CHAPTER

7

INHERITANCE

Chapter Includes :

- INHERITANCE
- SINGLE INHERITANCE
- MULTILEVEL INHERITANCE
- MULTIPLE INHERITANCE
- HIERARCHICAL INHERITANCE
- USING SUPER
- CONSTRUCTOR -ORDER OF EXECUTION IN INHERITANCE
- OVERRIDING METHODS
- FINAL VARIABLES AND METHODS
- FINAL CLASSES
- ABSTRACT METHODS AND CLASSES
- CONTAINERSHIP
- VISIBILITY CONTROL

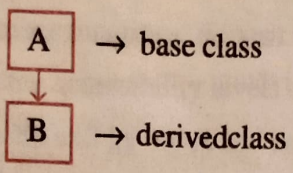
INHERITANCE

Inheritance is the property of a class to inherit property of another class. The class whose properties are inherited is called as base class or super class or parent class. The class that inherits properties is known as derived class or subclass or child class. It supports the concept of **reusability**. It means a new class can inherit the features of existing class. You can reuse existing class instead of defining same features again and again.

Various types of Inheritance are :

1. Single
2. Multilevel
3. Multiple
4. Hierarchical

SINGLE INHERITANCE



In single inheritance, there is only one base class and one derived class. The derived class inherits the features of base class.

General form :

class classname extends base classname

Program that demonstrate use of single inheritance

```
class Base
    {
        int a=1,b=2;
        void display()
        {
            System.out.println("value of a = "+a);
            System.out.println("value of b = "+b);
        }
    }

class Derived extends Base
    {
        void show()
        {
            System.out.println("This is derived class");
        }
    }
```

```
class Result
  {
     public static void main(String args[])
     {
        Derived obj1=new Derived();
        obj1.show();
        obj1.display();        // can access members of base
     }
  }
```

OUTPUT

This is derived class
value of a = 1
value of b = 2

Here class named as Derived is inheriting all features of Base class (super class). So it can access all members of super class. That's why we are accessing display() method with the help of Derived class object. Base is the name of base class and Derived is the name of derived class. We are extending features of Base class. So we can access methods and variables of Base class with the help of objects of Derived class like here obj1 (object of Derived class) is accessing display().

As you know all members of class possess access specifiers. So there is a limit up to which they will be visible to outside world.

If you define a member as **private** then it can be accessed by members of its class only. It is not accessible by members of other class, not even derived class can access private members of base class.

If you define a member as **public** then it can be accessed by any members outside that class also. When a derived class inherits the base class, it can access all public members.

Let's see an example.

program that demonstrate member access in inheritance

```
class A
  {
     public int i=7,j=5;
     private int b;
     void display()
        {
           System.out.println("value of b = 3");
        }
  }
class B extends A
  {
     void show()
```

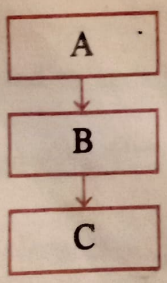
```
        {
            b=2;    // error
                        System.out.println("value of i=" + i);
            System.out.println("value of j="+j);
            System.out.println("value of b="+b);
        }
    }

class Result
    {
        public static void main(String args[])
        {
            B obj1 = new B();
            obj1.display();
            obj1.show();
        }
    }
```

This program will not be compiled. This will give you an error. In this program i and j are public members so derived class can access them. b is a private members so it can be accessed by members of A class only, B class can't access it. So this program will give you an error as we are accessing b inside B class.

Protected : It means members declared as protected are accessible to all classes and sub classes in the same package and to the sub class in the other packages. They are not accessible to the non-sub classes in other packages. So derived class can access them.

friendly (default) : When a member is not declared private, public and protected then it is said to be friendly. In Java all members are friendly by default. When a member is declared as friendly then it can be accessed by other classes but in the same package. When a derived class inherits a base class, it can access all of the members that are friendly provided derived class should exist in same package.

MULTILEVEL INHERITANCE



In Multilevel Inheritance, class A acts as a base class for B and class B in turns acts as a base class for C. So we can say that "A class act as a base class and also act as a sub class" is known as multilevel Inheritance.

