# CHAPTER - 4

# INHERITANCE

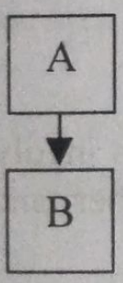
## 4. Inheritance:

The mechanism of deriving a new class from an old one is called inheritance. A class that is inherited (old class) is called a super class or base class or parent class. The class that does the inheriting (new class) is called a subclass or derived class or child class. Inheritance allows subclasses to inherit all the variables and methods of their parent classes.

### 4.1 Types of Inheritances

Inheritance is classified into different forms.

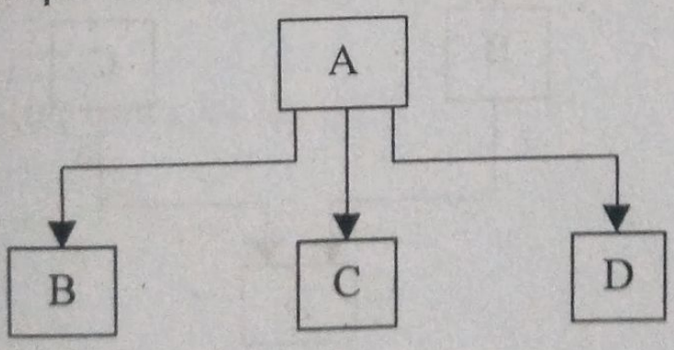
a) **Single Inheritance:** Derivation a subclass from only one super class is called Single Inheritance.



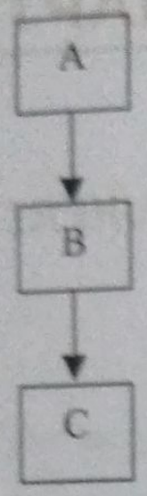

Super class (or) Base class (or) Parent class

Subclass (or) Derived class (or) Child class

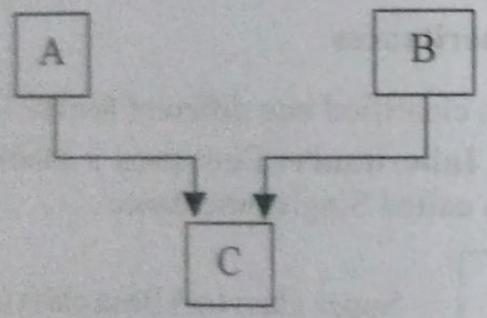
b) **Hierarchical Inheritance:** Derivation of several classes from a single super class is called Hierarchical Inheritance.



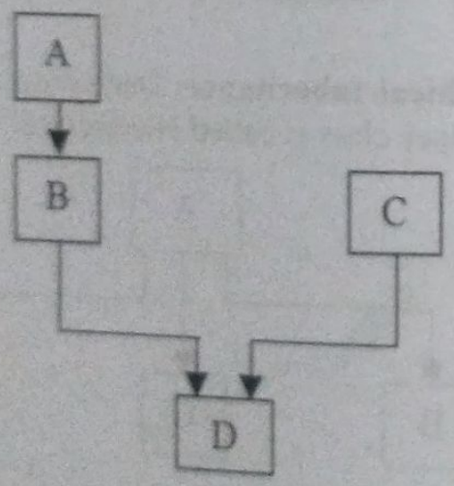
c) **Multilevel Inheritance:** Derivation of a class from another derived class s called Multilevel Inheritance.




d) **Multiple Inheritance:** Derivation of one class from two or more super classes is called Multiple Inheritance. But Java does not support Multiple Inheritance directly. It can be implemented by using Interface concept.




e) **Hybrid Inheritance:** Derivation of a class involving more than one form of Inheritance is called Hybrid Inheritance.






## 4.2 Defining a Subclass:

A subclass is defined as

Syntax:
```
class subclass-name extends superclass-name
{
        variable declaration;
        method declaration;
}
```

The keyword extends signifies that the properties of the super class name are extended to the subclass name. The subclass will now contain its own variables and methods as well as those of the super class. But it is not vice-versa.

