Ch-3 Inheritance, Interface and Package Marks: 12

Content

3.1 Inheritance: Concept of inheritance and types of inheritance

3.2 Single inheritance, multilevel inheritance, Hierarchical inheritance, methods and constructor overloading and overriding. Dynamic method Dispatch, final variables, final methods, use of super, abstract method and classes, static members.

3.3 Interfaces: Define Interface, Implementing interface, accessing Interface, variables and methods, extending interface, interface reference, nested interface.

3.4 Package: Define package, type of package naming and creating packages, accessing package, import statement. Static import, adding class and interface to a package.

Inheritance

- Inheritance can be defined as the process where one class acquires the properties (methods and fields) of another.
- The class which inherits the properties of other is known as subclass (derived class, child class) and the class whose properties are inherited is known as superclass (base class, parent class).

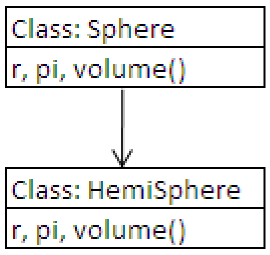
extends Keyword

- extends is the keyword used to inherit the properties of a class. Following is the syntax of extends keyword.
- Syntax

```
class super_class_name
{
------
------
}
class Sub_class_name extends super_class_name
{
------
------
}
```

Single Inheritance

- When a subclass is derived simply from its parent's class then this mechanism is known as single inheritance.
- In this type there is only one child class and one parent class.
- E.g



```
import java.lang.*;
class sphere
{
int r;
float pi;
sphere(int r, float pi)      //super class constructor
{
this.r=r;
this.pi=pi;
}
void volume()                          //method to display volume of sphere
{
float v=(4*pi*r*r)/3;
System.out.println("Volume of Sphere="+v);
}
}
class hemisphere extends sphere //subclass defined
```

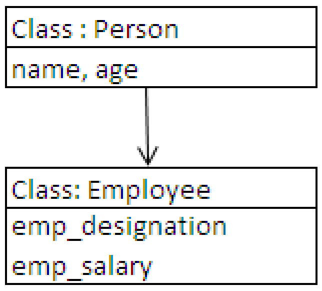
```
{
int a;
hemisphere(int r, float pi, int a)
{
super(r,pi);
this.a=a;
}
void volume()                    //method to display volume of hemisphere
{
super.volume();
float v=(a*pi*r*r*r)/3;
System.out.println("Volume of Hemisphere="+v);
}
}
class volume
{
public static void main(String args[])//main method
{
hemisphere h=new hemisphere(12, 3.14f, 2);
h.volume();
}
}
```

Output:

```
Volume of Sphere=602.88
Volume of Hemisphere=3617.28
```

---

2. e.g.



```
class person
{
String name;
int age;
void getdata(String n, int a)
{
name=n;
age=a;
}
void putdata()
{
System.out.println("Employee Details: ");
System.out.println("Employee Name: "+name);
System.out.println("Employee Age: "+age);
}
}
class emp extends person
{
String emp_desig;
int emp_sal;
void getdata(String n, int a, String e, int s)
{
super.getdata(n,a);
emp_desig=e;
emp_sal=s;
}
void putdata()
{
super.putdata();
System.out.println("Employee's Designation: "+emp_desig);
System.out.println("Employee's Salary; Rs. "+emp_sal);
}
}
class emp1
```

```
{
public static void main(String args[])
{
emp e1=new emp();
e1.getdata("vijay",30,"Manager",50000);
e1.putdata();
}
}
```

Output:

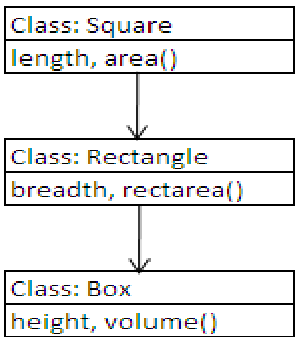
```
Employee Details:
Employee Name: vijay
Employee Age: 30
Employee's Designation: Manager
Employee's Salary; Rs. 50000
```