General form :

```
class A          // base class for B
  {
      body of class
  }
class B extends A          // base class for C
  {
      body of class
  }
class C extends B
  {
  }
```

program that demonstrate multilevel inheritance

```
class A
  {
    int i=2,j=4;
    void display()
      {
        System.out.println("This is the super class");
      }
  }
class B extends A
  {
    int sum;
    void sum()
      {
        sum = i + j;
        System.out.println("sum = "+sum);
      }
  }
class C extends B
  {
    void show()
      {
      System.out.println("This is an example of Multilevel Inheritance");
      }
  }
class Result
  {
    public static void main(String args[])
```

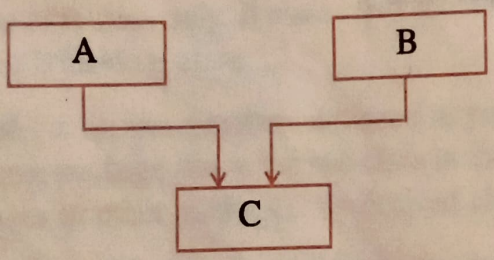
```
        {
            C obj = new C();
            obj.show();
            obj.sum();
            B obj1 = new B();
              obj1.display();
        }
    }
```

OUTPUT

This is an example of Multilevel Inheritance
sum =6
This is the super class

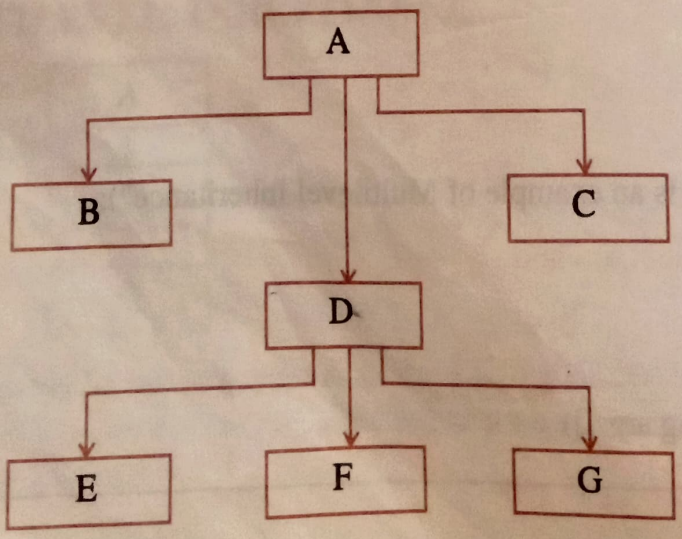
In this program, we have defined four classes A, B, C and Result. Result is the main class. A, B, C classes are demonstrating concept of Multilevel Inheritance class A is base class for B, so B class can derive all features of A class as per visibility of members. Then B class will acts as a base class for C. Now C class can inherit features of class B and A. You can access members of B and A with the help of objects of C and in turn B can access members of A.

MULTIPLE INHERITANCE



In Multiple Inheritance, there are two or more base classes and one derived class. **Java does not support multiple inheritance** but one can implement it by use of **Interface** (that we will discuss in next chapter).

HIERARCHICAL INHERITANCE



A Hierarchy is made in this type of inheritance. There are many problems that are solved on the basis of hierarchical design. Here is single base class but several derived class.

USING SUPER

As you have seen, there are situations when a derived class can't access members of base class. So in order to access them you can use **super** keyword. With the use of super, a subclass can refer to superclass. Super provides us with two uses :

1. It can call super class constructor.
2. It can help you accessing super class members.

1. Calling super class constructor

You can call superclass constructor inside the sub class.

General form

super(Parameter-list);

You just need to write super and inside brackets write down the parameters it contains if base class constructor is parameterized constructor, otherwise leaves the brackets empty. You must write this statement as first statement inside the subclass constructor.

Program that calls super class constructor by using super

```
class A
  {
     int i, j;
       A( )
         {
              i=0;
              j=0;
           }
       A(int a,int b)
         {
           i=a;
           j=b;
         }
    void display()
              {
              int p = i * j;
       System.out.println("Product of i and j =" +p);
        }
  }
```

```
class B extends A
    {
        int d;
        B( )
            {
                super( );        // calling default constructor of base class
                d = 1;
            }
        B(int f,int g)
            {
                super(f,g);   // calling parameterized constructor of base class
                d = 2;
            }
            void show()
                {
                System.out.println("d = "+d);
                }
    }
    class Result
        {
            public static void main(String args[])
            {
                B ob1 = new B( );
                B ob2 = new B(3,4);
                ob1.display( );
                ob1.show( );
                ob2.display( );
                ob2.show( );
            }
}
```

OUTPUT

Product of i and j = 0
d = 1
Product of i and j = 12
d = 2

Here B calls super( ) that contains no parameters. This causes the default constructor i.e. A( ) to be called. It initializes the variable i and j with 0 value. then B calls super( ) with two parameters which causes call to parameterized constructor A(int a, int b) and initializes variables i and j. Here constructors are overloaded so compiler will call constructors according to parameters. super( ) can be called by using any form defined in base class.

