

---

## Unit-III Inheritance, Interface and Package

### 3.1 Inheritance: -

#### Definition-

- Inheritance is a mechanism in java by which derived class can borrow the properties of base class and at the same time the derived class may have some additional properties.
- The inheritance can be achieved by incorporating the definition of one class into another using the keyword **extends**.

#### 3.1.1. Concept of inheritance: -

- The inheritance is a mechanism in which the child class is derived from a parent class.
- This derivation is using the keyword **extends**.

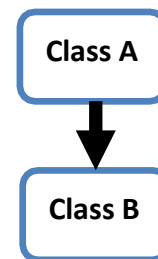
For example:

Class A <- base class

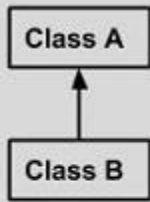
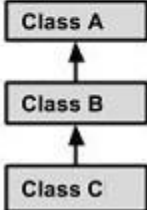
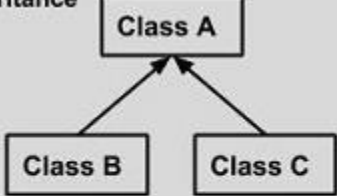
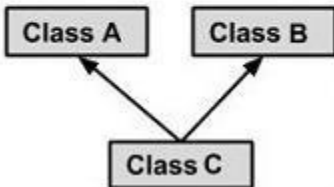
```
{  
.....  
}
```

Class B extends A

```
{  
---- //uses of properties of A  
}
```



### 3.2. Types of Inheritance

<b>Single Inheritance</b>  <pre> classDiagram     ClassB -- &gt; ClassA         </pre>	<pre> public class A {     ..... } public class B extends A {     ..... }         </pre>
<b>Multi Level Inheritance</b>  <pre> classDiagram     ClassC -- &gt; ClassB     ClassB -- &gt; ClassA         </pre>	<pre> public class A { .....} public class B extends A { .....} public class C extends B { .....}         </pre>
<b>Hierarchical Inheritance</b>  <pre> classDiagram     ClassB -- &gt; ClassA     ClassC -- &gt; ClassA         </pre>	<pre> public class A { .....} public class B extends A { .....} public class C extends A { .....}         </pre>
<b>Multiple Inheritance</b>  <pre> classDiagram     ClassC -- &gt; ClassA     ClassC -- &gt; ClassB         </pre>	<pre> public class A { .....} public class B { .....} public class C extends A,B {     ..... } // Java does not support multiple Inheritance         </pre>

#### Program.1. Program for implanting Single Inheritance

```

class Teacher
{
void Display()
{
System.out.println("This is teacher ");
}
}
class student extends Teacher{
void show()
{
System.out.println("This is Student");
}
}
        
```

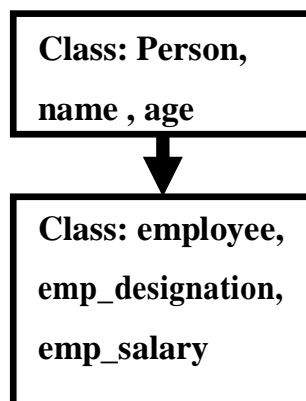


```
}  
class TestInheritance  
{  
public static void main(String args[])  
{  
Student s=new student();  
s.show();  
d.display();  
}  
}
```

### Output-

This is student  
This is teacher

**Program 2. Write a program to implement following inheritance**



```
class person  
{  
String name;  
int age;  
void accept(String n,int a)  
{  
    name=n;  
    age=a;
```



```
    }
    void display()
    {
        System.out.println("name--->" + name);
        System.out.println("age--->" + age);
    }
}

class employee extends person
{
    String emp_designation;
    float emp_salary;
    void accept_emp(String d, float s)
    {
        emp_designation = d;
        emp_salary = s;
    }

    void emp_dis()
    {
        System.out.println("emp_designation-->" + emp_designation);
        System.out.println("emp_salary-->" + emp_salary);
    }
}

class single_demo
{
    public static void main(String args[])
    {
        employee e = new employee();
        e.accept("ramesh", 35);
        e.display();
        e.accept_emp("lecturer", 35000.78f);
        e.emp_dis();
    }
}
```



```
    }  
}
```

**Program 3. Write a java program to implement multilevel inheritance with 4 levels of hierarchy.**

```
class emp  
{  
    int empid;  
    String ename;  
    emp(int id, String nm)  
    {  
        empid=id; ename=nm;  
    }  
}  
class work_profile extends emp  
{  
    String dept;  
    String job;  
    work_profile(int id, String nm, String dpt, String j1)  
    {  
        super(id,nm);  
        dept=dpt; job=j1;  
    }  
}  
class salary_details extends work_profile  
{  
    int basic_salary;  
    salary_details(int id, String nm, String dpt, String j1,int bs)  
    {  
        super(id,nm,dpt,j1);
```