Multilevel Inheritance

- When a class is derived from already derived class then this mechanism is known as multilevel inheritance.
- The derived class is called as subclass or child class for its parents class and this parent class work as child class for its just above parent class.
- Multilevel inheritance can go up to any level

---

1. e.g.



```
import java.lang.*;
```

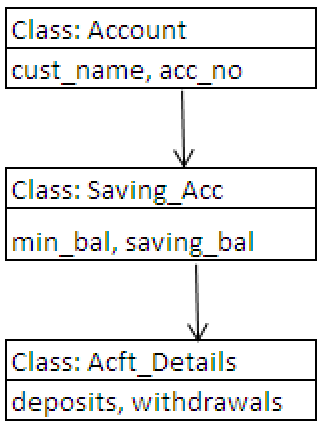
```
class Square         //super class
{
 int length;
Square(int x)
{
length=x;
}
void area() //method to calculate area of square
{
   int area=length*length;
   System.out.println("Area of Square="+area);
}
}
class Rectangle extends Square        //inherit class rectanglefrom class square
{
   int breadth;
   Rectangle(int x, int y)
{
   super(x); //calling super class constructor
   breadth=y;
}
void rectArea()
{
int area1=length*breadth;
System.out.println("Area of Rectangle="+area1);
}
}
class Box extends Rectangle    //inherit class Box from class Rectangle
{
  int height;
  Box(int x, int y, int z)   //calling rectangle class constructor
{
  super(x,y);
  height=z;  }
void volume()//method to display volume of box
{
 int volume=length*breadth*height;
  System.out.println("Volume of Box="+volume);
}
}
class Shape
{
public static void main(String args[])   //main method
{
Box b=new Box(10,20,30);
//creating object of Box class
b.volume();
b.rectArea();
b.area();
}
}
```

Output:

```
Volume of Box=6000
Area of Rectangle=200
Area of Square=100
```

---

2. e.g.



```
class account
{
String custname;
int accno;
void getdata(String n, int a)
{
custname=n;
accno=a;
}
void putdata()
{
System.out.println("Customers Account Details\n");
System.out.println("Customers Name: "+custname);
System.out.println("Customers Account Number: "+accno);
}
}
class savingacc extends account
{
int minbal;
int savbal;
void getdata(String n, int a, int m, int s)
{
super.getdata(n,a);
minbal=m;
savbal=s;
}
void putdata()
{
super.putdata();
System.out.println("Minimum Balance: "+minbal);
System.out.println("Saving Balance: "+savbal);
}
}
class accdetail extends savingacc
```

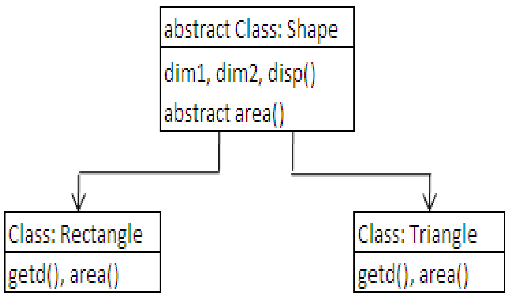
```
{
int deposits;
int withdrawal;
void getdata(String n, int a, int m, int s, int d, int w)
{
super.getdata(n,a,m,s);
deposits=d;
withdrawal=w;
}
void putdata()
{
super.putdata();
System.out.println("Deposit: "+deposits);
System.out.println("Withdrawal Amount: "+withdrawal);
}
}
class acc
{
public static void main(String args[])
{
accdetail a1=new accdetail();
a1.getdata("Vijay",123,500,10000,5000,2500);
a1.putdata();
}
}
```

Output:

```
Customers Account Details
Customers Name: Vijay
Customers Account Number: 123
Minimum Balance: 500
Saving Balance: 10000
Deposit: 5000
Withdrawal Amount: 2500
```

