**PART-1**

**INHERITANCE**

SUBCLASS

Java classes can be reused by creating new classes that are built upon existing ones. The mechanism of deriving a new class from an old one is called inheritance. The old class is called base class or super class or parent class and the new one is called the subclass or derived class or child class.

Subclasses inherit all the variables and methods of their parent classes.

Syntax of Defining a Subclass

**class** Subclassname **extends** Superclassname

{

   //methods and fields

}

The keyword **extends** indicates that the properties of Superclassname are extended to Subclassname. The Subclass will now contain its own variables and methods as well as those of the Superclass. This helps to add some properties to an existing class without actually modifying it.

SUPER KEYWORD

The **super** keyword in java is a reference variable that is used to refer immediate parent class object.

Whenever an instance of subclass is created, an instance of parent class is created implicitly i.e. referred by super reference variable.

## Usage of java super Keyword

1. super is used to refer immediate parent class instance variable.
2. super() is used to invoke immediate parent class constructor.
3. super is used to invoke immediate parent class method.

// super keyword in subclass is used to refer immediate parent class instance variable.

class Vehicle{

int speed=50;

}

class Bike4 extends Vehicle

{ int speed=100;

void display()

{ System.out.println(super.speed);//will print speed of Vehicle now

System.out.println(speed);//will print speed of Bike4 now

}

public static void main(String args[])

{ Bike4 b=new Bike4();

b.display();

}

} OUTPUT : 50

100

**SUBCLASS CONSTRUCTOR**

A subclass constructor is used to construct the instance variables of both the subclass and the superclass. The subclass constructor uses the keyword super to invoke the constructor method of the superclass. The call to the superclass constructor must appear as the first statement within the subclass constructor and the parameters in the **super** call must match the order and type of the instance variable declared in the superclass.

Example program

class Room

{ int l,b;

Room(int x, int y)

{ l=x;b=y; }

int area()

{ return(l\*b); }

}

class Bedroom extends Room //Inheriting Room

{ int h;

Bedroom(int x,int y, int z)

{ super(x,y); // Pass values to superclass

h = z;

}

int volume()

{ return(l\*b\*h); }

}

class subclasscons

{ public static void main(String args[])

{ Bedroom room1=new Bedroom(10,15,20);

int a=room1.area(); //superclass method

int v=room1.volume(); //subclass method

System.out.println("Area= "+a);

System.out.println("Volume="+v);

}

}

OUTPUT: Area=150

Volume=3000

**SINGLE INHERITANCE**

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

Example Program on Single Inheritance

class A

{ public void methodA()

{ System.out.println("Base class method");

}

}

class B extends A

{ public void methodB()

{ System.out.println("Child class method");

}

}

class single

{ public static void main(String args[])

{ B obj = new B();

obj.methodA(); //calling super class method

obj.methodB(); //calling local method

}

}

OUTPUT:

Base class method

Child class method

**MULTILEVEL INHERITANCE**

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

Example Program on Multilevel Inheritance

class X

{ public void methodX()

{ System.out.println("Class X method"); }

}

class Y extends X

{ public void methodY()

{ System.out.println("class Y method"); }

}

class Z extends Y

{ public void methodZ()

{ System.out.println("class Z method"); }

}

class multilevel

{

public static void main(String args[])

{ Z obj = new Z();

obj.methodX(); //calling grand parent class method

obj.methodY(); //calling parent class method

obj.methodZ(); //calling local method

}

}

OUTPUT :

Class X method

Class Y method

Class Z method

**HIERARCHICAL INHERITANCE**

In this type of inheritance one class is inherited by many**sub classes**.

Example Program on Hierarchical Inheritance

class A

{ int a=10; }

class B extends A

{ int b=30;

void display()

{ System.out.println("The values are:"+a+" "+b); }

}

class C extends A

{ int c=40;

void show()

{ System.out.println("The values are:"+a+" "+c); }

}

class hierarchicaldemo

{ public static void main(String args[])

{ B obj1=new B();

obj1.display();

C obj2=new C();

obj2.show();

}

}

OUTPUT:

The values are:10 30

The values are:10 40

**Method Overriding: Design a vehicle class hierarchy in Java, and develop a program to demonstrate Polymorphism.**

import java.io.\*;

class Vehicle

{ String regno;

int model;

Vehicle(String r, int m)

{ regno=r;

model=m;

}

void display()

{ System.out.println("registration no:"+regno);

System.out.println("model no:"+model);

}

}

class Twowheeler extends Vehicle

{ int noofwheel;