---

```
        basic_salary=bs;
    }
    double calc()
    {
        double gs;
        gs=basic_salary+(basic_salary*0.4)+(basic_salary*0.1);
        return(gs);
    }
}

class salary_calc extends salary_details
{
    salary_calc(int id, String nm, String dpt, String j1,int bs)
    {
        super(id,nm,dpt,j1,bs);
    }
    public static void main(String args[])
    {
        salary_calc e1=new salary_calc(101,"abc","Sales","clerk",5000);
        double gross_salary=e1.calc();
        System.out.println("Empid :"+e1.empid);
        System.out.println("Emp name :"+e1.ename);
        System.out.println("Department :"+e1.dept);
        System.out.println("Job :"+e1.job);
        System.out.println("BAsic Salary :"+e1.basic_salary);
        System.out.println("Gross salary :"+gross_salary);
    }
}
```



---

### 3.3. Method Overloading

Def- Method Overloading means to define different methods with the same name but different parameters lists and different definitions.

It is used when objects are required to perform similar task but using different input parameters that may vary either in number or type of arguments.

Overloaded methods may have different return types.

It is a way of achieving polymorphism in java.

```
int add( int a, int b)           // prototype 1
int add( int a , int b , int c)  // prototype 2
double add( double a, double b)  // prototype 3
```

#### Example :

```
class Sample
{
    int addition(int i, int j)
    {
        return i + j ;
    }
    String addition(String s1, String s2)
    {
        return s1 + s2;
    }
    double addition(double d1, double d2)
    {
        return d1 + d2;
    }
}
class AddOperation
{
    public static void main(String args[])
```



```
    {
    Sample sObj = new Sample();
    System.out.println(sObj.addition(1,2));
    System.out.println(sObj.addition("Hello ", "World"));
    System.out.println(sObj.addition(1.5,2.2));
    }
}
```

### 3.4. Constructor Overloading

In Java, a constructor is just like a method but without return type. It can also be overloaded like Java methods.

Constructor overloading in Java is a technique of having more than one constructor with different parameter lists. They are arranged in a way that each constructor performs a different task. They are differentiated by the compiler by the number of parameters in the list and their types.

//Java program to overload constructors in java

```
class Student
{
    int id;
    String name;
    int age;
    //creating two arg constructor
    Student(int i,String n)
    {
        id = i;
        name = n;
    }
    //creating three arg constructor

    Student(int i,String n,int a)
    {
        id = i;
        name = n;
```





---

```
        age=a;
    }
    void display()
    {
        System.out.println(id+ " "+name+" "+age);
    }

    public static void main(String args[])
    {
        Student s1 = new Student5(111,"Karan");
        Student s2 = new Student5(222,"Aryan",25);
        s1.display();
        s2.display();
    }
}
```

**Program 4. Define a class person with data member as Aadharno, name, Panno implement concept of constructor overloading. Accept data for 5 object and print it.**

```
import java.io.*;

class Person
{
    intAadharno;
    String name;
    String Panno;
    Person(intAadharno, String name, String Panno)
    {
        this.Aadharno = Aadharno;
        this.name = name;
        this.Panno = Panno;
    }
    Person(intAadharno, String name)
```



---

```
        {
            this.Aadharno = Aadharno;
            this.name = name;
            Panno = "Not Applicable";
        }
    void display()
    {
        System.out.println("Aadharno is :"+Aadharno);
        System.out.println("Name is: "+name);
        System.out.println("Panno is :"+Panno);
    }
    public static void main(String ar[])
    {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        Person p, p1, p2, p3, p4;
        int a;
        String n, pno;
        try
        {
            System.out.println("Enter Aadhar no");
            a = Integer.parseInt(br.readLine());
            System.out.println("Enter name");
            n = br.readLine();
            System.out.println("Enter panno");
            pno = br.readLine();
            p = new Person(a,n,pno);
            System.out.println("Enter Aadhar no");
            a = Integer.parseInt(br.readLine());
            System.out.println("Enter name");
            n = br.readLine();
            System.out.println("Enter panno");
            pno = br.readLine();
```



