. Since we are accessing the data member which is not overridden, hence it will access the data member of the Parent class always.

**class** Bike{

**int** speedlimit=90;

}

**class** Honda3 **extends** Bike{

**int** speedlimit=150;

**public** **static** **void** main(String args[]){

  Bike obj=**new** Honda3();

  System.out.println(obj.speedlimit);//90

}

Output:

90

## Java Runtime Polymorphism with Multilevel Inheritance

Let's see the simple example of Runtime Polymorphism with multilevel inheritance.

**class** Animal{

**void** eat(){System.out.println("eating");}

}

**class** Dog **extends** Animal{

**void** eat(){System.out.println("eating fruits");}

}

**class** BabyDog **extends** Dog{

**void** eat(){System.out.println("drinking milk");}

**public** **static** **void** main(String args[]){

Animal a1,a2,a3;

a1=**new** Animal();

a2=**new** Dog();

a3=**new** BabyDog();

a1.eat();

a2.eat();

a3.eat();

}

}

Output:

eating

eating fruits

drinking Milk

### **Try for Output**

**class** Animal{

**void** eat(){System.out.println("animal is eating...");}

}

**class** Dog **extends** Animal{

**void** eat(){System.out.println("dog is eating...");}

}

**class** BabyDog1 **extends** Dog{

**public** **static** **void** main(String args[]){

Animal a=**new** BabyDog1();

a.eat();

}}

Output:

Dog is eating

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

# Differentiate type of reference and type of objects

### A class in a blue print/user defined datatype in java that describes the behavior/state that the object of its type support.

### Example

public class Student {

   String name "Krishna";

   int age = 20;

   void greet() {

      System.out.println("Hello how are you");

   }

}

### An object is an instance of a class created from it using the new keyword. Once you create an object of a class, using it you can access he members of the class. In the below given code an object of the class Student is created.

public class Example {

   public static void main(String args[]) {

      Student obj = new Student();

   }

}

### Classes, interfaces, arrays, enumerations and, annotations are the in Java are reference types in Java. Reference variables hold the objects/values of reference types in Java

### **Difference between object and reference**

When you create an object of a class as −

Student obj = new Student();

### The objects are created in the heap area and, the reference **obj** just points out to the object of the Student class in the heap, i.e. it just holds the memory address of the object (in the heap).

### And since the String is also an object, under name, a reference points out to the actual String value (“Krishna”).

Diagram

Description automatically generated

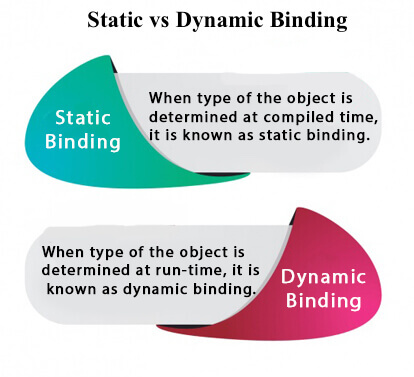
### In short, object is an instance of a class and reference (variable) points out to the object created in the heap area.

# Explain static and dynamic binding

### Connecting a method call to the method body is known as binding.

### There are two types of binding

1. Static Binding (also known as Early Binding).
2. Dynamic Binding (also known as Late Binding).



## Understanding Type

Let's understand the type of instance.

#### **1) variables have a type**

Each variable has a type, it may be primitive and non-primitive.

How to find Nth Highest Salary in SQL

**Next**

**Stay**

1. **int** data=30;

Here data variable is a type of int.

#### **2) References have a type**

**class** Dog{

**public** **static** **void** main(String args[]){

  Dog d1;//Here d1 is a type of Dog

 }

}

#### **3) Objects have a type**

|  |
| --- |
| An object is an instance of particular java class,but it is also an instance of its superclass. |

**class** Animal{}

**class** Dog **extends** Animal{

**public** **static** **void** main(String args[]){

  Dog d1=**new** Dog();

 }

}

|  |
| --- |
| Here d1 is an instance of Dog class, but it is also an instance of Animal. |

## static binding

When type of the object is determined at compiled time(by the compiler), it is known as static binding.