**Example 1:** Program for Single Inheritance

```
class A
{
    int len,bre;
    void getdata(int x,int y)
    {
    len=x;
    bre=y;
    }
    void printdata()
    {
    System.out.println("length="+len);
    System.out.println("Breadth="+bre);
    }
}
class B extends A
{
    int wid;
    void putdata(int x,int y,int z)
    {
    len=x;
    bre=y;
    wid=z;
    }
```

```
        void printdata1()
        {
        printdata();
        System.out.println("Width="+wid);
        }
        void calarea()
        {
        int area=len*bre*wid;
        System.out.println("Area="+area);
        }
}
class si
{
        public static void main(String[] args)
        {
        B obj=new B();
        obj.putdata(10,20,30);
        obj.printdata1();
        obj.calarea();
        }
}
```

Example 2: Program for Hierarchical Inheritance

```
class A
{
        int len,bre;
        void getdata(int x,int y)
        {
        len=x;
        bre=y;
        }
        void printdata()
        {
```

```
        System.out.println("length="+len);
        System.out.println("Breadth="+bre);
        }
}
class B extends A
{
        int wid;
        void getdata1(int x,int y)
        {
        getdata(x,y);
        }
        void calarea()
        {
        printdata();
        int area=len*bre;
        System.out.println("Area="+area);
        }
}
class C extends A
{
        int wid;
        void putdata(int x,int y,int z)
        {
        len=x;
        bre=y;
        wid=z;
        }
        void printdata1()
        {
        printdata();
        System.out.println("Width="+wid);
        }
```

```
        void calarea1()
        {
        int area=len*bre*wid;
        System.out.println("Area="+area);
        }
}
class hi
{
        public static void main(String[] args)
        {
        B obj=new B();
        obj.getdata1(10,20);
        obj.calarea();
        C obj1=new C();
        obj1.putdata(1,2,3);
        obj1.printdata1();
        obj1.calarea1();
        }
}
```

**Example:** //Example Program for Hierarchical Inheritance

```
import java.text.*;
class Employee
{
int Empno;
double Basic,DA,HRA;
String Empname;
void readdata(int x,String y)
{
Empno=x;
Empname=y;
}
```



```
void readsalary(double x,double y,double z)
{
Basic=x;
DA=y;
HRA=z;
}
void printdata()
{
System.out.println("Employee Number:"+Empno);
System.out.println("Employee Name :"+Empname);
}
void printsalary()
{
DecimalFormat df=new DecimalFormat("0.00");
System.out.println("Employee Basic :"+df.format(Basic));
System.out.println("Employee DA    :"+df.format(DA));
System.out.println("Employee HRA   :"+df.format(HRA));
}
}
class TSalary extends Employee
{
void showdata()
{
readdata(111,"Vishnu");
printdata();
readsalary(2589.00,567,1090);
printsalary();
}
void calsalary()
{
DecimalFormat df=new DecimalFormat("0.00");
double netsal;
```

```
netsal=Basic+DA+HRA;
System.out.println("Employee Netsalary="+df.format(netsal));
}
}
class PSalary extends Employee
{
double Itax;
void readtax(double x)
{
Itax=x;
}
void showdata()
{
readdata(1000,"Kamal");
printdata();
readsalary(12589.00,1567,2090);
printsalary();
}
void calsalary()
{
DecimalFormat df=new DecimalFormat("0.00");
double Gsal,netsal;
Gsal=Basic+DA+HRA;
netsal=Gsal-Itax;
System.out.println("Income Tax Amount="+df.format(Itax));
System.out.println("Employee Netsalary="+df.format(netsal));
}
}
class Harch
{
        public static void main(String[] args)
        {
        TSalary obj=new TSalary();
```



```
        obj.showdata();
        obj.calsalary();
        PSalary obj1=new PSalary();
        obj1.readtax(800);
        obj1.showdata();
        obj1.calsalary();
        }
}
```