---

```
p1 = new Person(a,n,pno);
System.out.println("Enter Aadhar no");
a = Integer.parseInt(br.readLine());
System.out.println("Enter name");
n = br.readLine();
p2 = new Person(a,n);
System.out.println("Enter Aadhar no");
a = Integer.parseInt(br.readLine());
System.out.println("Enter name");
n = br.readLine();
p3 = new Person(a,n);
System.out.println("Enter Aadhar no");
a = Integer.parseInt(br.readLine());
System.out.println("Enter name");
n = br.readLine();
System.out.println("Enter panno");
pno = br.readLine();
p4 = new Person(a,n,pno);
p.display();
p1.display();
p2.display();
p3.display();
p4.display();
    }
catch(Exception e)
    {
        System.out.println("Exception caught"+e);
    }
}
```



---

### 3.5. Overriding(Method Overriding)

If subclass (child class) has the same method as declared in the parent class, it is known as method overriding in java. If subclass provides the specific implementation of the method that has been provided by one of its parent class, it is known as method overriding. Method overriding is used for runtime polymorphism.

Example:

```
class Vehicle
{
    void run()
    {
        System.out.println("Vehicle is running");
    }
}
class Bike extends Vehicle
{
    void run()
    {
        System.out.println("Bike is running safely");
    }
}
public static void main(String args[])
{
    Bike2 obj = new Bike2();
    obj.run();
}
```

### 3.6. Dynamic Method Dispatch (Runtime Polymorphism)

Dynamic method dispatch is a mechanism by which a call to an overridden method is resolved at runtime. This is how java implements runtime polymorphism. When an overridden method is called by a reference, java determines which version of that method to



---

execute based on the type of object it refer to. In simple words the type of object which it referred determines which version of overridden method will be called.

### Example

```
class Game
{
    public void type()
    { System.out.println("Indoor & outdoor"); }
}
```

Class Cricket extends Game

```
{
    public void type()
    {
        System.out.println("outdoor game");
    }
}
```

```
public static void main(String[] args)
{
    Game gm = new Game();
    Cricket ck = new Cricket();
    gm.type();
    ck.type();
    gm=ck; //gm refers to Cricket object
    gm.type(); //calls Cricket's version of type
}
}
```



---

**Program.5. Write a single program to implement inheritance and polymorphism in java.**

```
class Employee
{
    String name;
    String address;
    int number;
    Employee(String name, String address, int number)
    {
        System.out.println("Constructing an Employee");
        this.name = name;
        this.address = address;
        this.number = number;
    }
    public void mailCheck()
    {
        System.out.println("Mailing a check to " + this.name + " " + this.address);
    }
    public String toString()
    {
        return name + " " + address + " " + number;
    }
    public String getName()
    {
        return name;
    }
    public String getAddress()
    {
        return address;
    }
    public void setAddress(String newAddress)
```



---

```
        {
            address = newAddress;
        }
        public int getNumber()
        {
            return number;
        }
    }
}

class Salary extends Employee
{
    private double salary;
    Salary(String name, String address, int number, double salary)
    {
        super(name, address, number);
        setSalary(salary);
    }
    public void mailCheck()
    {
        System.out.println("Within mailCheck of Salary class ");
        System.out.println("Mailing check to " + getName() + " with salary " + salary);
    }
    public double getSalary()
    {
        return salary;
    }
    public void setSalary(double newSalary)
    {
        if(newSalary >= 0.0)
        {
            salary = newSalary;
        }
    }
}
```



---

```
public double computePay()
    {
        System.out.println("Computing salary pay for " + getName());
        return salary/52;
    }
}

public class Demo
    {
        public static void main(String [] args)
        {
            Salary s = new Salary("RAM", "Dadar", 3, 3600.00);
            Employee e = new Salary("John ", "Thane", 2, 2400.00);
            System.out.println("Call mailCheck using Salary reference --");
            s.mailCheck();
            System.out.println("\n Call mailCheck using Employee reference--");
            e.mailCheck();
        }
    }
}
```

### 3.7. Final Variable and Final Methods

All variable and methods can be overridden by default in subclass. In order to prevent this, the final modifier is used. Final modifier can be used with variable, method or class.

**final variable:** the value of a final variable cannot be changed.

final variable behaves like class variables and they do not take any space on individual objects of the class.

Eg of declaring final variable:

```
final int size = 100;
```





---

**final method:** making a method final ensures that the functionality defined in this method will never be altered in any way, ie a final method cannot be overridden.

Eg of declaring a final method:

```
final void findAverage()  
{  
    //implementation  
}
```

### 3.8. Use of Super Keyword

when you will want to create a superclass that keeps the details of its implementation to itself (that is, that keeps its data members private). In this case, there would be no way for a subclass to directly access or initialize these variables on its own. Whenever a subclass needs to refer to its immediate super class, it can do so by use of the keyword super.