Hierarchical Inheritance

- In hierarchical inheritance one class is extended by many subclasses.
- It is one to many relationships.
- Many programming problems can be tasked into a hierarchy where certain features of one level are shared by many others below the level.



```
import java.io.*;
abstract class shape
{
float dim1,dim2;
void getdata()
{
DataInputStream d=new DataInputStream(System.in);
try
{
System.out.println("Enter the value of Dimension1: ");
dim1=Float.parseFloat(d.readLine());
System.out.println("Enter the value of Dimension2: ");
dim2=Float.parseFloat(d.readLine());
}
catch(Exception e)
{
System.out.println("General Error"+e);
}
}
void disp()
```

```
{
System.out.println("Dimension1= "+dim1);
System.out.println("Dimension2= "+dim2);
}
abstract void area();
}
class rectangle extends shape
{
double area1;
void getd()
{
super.getdata();
}
void area()
{
area1=dim1*dim2;
System.out.println("The Area of Rectangle is: "+area1);
}
}
class triangle extends shape
{
double area1;
void getd()
{
super.getdata();
}
void area()
{
area1=(0.5*dim1*dim2);
System.out.println("The Area of Triangle is: "+area1);
}
}
class methodover1
{
```

```
public static void main(String args[])
{
rectangle r=new rectangle();
 System.out.println("For Rectangle");
 r.getd();
 r.disp();
 r.area();
triangle t=new triangle();
 t.getd();
 t.disp();
 t.area();
}
}
```

Output:

```
For Rectangle
Enter the value of Dimension1:
Enter the value of Dimension2:
Dimension1= 5.0
Dimension2= 6.0
The Area of Rectangle is: 30.0
Enter the value of Dimension1:
Enter the value of Dimension2:
Dimension1= 6.0
Dimension2= 7.0
The Area of Triangle is: 21.0
```

Method and Constructor Overloading

- 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.
- Method overloading allows user to achieve the compile time polymorphism.

Syntax:

- 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

For e.g. 1

```
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));
}
}
```

For e.g. 2

```
class Overload
{
void test (int a)
{
System.out.println("a"+a);
}
void test (int a, int b)
{
System.out.println("a & b"+a "and" +b);
}
double test(double a)
{
System.out.println("Double a" +a);
Return a*a;
}
}
class demo
{
public static void main(String args[])
{
Overload obj = new Overload();
double result;
obj.test(10);
obj.test(10, 20);
result= obj.test(5.5);
System.out.println("result is:"+result);
}
}
```

Method Overloading

- There may be situation when we want an object to respond to the same method but have different behavior when that method is called.
- This is possible by defining a method in the subclass that has the same name, same arguments and same return type as a method in the superclass.
- Then, when that method is called, the method defined in the subclass is invoked and executed instead of the one in the superclass. This is known as overriding.

- In overriding return types and constructor parameters of method should match.

For E.g. No.1

```
class Super
{
int x;
Super (int x)
{    this.x=x;   }
void display()
{ System.out.println(“Super x=”‖+x); }
}
Class Sub extends Super
{
int y; Sub( int x, int y)
{
super (x);
this.y=y;
}
 void display()
{
System.out.println(“Super x=”‖ +x);
System.out.println(“Sub y=‖” +y);
}
}
class overrideTest
{
public static void main(String arg[])
{
Sub s1= new Sub(100,200);
s1.display();
}
}
```

For e.g. No.2

```
class A
{
int i;
A(int a, int b)
{
i=a+b;
}
void add()
{
System.out.println("Sum of a & b is=" +i);
}
}
class B extends A
{
int j;
B(int a, int b, int c)
{
super(a,b);
j=a+b+c;
}
void add()
{
super.add();
System.out.println("Sum of a, b & c"+j);
}
}
class MethodOverride
{
public static void main(String args[])
{
B b=new B(10,20,30);
b.add();
}
}
```

Method Overloading	Method Overriding
It happens within the same class	It happens between super class and subclass
Method signature should not be same	Method signature should be same.
Method can have any return type	Method return type should be same as super class method
Overloading is early binding or static binding	Overriding is late binding or dynamic binding
They have the same name but, have different parameter lists, and can have different return types.	They have the same name as a superclass method. They have the same parameter list as a superclass method. They have the same return type as a superclass method
It is resolved at compile time	It is resolved at runtime.
Inheritance does not blocked by method overloading.	Method overriding blocks the inheritance.
One method can overload unlimited number of times.	Method overriding can be done only once per method in the sub class.