Twowheeler(String r,int m,int n)

{ super(r,m);

noofwheel=n;

}

void display()

{ System.out.println("Two wheeler tvs");

super.display();

System.out.println("no of wheel" +noofwheel);

}

}

class Threewheeler extends Vehicle

{ int noofleaf;

Threewheeler(String r,int m,int n)

{ super(r,m);

noofleaf=n;

}

void display()

{ System.out.println("three wheeler auto");

super.display();

System.out.println("no of leaf" +noofleaf);

}

}

class Fourwheeler extends Vehicle

{ int noofleaf;

Fourwheeler(String r,int m,int n)

{ super(r,m);

noofleaf=n;

}

void display()

{ System.out.println("four wheeler car");

super.display();

System.out.println("no of leaf" +noofleaf);

}

}

public class Vehicledemo

{ public static void main(String arg[])

{ Twowheeler t1;

Threewheeler th1;

Fourwheeler f1;

t1=new Twowheeler("TN74 12345", 1,2);

th1=new Threewheeler("TN74 54321", 4,3);

f1=new Fourwheeler("TN34 45677",5,4);

t1.display();

th1.display();

f1.display();

}

}

OUTPUT :

Two wheeler tvs

registration no:TN74 12345

model no:1

no of wheel2

three wheeler auto

registration no:TN74 54321

model no:4

no of leaf3

four wheeler car

registration no:TN34 45677

model no:5

no of leaf4

**Final Keyword In Java**

The **final keyword** in java is used to restrict the user. The java final keyword can be used in many contexts. Final can be:

1. variable
2. method
3. class

The final keyword can be applied with the variables, a final variable that has no value it is called blank final variable or uninitialized final variable. It can be initialized in the constructor only. The blank final variable can be static also which will be initialized in the static block only.

**1) Java final variable**

A variable declared as final, cannot change its value(It will be constant).

**Example of final variable**

There is a final variable speedlimit, trying to change the value of this variable is not possible because final variable once assigned a value can never be changed.

class FinalVariable

{ final int speedlimit=90;//final variable

void run()

{ speedlimit=400; }

}

class Bike1 extends FinalVariable

{ int speedlimit=80;

public static void main(String args[])

{ Bike1 obj=new Bike1();

obj.run();

System.out.println("obj"+ obj.speedlimit);

}

}

OUTPUT:

FinalVariable.java:5: error: cannot assign a value to final variable speedlimit

speedlimit=400;

^

1 error

**2) Java final method**

Final method is inherited but cannot be overridden.

class finalmethod

{ final void demo()

{ System.out.println("finalmethod Class Method"); }

}

class finalmethoddemo extends finalmethod

{ void demo()

{ System.out.println("finalmethoddemo Class Method"); }

public static void main(String args[])

{ finalmethoddemo obj= new finalmethoddemo();

obj.demo();

}

}

OUTPUT:

finalmethoddemo.java:11: error: demo() in finalmethoddemo cannot override demo()

in finalmethod

void demo()

^

overridden method is final

1 error

**3) Java final class**

Any class declared as final, cannot be extended.

final class XYZ

{ void demo()

{ System.out.println("My Method"); }

}

class finalclassdemo extends XYZ

{ public static void main(String args[])

{ finalclassdemo obj= new finalclassdemo();

obj.demo();

}

}

OUTPUT:

finalclassdemo.java:8: error: cannot inherit from final XYZ

class finalclassdemo extends XYZ

^

1 error

**ABSTRACT METHODS AND CLASSES**

An abstract class is a class that is declared abstract—it may or may not include abstract methods. Abstract classes cannot be instantiated, but they can be subclassed.

An abstract method is a method that is declared without an implementation (without braces, and followed by a semicolon), like this:

abstract void moveTo(double deltaX, double deltaY);

If a class includes abstract methods, then the class itself must be declared abstract, as in:

public abstract class GraphicObject {

// declare fields

// declare nonabstract methods

abstract void draw();

}

When an abstract class is subclassed, the subclass usually provides implementations for all of the abstract methods in its parent class. However, if it does not, then the subclass must also be declared abstract.