As constructor can not be inherited, but derived class can called base class constructor using super () super has two general forms.

The first calls the super class constructor.

super() method)

The second is used to access a member of the super class that has been hidden by a member of a subclass.

#### Using super () to Call Super class Constructors

A subclass can call a constructor method defined by its super class by use of the following form of super:

```
super(parameter-list);
```

Here,

parameter-list specifies any parameters needed by the constructor in the super class. super( ) must always be the first statement executed inside a subclass" constructor.

#### A Second Use for super

The second form of super acts somewhat like this, except that it always refers to the super class of the subclass in which it is used.



---

This usage has the following general form: `super.member`

Here, `member` can be either a method or an instance variable. This second form of `super` is most applicable to situations in which member names of a subclass hide members by the same name in the super class.

**Example:** // Using `super` to overcome name hiding.

```
class A
{
    int i;
} // Create a subclass by extending class A.
class B extends A
{
    int i; // this i hides the i in A
    B(int a, int b)
    {
        super.i = a; // i in A
        i = b; // i in B
    }
    void show()
    {
        System.out.println("i in superclass: " + super.i);
        System.out.println("i in subclass: " + i);
    }
}
class UseSuper
{
    public static void main(String args[])
    {
        B subOb = new B(1, 2);
        subOb.show();
    }
}
```



---

**Q. What is importance of super and this keyword in inheritance? Illustrate with suitable example**

Using inheritance, you can create a general class that defines traits common to a set of related items. This class can then be inherited by other, more specific classes, each adding those things that are unique to it. In the terminology of Java, 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. Whenever a subclass needs to refer to its immediate superclass, it can do so by use of the keyword super. Super has two general forms. The first calls the super class constructor.

The second is used to access a member of the superclass that has been hidden by a member of a subclass.

super() is used to call base class constructor in derived class. Super is used to call overridden method of base class or overridden data or evoked the overridden data in derived class.

E.g.: use of super()

```
class BoxWeight extends Box
```

```
{
```

```
    BowWeight(int a ,int b,int c ,int d)
```

```
    {
```

```
        super(a,b,c) // will call base class constructor Box(int a, int b,  
int c)
```

```
        weight=d // will assign value to derived class member weight.
```

```
    }
```

E.g.: use of super.

```
Class Box
```

```
{
```

```
    Box()
```

```
    {
```

```
    }
```

```
    void show()
```



```
    {
        //definition of show
    }
} //end of Box class
```

Class BoxWeight extends Box

```
{
    BoxWeight()
    {
    }
    void show() // method is overridden in derived
    {
        Super.show() // will call base class method
    }
}
```

### 3.9. Abstract Methods and Classes

Abstract methods, similar to methods within an interface, are declared without any implementation. They are declared with the purpose of having the child class provide implementation. They must be declared within an abstract class.

A class declared abstract may or may not include abstract methods. They are created with the purpose of being a super class.

Syntax

```
modifier abstract class className {
    //declare fields
    //declare methods
    abstract dataType methodName();
}

modifier class childClass extends className {
    dataType methodName(){}
}
```



---

Abstract classes and methods are declared with the 'abstract' keyword. Abstract classes can only be extended, and cannot be directly instantiated.

Abstract classes provide a little more than interfaces. Interfaces do not include fields and super class methods that get inherited, whereas abstract classes do. This means that an abstract class is more closely related to a class which extends it, than an interface is to a class that implements it.

### Example

```
public abstract class Animal
{
    String name;
    abstract String sound(); //all classes that implement Animal must have a sound method
}

public class Cat extends Animal
{
    public Cat()
    {
        this.name = "Garfield";
    }
    public String sound()
    {
        //implemented sound method from the abstract class & method
        return "Meow!";
    }
}
```

### 3.10. Static Members

Static members are data members (variables) or methods that belong to a static or a non static class itself, rather than to objects of the class. Static members always remain the same, regardless of where and how they are used. Because static members are associated with the class, it is not necessary to create an instance of that class to invoke them.

```
static data_type variable_name;
```

### Example

```
class VariableDemo
{
```



```
static int count=0;
public void increment()
{
    count++;
}
public static void main(String args[])
{
    VariableDemo obj1=new VariableDemo();
    VariableDemo obj2=new VariableDemo();
    obj1.increment();
    obj2.increment();
    System.out.println("Obj1: count is="+obj1.count);
    System.out.println("Obj2: count is="+obj2.count);
}
}
```

### Output:

```
Obj1: count is=2
Obj2: count is=2
```

## Interface:-

### ➤ Defining Interfaces:-

Interface is also known as kind of a class. Interface also contains methods and variables but with major difference, the interface consist of only abstract method (i.e. methods are not defined, these are declared only) and final fields (shared constants).