If there is any private, final or static method in a class, there is static binding.

### **Example of static binding**

class Dog{

private void eat(){System.out.println("dog is eating...");}

public static void main(String args[]){

Dog d1=new Dog();

    d1.eat();

 }

}

## Dynamic binding

When type of the object is determined at run-time, it is known as dynamic binding.

### **Example of dynamic binding**

**class** Animal{

**void** eat(){System.out.println("animal is eating...");}

}

**class** Dog **extends** Animal{

**void** eat(){System.out.println("dog is eating...");}

**public** **static** **void** main(String args[]){

  Animal a=**new** Dog();

  a.eat();

 }

}

Output:dog is eating...

|  |
| --- |
| In the above example object type cannot be determined by the compiler, because the instance of Dog is also an instance of Animal.So compiler doesn't know its type, only its base type. |

# Explain virtual method invocation

### The virtual keyword is not used in Java to define the virtual function; instead, the virtual functions and methods are achieved using the following techniques:

#### We can override the virtual function with the inheriting class function using the same function name. Generally, the virtual function is defined in the parent class and override it in the inherited class.

#### The virtual function is supposed to be defined in the derived class. We can call it by referring to the derived class's object using the reference or pointer of the base class.

#### A virtual function should have the same name and parameters in the base and derived class.

#### For the virtual function, an IS-A relationship is necessary, which is used to define the class hierarchy in inheritance.

#### The Virtual function cannot be private, as the private functions cannot be overridden.

#### A virtual function or method also cannot be final, as the final methods also cannot be overridden.

#### Static functions are also cannot be overridden; so, a virtual function should not be static.

#### By default, Every non-static method in Java is a virtual function.

#### The virtual functions can be used to achieve oops concepts like runtime polymorphism.

Let's understand it with some examples:

**Parent.Java:**

**class** Parent {

**void** v1() //Declaring function

{

System.out.println("Inside the Parent Class");

}

}

**Child.java:**

**public** **class** Child **extends** Parent{

**void** v1() // Overriding function from the Parent class

            {

             System.out.println("Inside the Child Class");

            }

**public** **static** **void** main(String args[]){

            Parent ob1 = **new** Child(); //Refering the child class object using the parent class

            ob1.v1();

            }

            }

**Output:**

Inside the Child Class

## Java Interfaces as Virtual Function

### An interface in Java is a blueprint of a class; it holds static constants and abstract methods. All Java Interfaces are considered virtual functions, as they depend on the implementing classes to provide the method implementation.

### Consider the below example to understand the behavior of the interface:

**interface** Car{

**void** print();

}

**class** BMW **implements** Car{

**public** **void** print(){System.out.println("BMW X7");}

**public** **static** **void** main(String args[]){

BMW obj = **new** BMW();

obj.print();

 }

}

**Output:**

BMW X7

### From the above example, we can see the interface's method is executed using the implementing class BMW.

### Hence, we can also achieve polymorphism using the Interfaces.

## Pure Virtual Function

### A virtual function for which we are not required implementation is considered as pure virtual function. For example, Abstract method in Java is a pure virtual function. Consider the below example:

**abstract** **class** Animal {

**final** **void** bark(){

        System.out.println("Dog");

        }

**abstract** **void** jump(); // Abstract Method (Pure Virtual Function)

}

**class** MyPet **extends** Animal{

**void** jump(){

        System.out.println("MyPet is so sweet");

        }

}

**public** **class** Demo {

**public** **static** **void** main(String args[]){

        Animal ob1 = **new** MyPet();

        ob1.jump();

        }

        }

**Output:**

MyPet is so sweet

# Explain the use of abstract keyword

### **Example**

An abstract method belongs to an abstract class, and it does not have a body. The body is provided by the subclass:

// Code from filename: Main.java

// abstract class  
abstract class Main {

public String fname = "John";

public int age = 24;

public **abstract** void study(); // abstract method

}

// Subclass (inherit from Main)

class Student extends Main {

public int graduationYear = 2018;

public void study() { // the body of the abstract method is provided here

System.out.println("Studying all day long");