Dynamic Method Dispatch

- Dynamic method dispatch is the mechanism by which a call to a overridden method is resolved at runtime rather than at compile time
- It forms the basis for runtime polymorphism
- When an overridden method is called through a super class reference, java determines which method to execute based upon the type of the object being referred at the time the call is made.

Example:

```
	import java.lang.*;
	class A
	{
	void show()
	{
	System.out.println("in show of A");
	}
	}

	class B extends  A
	{
	void show()
	{
	System.out.println("in show of B");
	}
	}
class C extends  B
	{
	void show()
	{
	System.out.println("in show of C");
	}
	}
	class DynaMethod
	{
	public static void main(String args[])
	{
	A  a = new A();             // object of class A
	B  b = new B();             // object of class B
	  C  c = new C();           // object of class C
	A r;                   // obtain a reference of type A
	r = a;                 //r refer to object of A
	r.show();              // call show() of A
r = b;                  //r refer to object of B
r.show();               // call show() of B
r = c;          //r refer to object of C
r.show();       // call show() of C
}
}
```

Output:

```
in show of A
in show of B
in show of C
```

Reference of class A stores the reference of class A, class B and class C alternatively. Every call to show() method would result in execution of show() method of that class whose reference is currently stored by a reference of class A.

Final Variable and Final Method

- **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.

  Example 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.

  Syntax:

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

  Example of declaring a final method:

  ```
  class A
  {
  ```

```
final void show()
{
System.out.println("in show of A");
}
}
class B extends A
{
void show() // can not override because it is declared with final
{
System.out.println("in show of B");
}
}
```

Use of Super Key

- When we create super class that keeps the details of its implementation to itself (making it private). Sometime the variable and methods of super class are private, so they will be available to subclass but subclass cannot access it.
- In such cases there would be no way for a subclass to directly access these variables on its own. Whenever a subclass needs a reference to it immediate super class, it can do so by use of the keyword super.
- Super have two general forms:
    a. The first calls the super class's constructor.
    b. The second is used to access a member (variable or method) of the super class that has been declared as private.

Example use of super() class

```
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();
}
}
```

Abstract Class

- A class that is declared with abstract keyword is known as abstract class in java. It can have abstract and non-abstract methods (method with body).
- Abstraction is a process of hiding the implementation details and showing only functionality to the user. Another way, it shows only important things to the user and hides the internal details.
- A method must be always redefined in a subclass of an abstract class, as abstract class does not contain method body. Thus abstract makes overriding compulsory.
- Class containing abstract method must be declared as abstract.
- You cannot declare abstract constructor, and so, objects of abstract class cannot be instantiated.

  Syntax :

  ```
  abstract class < classname>
  {
  abstract method1(…);
  method2(….);
  ```

```
        }
```

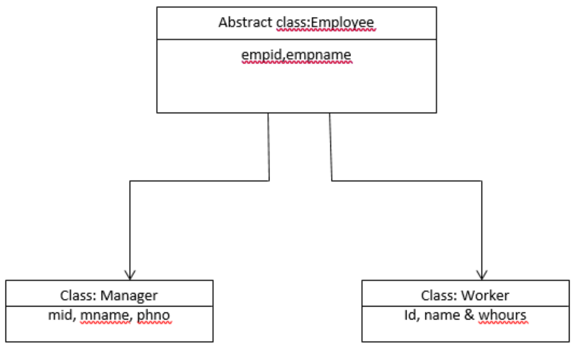
Example:1

```
abstract class A
{
abstract void disp();
void show()
{
System.out.println("show method is not abstract");
} }
class B extends A
{
void disp()
{
System.out.println("inside class B");
}
}
class test
{
public static void main(String args[])
{
B b = new B();
b.disp();
b.show();
}
}
```