This means that interface do not specify any code to implement those methods and data fields contains only constants. Therefore, it is the responsibility of the class that implements an interface to define the code for implementation of these methods. An interface is defined much like class.

### Syntax:

```
access interface InterfaceName
{
    return_type method_name1(parameter list);
    ....
    return_type method_nameN(parameter list);
}
```



```
type final-variable 1 = value1;
....
type final-variable N = value n;
}
```

Where,

access is either public or not used.

‘interface’ is the java keyword and “InterfaceName” is nay valid java variables.

Variables- of interface are explicitly declared final and static. This means that the implementing class cannot change them. They must be initialized with constant value.

Methods- The methods declared in interfaces are abstract, there can be no default implementation of any method specified within an interface.

Following is the example of interface definition

```
interface student
{
    static final int rollno=1;
    static final String name="Genelia"
    void show();
}
```

- **Features:**

1. Variable of an interface are explicitly declared final and static (as constant) meaning that the implementing the class cannot change them they must be initialize with a constant value all the variable are implicitly public of the interface, itself, is declared as a public
2. Method declaration contains only a list of methods without anybody statement and ends with a semicolon the method are, essentially, abstract methods there can be default implementation of any method specified within an interface each class that include an interface must implement all of the method

- **Need:**

1. To achieve multiple Inheritance.
2. We can implement more than one Interface in the one class.
3. Methods can be implemented by one or more class.



➤ **Difference between class and interface**

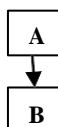
<b>Class</b>	<b>Interface</b>
It has instance variable.	It has final variable.
It has non abstract method.	It has by default abstract method.
We can create object of class.	We can,,t create object of interface.
Class has the access specifiers like public, private, and protected.	Interface has only public access specifier
Classes are always extended.	Interfaces are always implemented.
The memory is allocated for the classes.	We are not allocating the memory for the interfaces.
Multiple inheritance is not possible with classes	Interface was introduced for the concept of multiple inheritance
<pre>class Example { void method1() { Body } void method2() { body } }</pre>	<pre>interface Example { int x =5; void method1(); void method2(); }</pre>

➤ **Extending interfaces:-**

Like classes, Interfaces can also be extended. That is an interface can be sub-interfaced from other interfaces. The new sub-interfaced will inherit all the members of the sub-interfaces in the manner similar to subclasses.

- This is achieved using the keyword **extends** as shown below.

```
interface A extends B
{
    body of A.
}
```

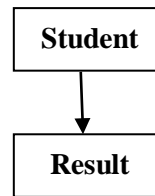


- We can put all the constant in one interface and the methods in the other. This will enable us to use the constants in the classes where the methods are not required.

**E.g**



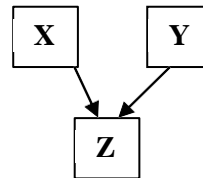
```
interface Student
{
    int RN=1;
    String name ="Genelia";
}
interface Result extends Student
{
    void display( );
}
```



- We can also combine several interfaces together into a single inheritance.

**E.g**

```
interface X
{
    int cost=100;
    String name="book";
}
interface Y
{
    void display();
}
interface Z extends X , Y
{
    -----
}
```



### ➤ **Implementing interfaces:-**

Interfaces are used as “super classes” whose properties are inherited by classes. It is therefore necessary to create a class that inherits the given interface. This is done as follows.

```
class classname implements interfacename
{
    Body of classname;
}
```

**E.g.**

```
class A implements B
{
    Body of class A;
}
```

Here, class A “implements ” the interface “B”.

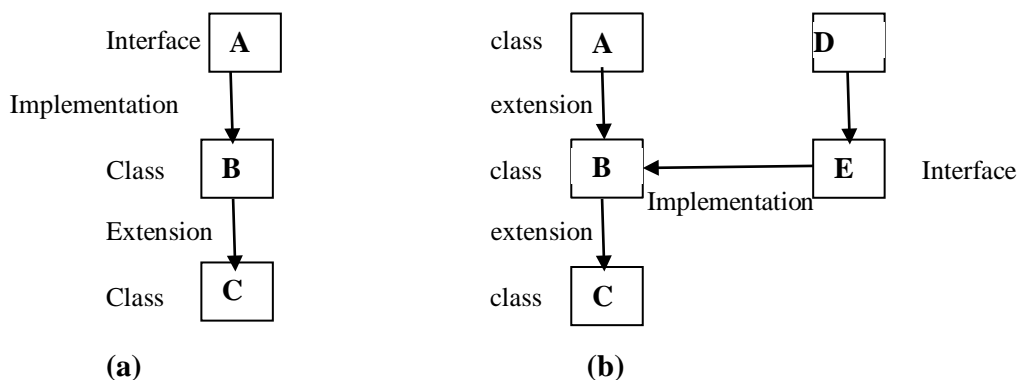
- A more general form of implementation may be as follows

```
class classname extends superclass implements interface1, interface2,.....
{
    body of classname;
}
```