Example of Abstract class and Abstract Methods

abstract class Bike

{ Bike()

{ System.out.println("bike is created"); }

abstract void run();

void changeGear()

{ System.out.println("gear changed"); }

}

class Honda extends Bike

{ void run()

{ System.out.println("running safely.."); }

}

class abstractdemo1

{ public static void main(String args[])

{ Honda obj = new Honda();

obj.run();

obj.changeGear();

}

}

OUTPUT:

bike is created

running safely..

gear changed

**VISIBILITY CONTROL**

|  |
| --- |
|  |

It is possible to inherit all the members of a class by a subclass using the keyword extends. The variables and methods of a class are visible everywhere in the program. However, it may be necessary in some situations where we may want them to be not accessible outside. We can achieve this in Java by applying visibility modifiers to instance variables and methods. The visibility modifiers are also known as access modifiers. Access modifiers determine the accessibility of the members of a class.

Java provides the following visibility/access modifiers:

* Public
* Friendly/Package(default)
* Private
* Protected

They provide different levels of protection as described below.

**Public Access:** Any variable or method is visible to the entire class in which it is defined. But, to make a member accessible outside with objects, we simply declare the variable or method as public. A variable or method declared as public has the widest possible visibility and accessible everywhere.

**Friendly Access (Default):** When no access modifier is specified, the member defaults to a limited version of public accessibility known as "friendly" level of access. The difference between the "public" access and the "friendly" access is that the public modifier makes fields visible in all classes, regardless of their packages while the friendly access makes fields visible only in the same package, but not in other packages.

**Protected Access:** The visibility level of a "protected" field lies in between the public access and friendly access. That is, the protected modifier makes the fields visible not only to all classes and subclasses in the same package but also to subclasses in other packages

**Private Access:** Private fields have the highest degree of protection. They are accessible only with their own class. They cannot be inherited by subclasses and therefore not accessible in subclasses. In the case of overriding public methods cannot be redefined as private type.

**Private protected Access:**A field can be declared with two keywords Private and Protected together. This gives a visibility level in between the "protected" access and "private" access. This modifier makes the fields visible in all subclasses regardless of what package they are in. Remember, these fields are not accessible by other classes in the same package.

The following table summarizes the visibility provided by various access modifiers.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Access modifier 🡪** | **public** | **protected** | **friendly** | **private protected** | **private** |
| **Own class** | ✔ | ✔ | ✔ | ✔ | ✔ |
| **Sub classin same package** | ✔ | ✔ | ✔ | ✔ | 🗶 |
| **Other classesIn same package** | ✔ | ✔ | ✔ | 🗶 | 🗶 |
| **Sub classin other package** | ✔ | ✔ | 🗶 | ✔ | 🗶 |
| **Other classesIn other package** | ✔ | 🗶 | 🗶 | 🗶 | 🗶 |

**Example 01**

abstract class Vehicle

{

abstract void speed();

abstract void mil();

void ac()

{

System.out.println("Ac is only to Car");

}

}

class Bike extends Vehicle

{

void speed()

{

}

void mil()

{

System.out.println("65 Kmpl");

}

}

class Car extends Vehicle

{

void speed()

{

}

void mil()

{

System.out.println("18 Kmpl");

}

}

public class TestAbs1

{

public static void main(String arg[])

{

Vehicle ref;

ref = new Bike();

ref.mil();

ref.ac();

}

}

**PART-2**

**INTERFACES**

**Interface**

Interface looks like class but it is not a class. Using interface we can fully abstract a class interface from its implementation. An interface can have methods and variables just like the class but the methods declared in interface are by default abstract (only method signature, no body). Also, the variables declared in an interface are public, static & final by default. Once it is defined a number of classes can implement an interface .Also, one class can implement any number of interfaces.

To implement a interface a class must create complete set of methods defined by the interface.

**What is the use of interfaces?**

As mentioned above they are used for abstraction. Also, java programming language does not support multiple inheritances, using interfaces we can achieve this as a class can implement more than one interfaces, however it cannot extend more than one classes.

**Declaration**

Interfaces are declared by specifying a keyword “interface”.

**interface interfacename**

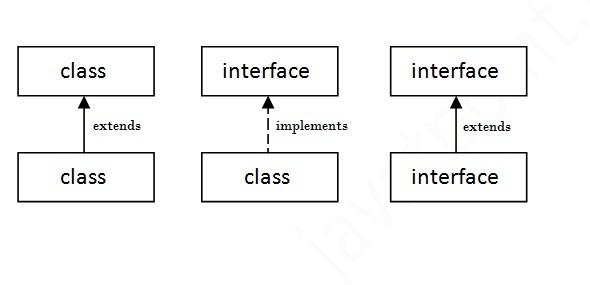
**{**

**variable declaration;**

**Methods declaration;**

**}**

**Relationship between classes and interfaces.**



**Example 1: Simple interface implementation**

interface MyInterface

{

public void method1();