Example 3: Program for Multilevel Inheritance

```
import javax.swing.JOptionPane;
class student
{
        int rno;
        void readdata1()
        {
        rno=Integer.parseInt(JOptionPane.showInputDialog("Enter
roll number"));
        }
        void printdata1()
        {
        System.out.println("Roll Number="+rno);
        }
}
class exam extends student
{
        int m1,m2,m3;
        void readdata2()
        {
        readdata1();
        m1=Integer.parseInt(JOptionPane.showInputDialog("Enter
sub1 marks"));
        m2=Integer.parseInt(JOptionPane.showInputDialog("Enter
```

```
sub2 marks"));
        m3=Integer.parseInt(JOptionPane.showInputDialog("Enter
sub3 marks"));
        }
        void printdata2()
        {
        printdata1();
        System.out.println("Sub 1 marks="+m1);
        System.out.println("Sub 2 marks="+m2);
        System.out.println("Sub 3 marks="+m3);
        }
}
class results extends exam
{
        int avg;
        void printdata3()
        {
        printdata2();
        avg=(m1+m2+m3)/3;
        if(avg<35)
        System.out.println("FAIL");
        else if(avg>=35 && avg<50)
        System.out.println("THIRD CLASS");
        else if(avg>=50 && avg<60)
        System.out.println("SECOND CLASS");
        else if(avg>=60 && avg<75)
        System.out.println("FIRST CLASS");
        else if(avg>=75)
        System.out.println("DISTINCTION");
        }
}
```



```
class mi
{
        public static void main(String[] args)
        {
                results obj=new results();
                obj.readdata2();
                obj.printdata3();
        }
}
```

## 4.3 Member Access Rules:

1. By means of sub class object it is possible to access all the instance variables and instance methods of the super class.
2. By means of sub class object it cannot be possible to access the instance variable of super class, if it is declared as private. Since, the data is protected in that class.

**Example:**

```
import javax.swing.JOptionPane;
class student
{
        private int rno;
        void readdata1()
        {
        rno=Integer.parseInt(JOptionPane.showInputDialog("Enter roll number"));
        }
}
class exam extends student
{
        void printdata1()
        {
        readdata1();
        System.out.println("Roll Number="+rno);
          // produces compilation error
        }
```

```
class mi2
{
    public static void main(String[] args)
    {
        exam obj=new exam();
        obj.printdata1();
    }
}
```

3. Referring to super class object with a subclass reference produces syntax error. In such case we need type casting.

Example:

```
import javax.swing.JOptionPane;
class student
{
    int rno;
    void readdata1()
    {
    rno=Integer.parseInt(JOptionPane.showInputDialog("Enter
roll number"));
    }
}
class exam extends student
{
    void printdata1()
    {
    readdata1();
    System.out.println("Roll Number="+rno);
    }
}
```



```
class mi2
{
    public static void main(String[] args)
    {
        student s1=new exam();
        exam e1=(exam)s1;
        e1.printdata1();
    }
}
```

## 4.4 Super keyword:

A subclass constructor is used to construct the instance variable of both the subclass and the super class. The subclass constructor uses the keyword super to invoke the constructor method of the super class. The keyword super is used subject to the following conditions.