**E.g.**

```
class A extends B implements X,Y, .....
{
    Body of class A;
}
```

- **Following are the various forms of interface implementation.**



### 1. Program for implementing interface:-(Use of interface )

```
interface Area // interface defined
{
    final static float pi=3.14F;
    float Cal(float x, float y);
}
class Rectangle implements Area
{
    public float Cal(float x, float y)
    {
        return(x * y);
    }
}
class Circle implements Area
{
```



```
        public float Cal(float x, float y)
        {
            return(pi *x * x);
        }
    }
```

```
class InterfaceDemo
{
    Public static void main(String args[ ])
    {
        Rectangle r = new Rectangle( );
        Circle c = new Circle();
        Area a ; // interface object
        a = r;
        System.out.println("Area of Rectangle=" +a.Cal(10,20));
        a = c;
        System.out.println("Area of Circle=" +a.Cal(10, 0));
    }
}
```

Output :-

```
C:\java\jdk1.7.0\bin> javac InterfaceDemo.java
C:\java\jdk1.7.0\bin> java InterfaceDemo
Area of Rectangle=200
Area of Circle=314
```

## 2. Program for Example of interface:- (Implementing Multilevel with hybrid inheritance)

```
interface sports
{
    int sport_wt=5;
    public void disp();
}
class Test
{
    int roll_no;
    String name;
    int m1,m2;
    Test (int r, String nm, int m11,int m12)
    {
        roll_no=r;
        name=nm;
        m1=m11;
        m2=m12;
    }
}
```



```
class Result extends Test implements sports
{
    Result (int r, String nm, int m11,int m12)
        {
            super (r,nm,m11,m12);
        }
    public void disp( )
        {
            System.out.println("Roll no : "+roll_no);
            System.out.println("Name : "+name);
            System.out.println("sub1 : "+m1);
            System.out.println("sub2 : "+m2);
            System.out.println("sport_wt : "+sport_wt);

            int t=m1+m2+sport_wt;

            System.out.println("total : "+t);
        }
}

class Demo
{
    public static void main(String args[])
    {
        Result r= new Result(101,"abc",75,75);
        r.disp();
    }
}
```

### ➤ Accessing interface:- References, Variables & Methods

Interfaces can be used to declare a set of constant that can be used in different classes. This is similar to creating header files in C++ to contain a large no. of constants. The constant values will be available to any class that implements the interface. The values can be used in declaration, or anywhere we can use a final value.

**E.g.**

```
interface A
{
    int m=10;
    int n=50;
}
class B implements A
{
    int x=m;
    void method_B(int s)
        {
            -----
        }
}
```



```
        if(s<n)
        }
    }
```

### ➤ Nesting of inheritances:-

Interfaces can be nested similar to a class. An interface can be nested within a *class* and *another interface*.

An interface can be declared inside a class body or interface body is called as nesting of interfaces. The nested interface cannot be accessed directly. To access nested interface, it must be referred by the outer interface or class.

- Rules for nesting interface:-

- 1) Nested interface must be public if it is declared inside the interface.
- 2) Nested interface can have access modifiers if it is declared inside the class.
- 3) Nested interfaces are declared as static implicitly.

- **interface inside interface-**

- **Syntax-**

```
interface interface_name
{
    ----
    interface nested_interface_name;
    {
        -----
    }
}
```

- **Nested interface within classes-**

- **Syntax-**

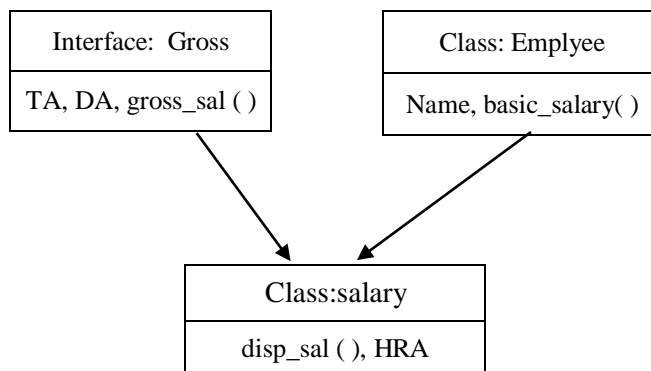
```
class class_name
{
    ----
    Interface nested_interface_name:
    {
        ----
    }
}
```

### ➤ Que. How to achieve multiple inheritance explain with suitable program?

- **Multiple inheritances:-**

It is a type of inheritance where a derived class may have more than one parent class. It is not possible in case of java as you cannot have two classes at the parent level. Instead, there can be one class and one interface at the parent level to achieve multiple interface.