}

class interfacedemo implements MyInterface

{

public void method1()

{

System.out.println("implementation of method1");

}

public static void main(String arg[])

{

MyInterface obj = new interfacedemo();

obj. method1();

}

}

**Output:**

implementation of method1

**Example 2: An interface cannot implement another interface. It has to extend the other interface if required.**

interface Myinterface1

{

public void method1();

}

interface Myinterface2 extends Myinterface1

{

public void method2();

}

class interfacedemo2 implements Myinterface2

{

public void method1()

{

System.out.println("implementation of method1");

}

public void method2()

{

System.out.println("implementation of method2");

}

public static void main(String arg[])

{

Interfacedemo2 obj = new interfacedemo2();

obj. method1();

obj. method1();

}

}

**Output:**

implementation of method1

implementation of method2

**Example 3: An interface can be implemented by any number of classes.**

interface area

{

final static float pi=3.14f;

float compute(float x,float y);

}

class rectangle implements area

{

public float compute(float x,float y)

{

return(x\*y);

}

}

class circle implements area

{

public float compute(float x,float y)

{

return(pi\*x\*x);

}

}

class interfacetest

{

public static void main(String args[])

{

rectangle rect=new rectangle();

System.out.println("area of rect="+rect.compute(10,10));

circle cir=new circle();

System.out.println("area of circle="+cir.compute(10,0));

}

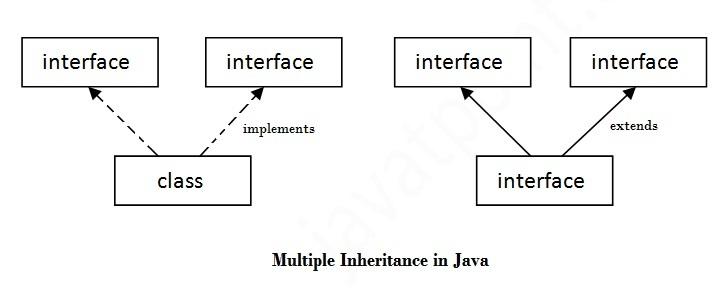
}

**Output:**

area of rect=100.0

area of circle=314.0

## **Multiple inheritance in Java by interface:**



**Example 4: Interface can extend many interfaces separated by commas.**

interface Myinterface1

{

public void method1();

}

interface Myinterface2

{

public void method2();

}

interface Myinterface3 extends Myinterface1, Myinterface2

{

public void method3();

}

class interfacedemo3 implements Myinterface3

{

public void method1()

{

System.out.println("implementation of method1");

}

public void method2()

{

System.out.println("implementation of method2");

}

public void method3()

{

System.out.println("implementation of method3");

}

public static void main(String arg[])

{

Interfacedemo3 obj = new interfacedemo3();

obj. method1();

obj. method2();

obj. method3();

}

}

**Output:**

implementation of method1

implementation of method2

implementation of method3

**Example 5: A Class can implement two interfaces separated by commas.**

interface Myinterface1

{

public void method1();

}

interface Myinterface2

{

public void method2();

}

class interfacedemonotp implements Myinterface1,Myinterface2

{

public void method1()

{

System.out.println("implementation of method1");

}

public void method2()

{

System.out.println("implementation of method2");

}

public static void main(String arg[])

{

interfacedemonotp obj = new interfacedemonotp();

obj. method1();

obj. method2();

}

}

**Output:**

implementation of method1

implementation of method2

**NOTE: If declared in public need to be saved in different files**

public interface Myinterface1 //save with the file name Myinterface1.java

{

public void method1();

}

public interface Myinterface2 //save with the file name Myinterface2.java

{

public void method2();

}

//save with the file name Myinterface3.java

public interface Myinterface3 extends Myinterface1, Myinterface2

{

public void method3();

}

//save with the file name interfacedemo3.java

public class interfacedemo3 implements Myinterface3

{

public void method1()

{

System.out.println("implementation of method1");

}

public void method2()

{

System.out.println("implementation of method2");

}

public void method3()

{

System.out.println("implementation of method3");

}

public static void main(String arg[])

{

Interfacedemo3 obj = new interfacedemo3();

obj. method1();

obj. method2();

obj. method3();

}}

# Java Nested Interface

An interface, i.e., declared within another interface or class, is known as a nested interface. The nested interfaces are used to group related interfaces so that they can be easy to maintain. The nested interface must be referred to by the outer interface or class. It can't be accessed directly.