2. Use of super to access super class members.

Sometimes we encounter a situation where we use same variable and method name in base class as well as in derived class. Then the members of base class loose their functionality, and all variables and methods work in accordance with derived class.

For example :

If we declare a variable i in base class, and again we declare a variable i in derived class. Then i will show you contents that derived class will provide you.

So in order to avoid this situation and to access the super class members, we write super before variables and methods of base class.

General form :

```
super.member variable;
super.methodname(parameterslist);
```

Dot operator is used to relate super with members of base class.

Program that demonstrate use of super to access super class members.

```
class A
  {
     int c=4;

  }
class B extends A
  {
     int c=7,d;

   void display( )
     {
        System.out.println("value of c in super class= "+super.c);
        System.out.println("value of c in subclass= "+c);
     }
   }

class Result
  {
     public static void main(String args[])
        {
           B obj = new B();
           obj.display();
        }
  }
```

OUTPUT

value of c in superclass= 4
value of c in subclass = 7

Here we are accessing variable c defined in super class by using super.c and c defined in subclass also. super.c will show you value defined in base class and c will show you value defined in derived class.

CONSTRUCTOR - ORDER OF EXECUTION IN INHERITANCE

In inheritance, constructors are executed in the same order as they are derived. Order of execution is from super class constructor to subclass constructor. No matter whether you are using super or not, default or parameterized constructor of each class is executed.

Program showing order of execution of constructor

```
class R
    {
        R ( )
        {
            System.out.println("Inside R constructor");
        }
    }
class S extends R
    {
        S( )
        {
            System.out.println("Inside S constructors");
        }
    }
class T extends S
    {
        T( )
        {
            System.out.println("Inside T constructor");
        }
    }

class Result
    {
        public static void main(String args[])
            {
                T obj = new T();
            }
    }
```

OUTPUT

```
Inside R constructor
Inside S constructor
Inside T constructor
```

OVERRIDING METHODS

When we inherit a class, then we inherit all its members. Sometimes it happens that base class and derived class contain method with same name and parameters. So when that method is invoked with the help of derived class objects, then it will always display methods defined in derived class not in base class. Actually derived class method overrides the method of base class. This concept is known as method overriding.

Program : Showing Method Overriding

```
class A
  {
      void display()
      {
          System.out.println("Base class");
      }
  }
class B extends A
  {

  void display()
      {
          System.out.println("Method overriding");
      }
  }
class Test
  {
      public static void main(String args[])
      {
          B ob = new B();
          ob.display();
      }
  }
```

OUTPUT

```
Method overriding
```

Here compiler will invoke display( ) method of derived class.

So in this type of situation, if we want to call method of super class also then we can call it by the use of super keyword.

Program that calls method of base class by using super

```
class Alpha
{
    private int x;
    void set(int a)
    {
        x=a;
    }
    void print()
    {
        System.out.println("X = "+x);
    }
}
class Beta extends Alpha
{
    int y;
    void set(int b,int c)
    {
        y=b;
        super.set(c);
    }
    void print()
    {
        System.out.println("Y = "+y);
        super.print();
    }
}
class Test
{
    public static void main(String args[])
    {
        Beta b=new Beta();
        b.set(10,20);
        b.print();
    }
}
```

OUTPUT

Y= 10

X= 20

Here with the use of super we can call method of base class inside derived class method. Hence though derived class is overriding method of base class still we can access methods of base class by using super.

FINAL VARIABLE AND METHODS

In chapter-2, you study about symbolic constants. They are constant-names that precede with final keyword. When a constant-name procedes with final keyword then you can't alter its value. Next use of final is studied in this section. When you inherit a class, then there are chances of overriding of variables and methods. So in order to prevent overriding of methods and variables, we declare them as FINAL. All those methods and variables that precede with final keyword cannot be redefined in derived class.

Program showing use of final

```
class A
    {
        final int a=4, b=9;

        final void display()
            {
                System.out.println("value of a = " + a);
                System.out.println("value of b = " + b);
            }
    }
class B extends A
    {
        a=5;    // error, can't alter
        void display()  // error, can't redefine
            {
            System.out.println("Wrong");
            }
    }
class Result
    {
    public static void main(String args[])
        {
            B obj = new B( );
        }
    }
```

This program will give you an error, as we are altering value of 'a' which is a final variable and redefining a final method.

Let's alter this program :

Program Showing use of final

```
class A
    {
        final int a=4, b=9;
        final void display()
            {
                System.out.println("value of a = "+a);
                System.out.println("value of b = "+b);
            }
    }
class B extends A
    {
        int sum=0;
        void sum()
            {
            sum=a+b;
            System.out.println("Sum ="+sum);
            }
    }
class Test1
    {
        public static void main(String args[])
        {
            B obj = new B();
            obj.display();
            obj.sum();
        }
    }
```

OUTPUT

value of a = 4
value of b = 9
sum = 13

Now compiler will invoke method of base class.