Interface is similar to classes but can contain final variables and abstract methods. Interfaces can be implemented to a derived class.



### Program for above hierarchy (To achieve Multiple inheritance)

```
interface Gross
{
    double TA=800.0;
    double DA=3500;
    void gross_sal();
}
class Employee
{
    String name;
    double basic_sal;

    Employee(String n, double b)
    {
        name=n; basic_sal=b;
    }
    void display()
    {
        System.out.println("Name of Employee :"+name);
        System.out.println("Basic Salary of Employee :"+basic_sal);
    }
}

class Salary extends Employee implements Gross
{
    double HRA;
    Salary(String n, double b, double h)
    {
        super(n,b);
        HRA=h;
    }
    void disp_sal()
    {
        display();
        System.out.println("HRA of Employee :"+hra);
    }
}
```

```

        public void gross_sal()
        {
            double gross_sal=basic_sal + TA + DA + HRA;
            System.out.println("Gross salary of Employee :"+gross_sal);
        }
    }
class EmpDetails
{
    public static void main(String args[])
    {
        Salary s=new Salary("Sachin",8000,3000);
        s.disp_sal();
        s.gross_sal();
    }
}

```

### 3.2 Packages-

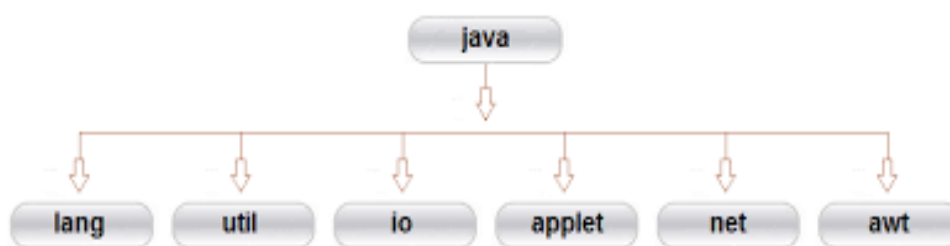
Java provides a mechanism for partitioning the class namespace into more manageable parts called package (i.e package are container for a classes). The package is both naming and visibility controlled mechanism.

We can define classes inside a package that are not accessible by code outside that package. We can also define class members that are only exposed to members of the same package.

#### ➤ Java API packages:-

The java API provides a no. of classes grouped into different packages according to their functionality.

The following fig1. shows the java API packages.



*fig1. Java API Packages*

- Packages and their use

Package Name	Description (Use of Package)
java.lang	Language support classes. These are classes that java compiler itself uses and therefore they are automatically imported. They include classes for primitive types, strings, math functions, threads and exceptions.



java.util	Language utility classes such as vectors, hash tables, random numbers, date etc.
java.io	Input/output support classes. They provide facilities for the input and output of data
java.awt	Set of classes for implementing graphical user interface. They include classes for windows, buttons, lists, menus and so on.
java.net	Classes for networking. They include classes for communicating with local computers as well as with internet servers.
java.applet	Classes for creating and implementing applets.

- **Naming Conventions:-**

Packages in java can be named using standard java naming rules.

**E.g.**

1) java.awt.Color;

**awt-** package name, **Color-**class name

2) double x= java.lang.Math.sqrt(a);

**lang-** package name, **math-** class name, **sqrt-** method name

➤ **Creating Packages:- (Defining Packages)**

Creation of packages includes following steps.

1) Declare a package at the beginning of the file using the following form.

**package** *package\_name*

**e.g.**

package pkg; - name of package

package is the java keyword with the name of package. This must be the first statement in java source file.

2) Define a class which is to be put in the package and declare it public like following way.

```
Package first_package;  
Public class first_class  
{  
----  
Body of class;  
}
```

In above example, “first\_package” is the package name. The class “first\_class” is now considered as a part of this package.