### Points to remember for nested interfaces

There are given some points that should be remembered by the java programmer.

* The nested interface must be public if it is declared inside the interface, but it can have any access modifier if declared within the class.
* Nested interfaces are declared static

### Syntax of nested interface which is declared within the interface

1. **interface** interface\_name{
2. ...
3. **interface** nested\_interface\_name{
4. ...
5. }
6. }
7. **interface** Showable{
8. **void** show();
9. **interface** Message{
10. **void** msg();
11. }
12. }
13. **class** TestNestedInterface1 **implements** Showable.Message{
14. **public** **void** msg(){System.out.println("Hello nested interface");}
16. **public** **static** **void** main(String args[]){
17. Showable.Message message=**new** TestNestedInterface1();//upcasting here
18. message.msg();
19. }
20. }

****Output:****

hello nested interface

Summary:

**Abstract class:**

1. Abstract classes are there to impose prototyping / rules on sub classes

2. they use abstract methods (empty methods or no definition) to do the above

3. if there is an empty method u must declare that class as abstract

4. abstract class may or may not have more than one abstract method

5. it can also have normal methods

6. we can not create objects to abstract classes since it contains empty or abstract methods

7. the only way to use them is thorough inheritance

8. but when we inherit such abstract classes it is our responsibility to override its all abstract methods

**Note: In java we can not extend multiple classes but we can implement multiple interfaces.**

1. interface is similar to abstract class

2. by default the var are public static final and methods are public abstract

3. in java we can inherit multiple interfaces

4. interface can extend another interface

5. till java 1.7 interfaces can have only abstract methods

but in 1.8 version normal methods introduced in interfaces using keywords default or static

public default/static void read()

{

}

from 1.9 version private methods in interfaces

**Example 01:**

interface Electricity

{

String cname = "East. Power Dist.";

public void calBill(int x);

}

class HouseHold implements Electricity

{

int unitcharge;

HouseHold()

{

unitcharge = 5;

}

public void calBill(int x) // this will override super class method

{

float bill = unitcharge\*x;

System.out.println("Bill = "+bill);

}

}

class Commercial implements Electricity

{

int unitcharge;

Commercial()

{

unitcharge = 10;

}

public void calBill(int x) // this will override super class method

{

float bill = unitcharge\*x+500;

System.out.println("Bill = "+bill);

}

}

public class TestElecBill

{

public static void main(String arg[])

{

Electricity ref; // reference var

ref = new HouseHold(); // object

ref.calBill(340);

}

}

**Example 02:**

interface AI

{

public default void f1()

{

System.out.println("AI Enabled");

}

}

interface MediaPlayer extends AI

{

public default void f2()

{

System.out.println("Media Player");

}

}

class MyPlayer implements MediaPlayer

{

}

public class TestInterface2

{

public static void main(String arg[])

{

MyPlayer obj = new MyPlayer();

obj.f1();

obj.f2();

}

}

**Example 03:**

interface AI

{

public default void f1()

{

System.out.println("AI Enabled");

}

}

interface MediaPlayer

{

public default void f2()

{

System.out.println("Media Player");

}

}

class MyPlayer implements AI,MediaPlayer

{

}

public class TestInterface2

{

public static void main(String arg[])

{

MyPlayer obj = new MyPlayer();

obj.f1();

obj.f2();

}

}

**Example 04:**

interface Account

{

public static final String bname = "SBI";

public static final String ifsc = "SBI0012";

public void withdraw(float amt);

public void deposit(float amt);

public void balChk();

public static void aeps(int x)

{

ssl();

if(x==1)

{

System.out.println("AEPS enabled");

}

else

{

System.out.println("AEPS Disabled");

}

}

private static void ssl()

{

System.out.println("Transactions secured");

}

}

class Saving implements Account

{

public void withdraw(float amt)

{

System.out.println("Savings withdraw");

}

public void deposit(float amt)

{

System.out.println("Savings Deposit");

}

public void balChk()

{

System.out.println("Savings Bal Chk");

}

}

class Current implements Account

{

public void withdraw(float amt)

{

System.out.println("Current withdraw");

}

public void deposit(float amt)

{

System.out.println("Currrent Deposit");

}

public void balChk()

{

System.out.println("Current Bal Chk");

}

}

public class TestInterface

{

public static void main(String arg[])

{

Account ref;

ref = new Current();

System.out.println(Account.bname+" IFSC : "+Account.ifsc);

ref.withdraw(1);

ref.deposit(1);

ref.balChk();

Account.aeps(1);

}

}