FINAL CLASS

Sometimes we want that our class should not be inherited. i.e. No one can inherit our class. To do so we use **final** keyword. When we precedes final with class name, then class will not be inherited. If we will try to inherit that class then it will give compile-time error. All methods of final class should be static so that one can access methods of final class by using class name only.

General form :

```
Final class name
    {
        Body of class
    }
```

Program Showing use of final

```
final class A
  {
    public static void show( )
    {
        System.out.println("Final class");
    }
  }
class B extends A  // error
  {
    void display()
    {
        System.out.println("Can't inherit final class");
    }
  }
class Test1
    {
      public static void main(String args[])
        {
            B ob = new B();
            ob.show();
            ob.display();
        }
    }
```

This program will give you compile time error. To run this program, do not inherit the class A.

So final can be used in 2 ways :

1. To prevents method overriding.
2. To prevents inheritance.

ABSTRACT METHOD AND CLASSES

When we make final variables and methods, then we can't redefine them in subclass. There are situations when we want to redefine a method. We do it by making a method abstract. When a method is defined as abstract then it is necessary for derived class to redefine it.

Abstracts methods are the methods that must be redefined in derived class, thus making overriding compulsory. When a class contains one or more abstract methods, it should be declared as abstract.

General form for declaring abstract method :

abstract type method name (parameter list)

It is the responsibility of subclass to implement all abstract methods inside it.

We can't create objects of abstract class. As abstract class is not fully defined so these objects will be useless. We can't declare abstract constructors and abstract static methods. When a class inherit an abstract class then it must define all abstract methods of abstract class or it should declare itself as abstract.

Program Showing abstract methods and classes

```
abstract class A
    {
        abstract void show();

    }
class B extends A
    {
        void show( )
        {
            System.out.println("Example of abstract class");
        }
    }
class Test1
        {
        public static void main(String args[])
        {
            B ob = new B( );
```

```
        ob.show( );
    }
}
```

OUTPUT

Example of abstract class

Summary of three classes in accordance with inheritance :

Classes	Use	Extends
General Class	yes	yes
Final class	yes	no
Abstract Class	no	yes

CONTAINERSHIP

We can create object of one class in another class without inheritance. This concept is called containership. It means that we do not inherit a class but use it by making its objects. Two classes do not possess any relationship but one class creates object of another class explicitly.

Program showing concept of containership

```
class A
  {
        void show( )
          {
          System.out.println("Inside A class");
          }
  }
class B
  {
      A ob=new A( );
       public void show1()
     {
                ob.show();
       System.out.println("Inside B class");
     }
  }
class Test1
```

```
    {
    public static void main(String args[])
    {
        B obj = new B( );
        obj.show1( );
    }
}
```

OUTPUT
Inside A class
Inside B class

Here A and B do not have any relationship. We are creating objects of A explicitly.

A is type of B .....it means inheritance.
A contains B .....it means containership.

VISIBILITY CONTROL

As you already know there are four access specifiers in Java. Access Specifiers are added in order to determine the level of protection. Let's see visibility of class members in different location according to access specifier used.

There are five access locations that exist in Java. They are:

- Same class
- Subclass in same package.
- Other classes in same package.
- Subclasses in other package.
- Non-subclass in other package.

Let's see visibility of various class members in these locations.

Same class

Private member —— You can access.

Public member —— You can access.

Protected member —— You can access.

Friendly (default) —— You can access.

Subclass in same package

Private member —— You can't access.

Public member ——— You can access.

Protected member —— You can access.

Friendly (default) —— You can access.

Other classes in same package

Private member —— You can't access.

Public member——— You can access.

Protected member——You can access.

Friendly (default)——You can access.

Subclasses in other package

Private member —— You can't access.

Public member ——— You can access.

Protected member —— You can access.

Friendly (default) —— You can't access.

Non-subclass in other package

Private member —— You can't access.

Public member ——— You can access.

Protected member ——You can't access.

Friendly (default) ——You can't access.

So now you know which members are visible in which location.

Difference between method overloading and method overriding

- Method overloading means to create multiple methods with same name but different parameters in order to perform different tasks while during inheritance if there is a method with same name in base as well as in derived class then derived class method override the base class method, this is known as method overriding.
- All methods involved in method overloading are members of same class so relationship exists between members of same class. In method overriding methods belongs to subclass and superclass. Hence relationship exists between members of base and derived class.