- super may be used only within the subclass constructor method
- The call to super class constructor must appear as the first statement with in the subclass constructor
- If a parameterized constructor is used, then the signature is matched with the parameters passed to the super method as super(parameter-list);

Example 1:

```
class student
{
    public student()
    {
    System.out.println("super");
    }
}
class exam extends student
{
    public exam()
    {
    System.out.println("sub");
    }
}
```

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

O/P: super
sub

Example 2:

```
class student
{
        public student()
        {
        System.out.println("super");
        }
}
class exam extends student
{
        public exam()
        {
        super( );               // explicit call of super( )
        System.out.println("sub");
        }
}
class s
{
        public static void main(String[] args)
        {
            exam obj=new exam();
        }
}
```

O/P: super
sub



Example 3:

```
class student
{
        int len,bre;
        public student(int a,int b)
        {
        len=a;
        bre=b;
        }
        int area()
        {
        return len*bre;
        }
}
class exam extends student
{
        int ht;
        public exam(int x,int y,int z)
        {
        super(x,y);
        ht=z;
        }
        int volume()
        {
        return len*bre*ht;
        }
}
class s
{
        public static void main(String[] args)
        {
                exam obj=new exam(2,3,4);
                System.out.println("area="+obj.area());
                System.out.println("volume="+obj.volume());
        }
}
```

Another important use of super is applicable to situations in which member names of a subclass hide members by the same name in the super class. It is similar to the 'this' keyword except that it always refers to the super class of the subclass in which it is used. The general form is as

super.member

where member can be either a method or an instance variable

Example:

```
class A
{
    int i=7;
}
class B extends A
{
    int i=6;
    void show()
    {
    System.out.println("super i="+super.i);
    System.out.println("sub   i="+i);
    }
}
class s1
{
    public static void main(String[] args)
    {
        B obj=new B();
        obj.show();
    }
}
```

## 4.5 Method Overriding

A method in a subclass has the same name, type of the variables and order of the variables as a method in its super class, then the method in the subclass is said to be override the method in the super class. The process is called Method Overriding. In such process the method defined by the super class will be hidden.



When the method is called, the method defined in the super class has invoked and executed instead of the method in the super class. The super reference followed by the dot (.) operator may be used to access the original super class version of that method from the subclass.

Note: 1. Method Overriding occurs only when the name and type signature of the two methods are identical.

2. If method name is identical and the type signature is different, then the process is said to be Method Overloading.

Example 1:

```
class A
{
    int i,j;
    public A(int a,int b)
    {
    i=a;
    j=b;
    }
    void show()
    {
    System.out.println("i="+i+"\n"+"j="+j);
    }
}
class B extends A
{
    int k;
    B(int a,int b,int c)
    {
    super(a,b);
    k=c;
    }
    void show()
    {
    System.out.println("k="+k);
    }
}
```

```
class override
{
        public static void main(String[] args)
        {
                B obj=new B(1,2,3);
                obj.show();
        }
}
```

O/P: k=3

Example 2:

```
class A
{
        int i,j;
        public A(int a,int b)
        {
        i=a;
        j=b;
        }
        void show()
        {
        System.out.println("i="+i+"\n"+"j="+j);
        }
}
class B extends A
{
        int k;
        B(int a,int b,int c)
        {
        super(a,b);
        k=c;
        }
```



```
        void show()
        {
        super.show();
        System.out.println("k="+k);
        }
}
class override
{
        public static void main(String[] args)
        {
                B obj=new B(1,2,3);
                obj.show();
        }
}
```

O/P: i=1
j=2
k=3

Resolve the method call at compile time is known as "compile time" or "early binding" or "static binding". Resolve the method at run time is known as "run time" or "late binding" or "dynamic binding".

Method Overriding forms on the basis of Java concept Dynamic Method Dispatch. It is the mechanism by which a call to an overridden function is resolved at run time, rather than compile time. And this method is very useful to show for Java implements run-time polymorphism.

"Super class reference variable can refer to a subclass object" is the principle to resolve calls to overridden methods at run time. If a super class contains a method that is overridden by a subclass, then when different types of objects are referred to through a super class reference variable, different version of the methods are executed and the determination is made at run time.