Output :

```
show method is not abstract
inside class B
```

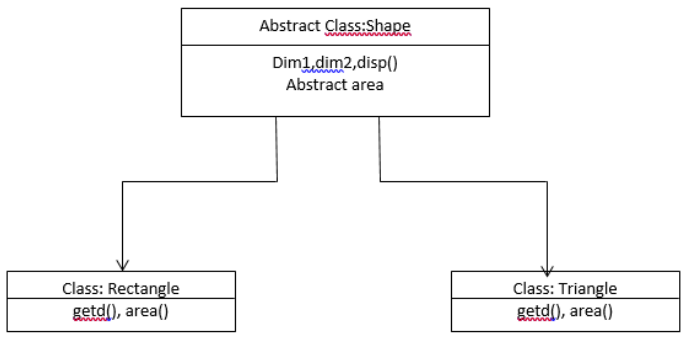
Example: 2



```
abstract class Employee
{
int empid;
String empname;
Employee(int e,String n)
{
empid=e;
empname=n;
}
void display()
{
System.out.println("Employee ID is: "+empid);
System.out.println("Employee name is: "+empname);
}
}
class Manager extends Employee
{
int mid;
String mname;
int phno;
Manager(int e,String n,int m,String mn,int p)
{
super(e,n);
mid=m;
mname=mn;
phno=p;
}
```

```
void displaym()
{
super.display();
System.out.println("Manager ID is: "+mid);
System.out.println("Manager name is: "+mname);
System.out.println("Phone no is: "+phno);
}
}
class Worker extends Employee
{
int wid;
String wname;
double whours;
Worker(int e,String n,int w,String wn,double wh)
{
super(e,n);
wid=w;
wname=wn;
whours=wh;
}
void displayw()
{
super.display();
System.out.println("worker ID is: "+wid);
System.out.println("worker name is: "+wname);
System.out.println("Working hours= "+whours);
}
}
class Company
{
public static void main(String arg[])
{
Manager m1=new Manager(11,"Neel",121,"Sameer",123456);
m1.displaym();
Worker w1=new Worker(22,"Ishan",222,"Priyank",6);
w1.displayw();
}
}
```

Example: 3



```
import java.io.*;
abstract class shape
{
float dim1,dim2;
void getdata ()throws IOException
{
BufferedReader br=new BufferedReader (new InputStreamReader (System.in));
System.out.println("Enter dimension 1");
dim1 =Float.parseFloat(br.readLine());
System.out.println("Enter dimension 2");
dim2=Float.parseFloat(br.readLine());
}
void disp()
{
```

```
System.out.println("dimension 1="+dim1);
System.out.println("dimension="+dim2);
}
abstract void area();
}
class Rectangle extends shape
{
double area1;
float dim1,dim2;
void getd() throws IOException
{
super.getdata();
}
void area()
{
area1=dim1*dim2;
System.out.println("area of rectangle is"+area1);
}
}
class triangle extends shape
{
double area;
float dim1,dim2;
void getd1() throws IOException
{
super.getdata();
}
void area()
{
area=(0.5*dim1*dim2);
System.out.println("area of triangle is"+area );
}
}
class abs
```

```
{
public static void main(String args[])  throws IOException
{
Rectangle r =new Rectangle();
System.out.println("for Rectangle:");
r.getd();
r.disp();
r.area();
System.out.println("for triangle:");
triangle t =new triangle ();
t.getd1();
t.disp();
t.area();
}
}
```