- 3) Create a sub-directory under the directory where the main source files are stored.
- 4) Store the listing of, as classname.java file is the sub-directory created.

e.g. from the above package we can write “first\_class.java”

- 5) Compile the source file. This creates .class file is the sub-directory. The .class file must be located in the package and this directory should be a sub-directory where classes that will import the package are located.

➤ **Que. How to create package?**

The syntax for creating package is:

**package *pkg*;**

Here, *pkg* is the name of the package

e.g : package mypack;

Packages are mirrored by directories. Java uses file system directories to store packages. The class files of any classes which are declared in a package must be stored in a directory which has same name as package name. The directory must match with the package name exactly.

A hierarchy can be created by separating package name and sub package name by a period(.) as pkg1.pkg2.pkg3; which requires a directory structure as pkg1\pkg2\pkg3. The classes and methods of a package must be public.

➤ **Accessing a package:-**

To **access** package In a Java source file, **import** statements occur immediately. Following the **package** statement (if it exists) and before any class definitions.

**Syntax:**

**import *pkg1*[,*pkg2*].(*classname*|\*);**

Here, “pkg1” is the name of the top level package. “pkg2” is the name of package is inside the package1 and so on.

“classname”-is explicitly specified statement ends with semicolon.

- **Second way** to access the package

**import packagename.\*;**



Here, packagename indicates a single package or hierarchy of packages.  
The star(\*) denotes the compiler should search this entire package hierarchy when it encountered a class.

- **Access Specifiers /Access Modifiers in Package**

Access Modifier → Access Location ↓	Public	Private	Protected
Same class	Yes	Yes	Yes
Subclass in same package	Yes	No	Yes
Other classes in same package	Yes	No	Yes
Subclass in other packages	Yes	No	Yes
Non subclasses in other packages	Yes	No	No

➤ **Example of creating package and accessing the package.**

**package1:**

```
package package1;
public class Box
{
    int l= 5;
    int b = 7;
    int h = 8;
    public void display()
    {
        System.out.println("Volume is:"+(l*b*h));
    }
}
```

**Source file:**

```
import package1.Box;
class VolumeDemo
{
    public static void main(String args[])
    {
        Box b=new Box();
        b.display();
    }
}
```



- **Que:- Design a package containing a class which defines a method to find area of rectangle. Import it in java application to calculate area of a rectangle.**

```
package Area;
public class Rectangle
{
    double length,bredth;
    public double rect(float l,float b)
    {
        length=l;
        bredth=b;
        return(length*bredth);
    }
}

import Area.Rectangle;

class RectArea
{
    public static void main(String args[])
    {
        Rectangle r=new Rectangle( );
        double area=r.rect(10,5);
        System.out.println("Area of rectangle= "+area);
    }
}
```

- **Adding class to a package:-**

Consider the we have package 'P' and suppose class B is to be added to following package.

```
Package P;
{
    Public class X;
    {
        //body of X;
    }
}
```

The package 'P' contains one public class named as 'A'  
For adding class X to this package follows given steps

- 1) Define the class & make it as public.
- 2) Place package statement.



---

Package P1;  
Before the class definition as follows

```
Package P1;  
    Public class B  
    {  
        // Body of B  
    }
```

- 3) Store this as B.java file under directory P1;
- 4) Compile B.java file. This will create B.class file and place it in the directory P1.  
Now, the package “P2” contains both the classes A and B. So to import both classes use **import P1.\***

➤ **Very Important Questions:-**

- 1) Explain the types of Inheritance?
- 2) Write a java program to implements to implements multilevel inheritance with 4 levels of hiererachy?
- 3) What is the single level inheritance? Explain with suitable example?
- 4) Explain inheritance and polymorphism features of java?
- 5) Explain constructor overloading?
- 6) Explain method overriding?
- 7) Explain final variables and final methods.?
- 8) Describe the final method and final variables with respect to inheritance?
- 9) Explain abstract methods and classes?
- 10) Explain static members?
- 11) What is meant by an interface? State its need and write syntax and features of an interface. Give one example
- 12) State any four system packages along with their use.
- 13) Write the effect of access specifiers public, private and protected in package.
- 14) Explain with example how to achieve multiple inheritance with interface.
- 15) What is package ? State any four system packages along with their use ? How to add class to a user defined package ?
- 16) Write syntax of defining interface. Write any major two differences between interface and a class.
- 17) Design a package containing a class which defines a method to find area of rectangle. Import it in Java application to calculate area of a rectangle.



- 
- 18) What is interface ? How it is different from class ? With suitable program explain the use of interface.
- 19) What is package ? How to create package ? Explain with suitable example.
- 20) Write a program to implement following hierarchy