**Lab Program 21:** Example program for Java supports Run time Polymorphism.

```
class figure
{
    double dim1,dim2;
    figure(double x,double y)
    {
    dim1=x;
    dim2=y;
    }
    double area()
    {
    System.out.println("Area undefined");
    return 0;
    }
}
class rectangle extends figure
{
    rectangle(double a,double b)
    {
    super(a,b);
    }
    double area()
    {
    System.out.println("Rectangle Area");
    return dim1*dim2;
    }
}
class triangle extends figure
{
    triangle(double x,double y)
    {
    super(x,y);
    }
    double area()
    {
    System.out.println("Triangle Area");
    return dim1*dim2/2;
    }
}
class run
{
    public static void main(String[] args)
    {
    figure obj=new figure(10,10);
    rectangle obj1=new rectangle(9,5);
    triangle obj2=new triangle(10,8);

    figure a;           //a is a reference of type figure(super class)
    a=obj1;             //a refers to object of rectangle(sub class)
    System.out.println("Area="+a.area());

    a=obj;              //a refers to object of figure
    System.out.println("Area="+a.area());

    a=obj2;             //a refers to object of triangle
    System.out.println("Area="+a.area());
    }
}
```

O/P: Rectangle Area
Area=45.0
Area undefined
Area=0.0
Triangle Area
Area=40.0



## 4.6 Abstract Class:

Classes from which objects cannot be instantiated with new operator are called Abstract Classes. Each abstract class contains one or more

abstract methods. In a class if there exist any method with no method body is known as abstract method. An abstract method is declared as

Syntax: abstract type method-name (parameter-list);

"abstract" is a keyword used for declaring abstract methods. To declare a class as abstract, use the abstract keyword in front of the class keyword at the beginning of the class declaration. No objects are created to an abstract class. For the abstract methods, the implementation code will be defined in its subclass.

Example:

```
abstract class figure
{
    double dim1,dim2;
    figure(double x,double y)
    {
    dim1=x;
    dim2=y;
    }
    abstract double area();
}
class rectangle extends figure
{
    rectangle(double a,double b)
    {
    super(a,b);
    }
    double area()
    {
    System.out.println("Rectangle Area");
    return dim1*dim2;
    }
}
class triangle extends figure
{
    triangle(double x,double y)
    {
```



```
    super(x,y);
    }
    double area()
    {
    System.out.println("Triangle Area");
    return dim1*dim2/2;
    }
}
class abs
{
    public static void main(String[] args)
    {
    //figure obj=new figure(10,10); //error
    rectangle obj1=new rectangle(9,5);
    System.out.println("Area="+obj1.area());
    triangle obj2=new triangle(10,8);
    figure a;
    a=obj2;
    System.out.println("Area="+a.area());
    }
}
```

Note: 1. We cannot declare any abstract constructors.

2. Concrete (general) methods are also being inside any abstract class.
   - i) Abstract classes are used for general purposes.
   - ii) An abstract class must be subclass and override it methods.
   - iii) Super class has only method name and signature end with semicolon.
   - iv) Abstract classes cannot be used to instantiate objects, but they can used to create object reference due to Java supports Run-time Polymorphism.

## 4.7 Final keyword

A final is a keyword used for three purposes. They are

a) **final as constant:** A variable can be declared as constant with the keyword final. It must be initialized while declaring. One we initialized the variable with final keyword it cannot be modified in the program. This is similar to the const in C/C++.

Example:
```
final int x = 10;
x = 45 //error
```

**b) final to prevent overriding:** A method that is declared as final cannot be overridden in a subclass method. It the methods that are declared as private are implicitly final.

Example:
```
class A
{
    final void show( )
    {
    System.out.println("Hello");
    }
}
class B extends A
{
    void show( )
    {    //error
    System.out.println("Hai");
    }
}
```

**c) final to prevent inheritance:** The class that is declared as final implicitly declares all of its methods as final. The class cannot be extended to its subclass.