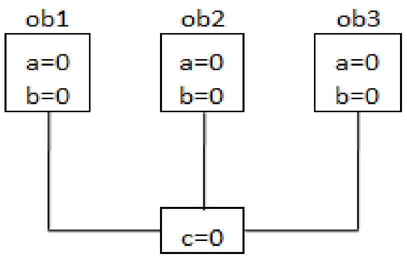
Static Members

- When a number of objects are created from the same class, each instance has its own copy of class variables. But this is not the case when it is declared as **static**.
- **Static** method or a variable is not attached to a particular object, but rather to the class as a whole. They are allocated when the class is loaded.
- If some objects have some variables which are common, then these variables or methods must be declared static.
- To create a static variable, precede its declaration with keyword static. Static variables need not be called on any object.

Example:

```
int a;          // normal variable
static int a; //static variable
class s
{
int a,b;
static int c;
}
class statictest
{
public static void main(String args[])
{
s ob1=new s();
s ob2=new s();
s ob3=new s();
}
}
```

In above program variable a and b are declared as int variable whereas the variable c is declared as static variable. Static variable will be common between all the objects.



When we accessing variable as ‘a’ we cannot access it directly without the reference of the object because ‘a’ exist in all three objects. So to access the value of ‘a’ we must link it with the name of the object. But the ‘c’ is static variable and it is common between all the objects so we can access it directly. But when we are accessing it outside the class in which it is declared, we must link it with the name of the class i.e. with S as S.c=20;

Sr.No.	Static	Final
1.	Static keyword can be applied to instance variables and methods but not to the classes.	Final keyword can be applied to all constructors that is variable, methods and classes.
2.	Static variable can be reinitialized.	Final variable can be reinitialized.
3.	Static variable is global variable shared by all the instances of	Final variable is a constant variable and it cannot be

	object and it has only single copy.	changed.
4.	Static variable can change their value.	Final variables cannot be changed because they are constants.

3.2 Interfaces

- An interface is collection of abstract method and it is defined much like class.
- Writing an interface is similar to the class, with abstract methods and final field. This means that interface do not specify any code to implement those methods & data fields containing only constants.
- Interfaces cannot be instantiated they can be only implemented by the classes or extended by other
  Interfaces.
- Interfaces are design to support dynamic method resolution at run time.
- The general from of interface is

```
access interface interface_name
{
variable declaration;
method declaration;
}
```

- Access is either public or not used when no access specifier is included, then default access result and interface is only available to other member of the package in which it is declared.
- For e.g.-

```
public interface IF4IA
{
String Lecture=”JPR”;
int benches=35;
void display();
}
```

Implementing Interface

- A class can only inherit from one super class. However, a class may implement several interfaces.
- To declare a class that implements an interface, you need to include ‘implements’ clause in the class declaration.
- Your class can implements keyword is followed by a ‘comma separated list’ of the interface implemented by a class.

  Syntax:- class class_name implements interface
  ```
  {
  //body
  }
  ```
  OR

  class class_name extends class_name implemnts interface 1,2

  For e.g.-
  ```
  inteface shape
  {
  public String baseclass="Shape";
  public void draw();
  }
  class circle implements shape
  {
  public void draw()
  {
  System.out.println("Circle drawn here");
  }
  ```

```
}
```

Accessing Interface Variables and Method

- An interface can use to declare set of constants that can be used in different classes. This is similar to creating header files in c++ to contain a large no. of constants.
- Such interface does not contain methods, there is no need to warry about implementing methods.
- All variable declared in interface must be constant.