}

}

// End code from filename: Main.java

// Code from filename: Second.java

class Second {

public static void main(String[] args) {

// create an object of the Student class (which inherits attributes and methods from Main)

Student myObj = new Student();

System.out.println("Name: " + myObj.fname);

System.out.println("Age: " + myObj.age);

System.out.println("Graduation Year: " + myObj.graduationYear);

myObj.study(); // call abstract method

}

}

### The abstract keyword is a non-access modifier, used for classes and methods.

#### **Class:** An abstract class is a restricted class that cannot be used to create objects (to access it, it must be inherited from another class).

#### **Method:** An abstract method can only be used in an abstract class, and it does not have a body. The body is provided by the subclass (inherited from).

Ex:

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

// Java program to illustrate the

// concept of inheritance

// base class

**class** Bicycle {

    // the Bicycle class has two fields

**public** **int** gear;

**public** **int** speed;

    // the Bicycle class has one constructor

**public** Bicycle(**int** gear, **int** speed)

    {

**this**.gear = gear;

**this**.speed = speed;

    }

    // the Bicycle class has three methods

**public** **void** applyBrake(**int** decrement)

    {

        speed -= decrement;

    }

**public** **void** speedUp(**int** increment)

    {

        speed += increment;

    }

    // toString() method to print info of Bicycle

**public** String toString()

    {

**return** ("No of gears are " + gear + "\n"

                + "speed of bicycle is " + speed);

    }

}

// derived class

**class** MountainBike **extends** Bicycle {

    // the MountainBike subclass adds one more field

**public** **int** seatHeight;

    // the MountainBike subclass has one constructor

**public** MountainBike(**int** gear, **int** speed,

**int** startHeight)

    {

        // invoking base-class(Bicycle) constructor

**super**(gear, speed);

        seatHeight = startHeight;

    }

    // the MountainBike subclass adds one more method

**public** **void** setHeight(**int** newValue)

    {

        seatHeight = newValue;

    }

    // overriding toString() method

    // of Bicycle to print more info

    @Override **public** String toString()

    {

**return** (**super**.toString() + "\nseat height is "

                + seatHeight);

    }

}

// driver class

**public** **class** Test {

**public** **static** **void** main(String args[])

    {

        MountainBike mb = **new** MountainBike(3, 100, 25);

        System.out.println(mb.toString());

    }

}

**Output**

No of gears are 3

speed of bicycle is 100

seat height is 25

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

// Java program to illustrate the

// concept of single inheritance

**import** java.io.\*;

**import** java.lang.\*;

**import** java.util.\*;

**class** one {

**public** **void** print\_geek()

    {

        System.out.println("Geeks");

    }

}

**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\_geek();

        g.print\_for();

        g.print\_geek();

    }

}

* 1. **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 the below image, 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

/ Java program to illustrate the

// concept of Multilevel inheritance

**import** java.io.\*;

**import** java.lang.\*;

**import** java.util.\*;

**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();

    }

}

* 1. **Hierarchical Inheritance:**In Hierarchical Inheritance, one class serves as a superclass (base class) for more than one subclass. In the below image, class A serves as a base class for the derived class B, C and D.

// Java program to illustrate the

// concept of Hierarchical  inheritance

**class** A {

**public** **void** print\_A() { System.out.println("Class A"); }

}

**class** B **extends** A {

**public** **void** print\_B() { System.out.println("Class B"); }

}

**class** C **extends** A {

**public** **void** print\_C() { System.out.println("Class C"); }

}

**class** D **extends** A {

**public** **void** print\_D() { System.out.println("Class D"); }

}

// Driver Class

**public** **class** Test {

**public** **static** **void** main(String[] args)

    {

        B obj\_B = **new** B();

        obj\_B.print\_A();

        obj\_B.print\_B();

        C obj\_C = **new** C();

        obj\_C.print\_A();

        obj\_C.print\_C();

        D obj\_D = **new** D();

        obj\_D.print\_A();

        obj\_D.print\_D();

    }

}

Class A

Class B

Class A

Class C

Class A

Class D