Example:
```
final class A
{
    void show( )
    {
    System.out.println("Hello");
    }
}
class B extends A                    //error
{
    void show( )
    {
    System.out.println("Hai");
    }
}
```

## 4.8 Object Class

Java provides a special class called "Object" class that is available in java.lang package. It is the super class of all other classes. So, a reference variable of type Object is created, it refers to object of any other class. Object class defines the following methods that are available in every object.



| Method | Purpose |
|---|---|
| Object clone( ) | Creates a new object that is the same as the object being cloned |
| boolean equals(Object object) | Determines whether one object is equal to another |
| void finalize( ) | Called before an unused object is recycled |
| Class getClass( ) | Obtains the class of an object at run time |
| int hashCode( ) | Returns the hash code associated with the invoking object |
| void notify( ) | Resumes execution of a thread waiting on the invoking object |
| void notifyAll( ) | Resumes execution of all threads waiting on the invoking object |
| String toString( ) | Returns a string that describes the object |
| void wait( ) | Waits on another thread of execution |
| void wait(long milliseconds) | |
| void wait(long milliseconds, int nanoseconds) | |

## 4.9 Array of Objects:

To instantiate more than one object for a single class, we use Array of Objects.

Example:
```
import javax.swing.JOptionPane;
class a
{
    int x,y;
    void readdata()
    {
```

```
        x=Integer.parseInt(JOptionPane.showInputDialog("Enter x value"));
        y=Integer.parseInt(JOptionPane.showInputDialog("Enter y value"));
        }
        void printdata()
        {
        System.out.println("x="+x+"\n"+"y="+y);
        }
}
class aob
{
        public static void main(String[] args)
        {
        a ob[]=new a[5];
        for(int i=0;i<5;i++)
        {
        ob[i]=new a();
        ob[i].readdata();
        ob[i].printdata();
        }
        }
}
```

## 4.10 Dynamic Method Dispatch

Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time. Dynamic method dispatch is important because this is how Java implements run-time polymorphism.

The important principle: a superclass reference variable can refer to a subclass object. Java uses this fact to resolve calls to overridden methods at run time. Here is how. When an overridden method is called through a superclass reference, Java determines which version of that method to execute based upon the type of the object being referred to at the time the call occurs. Thus, this determination is made at run time. When different types of objects are referred to, different versions of an overridden method will be called. In other words, *it is the type of the object being referred to* (not the type of the reference variable) that determines which version of an overridden method will be executed. Therefore, if a superclass contains a method that is

overridden by a subclass, then when different types of objects are referred to through a superclass reference variable, different versions of the method are executed.

Here is an example that illustrates dynamic method dispatch:

```
// Dynamic Method Dispatch
class A {
void callme() {
System.out.println("Inside A's callme method");
}
}
class B extends A {
// override callme()
void callme() {
System.out.println("Inside B's callme method");
}
}
class C extends A {
// override callme()
void callme() {
System.out.println("Inside C's callme method");
}
}
class Dispatch {
public static void main(String args[]) {
A a = new A(); // object of type A
B b = new B(); // object of type B
C c = new C(); // object of type C
A r; // obtain a reference of type A
r = a; // r refers to an A object
r.callme(); // calls A's version of callme
r = b; // r refers to a B object
r.callme(); // calls B's version of callme
r = c; // r refers to a C object
r.callme(); // calls C's version of callme
}
}
```

The output from the program is shown here:

```
Inside A's callme method
Inside B's callme method
Inside C's callme method
```

This program creates one superclass called **A** and two subclasses of it, called **B** and **C**. Subclasses **B** and **C** override **callme( )** declared in **A**. Inside the **main( )** method, objects of type **A**, **B**, and **C** are declared. Also, a reference of type **A**, called **r**, is declared.

The program then assigns a reference to each type of object to **r** and uses that reference to invoke **callme( )**. As the output shows, the version of **callme( )** executed is determined by the type of object being referred to at the time of the call. Had it been determined by the type of the reference variable, **r**, you would see three calls to **A**'s **callme( )** method.