  For e.g._

```
interface one
{
int x=12;
}
interface two
{
int y=10;
void display();
}
class demo implemets one, two
{
int a=x;
int b=y;
public viod display()
{
System.out.println("x in interfece one" +x);
System.out.println("y in interfece one" +y);
System.out.println("x & y"+(x+y));
}
public void disp()
{
System.out.println("Value access from interface one:" +a);
System.out.println("Value access from interface two:" +b);
}
}
class multipleinterface
{
public static void main(String args[])
{
demo d= new demo();
d.display();
d.disp();
}
}
```

Extending Interface

- An interface can extend another interface, similarly to the way that a class can extend another class.
- The extends keywords is used to extend an interface, and the child interface inheritas the methods of the parent interface.
- Syntax: -

```
interface class2 ectends class1
{
//body
}
```

For e.g.:-

```
inteface A
{
void show();
}
interface B extends A
{
void display();
}
class C implements B
public void show()
```

```
{
System.out.println("calling interface A");
}
public void display()
{
System.out.println("calling interface B");
}
public void disp()
{
System.out.println("class C is working");
}
}
class D
{
Public static void main(String a[])
{
C object=new C();
object.show();
object.display();
object.disp();
}}
```

How to achieve multiple inheritance?

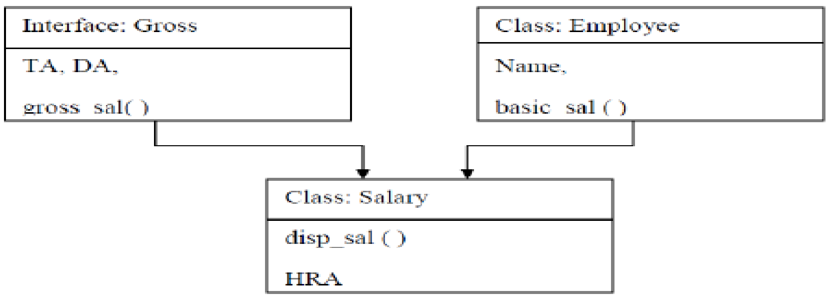
- It is type of interface where a derived class may have more than one parent class.
- It is not possible in case of Java as you cannot have two parent classes at the parent level, instead there can be one class and one interface at parent level to achieve multiple inheritance.
- Interface is similar to classes but can contain final variable and abstract methods. Interfaces can be implemented to a derived class.
- For e.g : -

```
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);
}
public static void main(String args[])
{
result r= new result(101,"abc",75,75);
```

```
        r.disp();
        }
        }
```

Example No. 2



```
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()
{
Super.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();
}
}
```

Nested Interface

- An interface which is declared within another interface or class is known as nested interface.
- The nested interface is used to group related interfaces so that they can be easy to maintain.
- The nested interface must be referred by the outer interface or class.
- A nested interface can be declared as public, private or protected.

- Syntax: -

1. Nested interface declared within interface

```
interface interface_name
{
inetrface nested_interface_name
{
-------
-------
}
}
```

For E.g.

```
interface display
{
void show();
interface message
{
void msg();
}
}
class test implemets display.message
{
public void msg()
{
System.out.println("Hello nested interface");
}
public static void main(String a[])
{
display.message message= new test();
message.msg();
}
}
```

---

2. Interface within class

```
interface class_name
{
inetrface nested_interface_name
{
-------
-------
}
}
```

For E.g

```
class A
{
interface message
{
void msg();
}
}
class test implemets A.message
{
public void msg()
{
System.out.println("Hello nested interface");
}
public static void main(String a[])
{
display.message message= new test();
message.msg();
}
}
```

Class	**interface**
Classes are not used to implement multiple inheritence.	The interfaces are used in java to implementing the concept of multiple inheritance.
The member of a class can be constant	The members of an interface are

or variables.	always declared as constant i.e. their values are final.
The class definition can contain the code for each of its methods. That is the methods can be abstract or non-abstract.	The methods in an interface are abstract in nature. I.e. there is no code associated with them. It is defined by the class that implements the interface.
Class contains executable code.	Interface contains no executable code.
Memory is allocated for the classes.	We are not allocating the memory for the interfaces.
We can create object of class.	We can‵t create object of interface.
Classes can be instantiated by declaring objects.	Interface cannot be used to declare objects. It can only be inherited by a class.
Classes can use various access specifiers like public, private or protected.	Interface can only use the public access specifier.
Class contains constructors.	An interface does not contain any constructor.
Classes are always extended.	Interfaces are always implemented.
A class can extend only one class (no multiple inheritance), but it can implement many interfaces.	Interfaces can extend one or more other interfaces. Interfaces cannot extend a class, or implement a class or interface.
An abstract class is fast.	An interface requires more time to find the actual method in the corresponding classes.

Package

- Package is nothing more than the way we organize files into different directories according to their functionality, usability as well as category they should belongs to.

- Packaging also helps us to avoid class name collision when we use the same class name as that of others.
- Packages act as containers for classes.
- Syntax: -

  ```
  package pakage_name;
  For e.g.
  package A;
  class ABC
  {
  ------
  ------
  }
  ```

Types of packages: -

1. Java API 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.

Classes stored in package can be accessed in two ways

- import java.awt.font;
- import java.awt.*;

Naming Convention:

- Package names are written in all lower case to avoid conflict with the names of classes or interfaces.
- All class name begins with an uppercase letter and method name begin with lowercase letters
- For e.g. :- double x=java lang.Math.sqrt(x); where lang is package name and Math is class name

Creating Package:

- Declare a package at beginning of a file using syntax: - package package_name;
- Define the class that is to be put inside the package and declare it as public.
- Create subdirectory under the directory where the main source files are stored.
- Store the listing as classname.java files in the subdirectory created.
- Compile the file, this creates .class file in the subdirectory.

Accessing Package:

- Package can be accessed using keyword import.
- There are 2 ways to access java system packages:
    - Package can be imported using import keyword and the wild card(*) but drawback of this shortcut approach is that it is difficult to determine from which package a particular member name.
      Syntax: import package_name.*;
      For e.g. import java.lang.*;
    - The package can be accessed by using dot(.) operator and can be terminated using semicolon(;)
      Syntax: import package1.package2.classname;
      For e.g. VP.IF.IF4I.IF4IA

Example 1: Develop a program named myInstitute include class name as department with one method to display the staff of that department. Develop a program to import this package in java application and call the method define in the package.

```
package myInstitute;
public class department
{
public void display()
{
System.out.println("leon");
System.out.println("shreyas");
}
(main code)
import myInstitute.*;
public class demo
{
public static void main(String a[])
{
department d=new department();
d.display();
}
}
```

Example 2:

Develop a program which consist of the package named let_me_calculate with a class named calculator and a method name add to add two integer no. import let_me_calculate package in anather program to add two no.

```
package let_me_calculate;
public class calculate
{
public void add()
{
int a=20;
int b=30;
int c=a+b;
System.out.println("addition="+c);
}
}
import let_me_calculate.*;
public class calculator
{
public static void main(String arg[])
{
calculate c=new calculate();
c.add();
}
}
```

‘Import’ Statement: -

i) The import keyword is used to import built-in packages & user defined packages. So that your class can refer a class that is in another package by directly using its name.

ii) There are different ways to refer to a class that is present in different packages:-

1. Import only the class you want to use

   For eg:-

   ```
   Import java.util.Date;
   Class MyDate extends Date
   {
   //Body of Class
   }
   ```

2.) Import all the classes from particular package.

   For eg:-

   ```
   import java.util.*;
   Class MyDate extends Date
   {
   //Body of Class
   }
   ```

Static Import:-

i) Static import is a feature that expands capabilities of import keywords. It is used to import Static member of a class.
ii) Using static import, it is impossible to refer to static member directly without its class name.
iii) There are 2 general forms of Static import statement

1. Import only a single static member of a class:-
   Syntax:
   import static package.classname static membername;
   For eg:-
   Import static java.lang.Math.sqrt;
2. Import all static members a class.
   Syntax:
   Import static package.classname.*;
   For Eg:-

<u>Adding class to a package</u>

Structure:-
Package p1
public class A
{
/body of class
}
Create a file outside of package which consists of main method
import p1.*;
Class B
{
public static void main (String args[])
{
// create object of class which is in package
}
}