UNIT II

INHERITANCE AND PACKAGES

1. **Introduction**
2. **Defining a Class**
3. **Adding Variables**
4. **Adding Methods**
5. **Creating Objects**
6. **Accessing Class Members**
7. **Constructors**
8. **Method Overloading**
9. **Static Members**
10. **Inheritance**
11. **Overriding Methods**
12. **Final Variables, Methods and Classes**
13. **Abstract Methods and Classes**
14. **Visibility Control**
15. **Packages**
    - **15.1 Introduction**
    - **15.2 Java API Package**
    - **15.3 Using System Package**
    - **15.4 Naming Conventions**
    - **15.5 Creating Packages**
    - **15.6 Accessing Package**

1. Introduction

- Java is a true object-oriented language and therefore the underlying structure of all java programs is classes.
- Anything we wish to represent in a java program must be encapsulated in a class that defines the *state* and *behaviour* of the basic program components known as *objects*.
- Classes provide a convenient method for packing together a group of logically related ***data items*** and ***functions*** that work on them.
- In Java, the **data items** are called ***fields*** and the **functions** are called ***methods.***

2. Defining a Class

- A class is a user-defined data type with a template that serves to define its properties.
- Once the class type has been defined, we can create "variables" of that type using declarations that are similar to the basic type declarations.
- In Java, these variables are termed as ***instances of classes***, which are the actual ***objects.***
- **Class Defines Data and Methods that manipulate the Data.**
- The basic form of a class definition is

```
class ClassName [extends SuperClassName]
{
    [fields declaration]
    [methods declaration]
}
```

- *classname* and *superclassname* are any valid java identifiers. The keyword extends indicates that the properties of the *superclassname* class are extended to the *classname* class.

3. Fields Declaration

- Data is encapsulated in a class by placing data fields inside the body of the class definition.

- These variables are called ***instance variables*** because they are created whenever an object of the class is instantiated.
- We can declare the instance variables exactly the same way as we declare local variables

```
Class Rectangle
{
        int length;
        int width;
}
```

- The class Rectangle contains two integer type instance variables.It is allowed them in one line as
  int length, width;
- Remember these variables are only declared and therefore no storage space has been created in the memory. ***Instance variables are also known as member variables.***

4. Methods Declaration

- A class with only data fields(and without methods that operate on that data) has no life.
- The objects created by such a class cannot respond to any messages.
- We must therefore add methods that are necessary for manipulating the data contained in the class.
- Methods are declared inside the body of the class but immediately after the declaration of instance variables.
- The General form of a method declaration is

```
type methodName(parameter-list)
{
        Method-body;
}
```

- Method declarations have four basic parts
  - The name of the method(methodname)
  - The type of the value the method returns(type)
  - A list of parameters(parameter-list)
  - The body of the method
- The ***type*** specifies the type of value the method would return. This could be a simple data type such as int as well as any class type. It could even be ***void*** type, if the method does not return any value.
- The ***methodname*** is a ***valid identifier.***
- The ***parameter list*** is always enclosed in ***parantheses***. This list contains variable names and types of all the values we want to give to the method as input. The variables in the list are separated by commas. when no input data are required , the declaration must retain the empty parantheses.
- ***The body actually describes the operations to be performed on the data.***

```
Class Rectangle
{
        int length,width;
        void getData(int x, int y) // method with void type
        {
                length =x;
                width =y;
        }
        int rectArea()          // method with return type
        {
                int area = length *width;
                return area;
        }
}
```

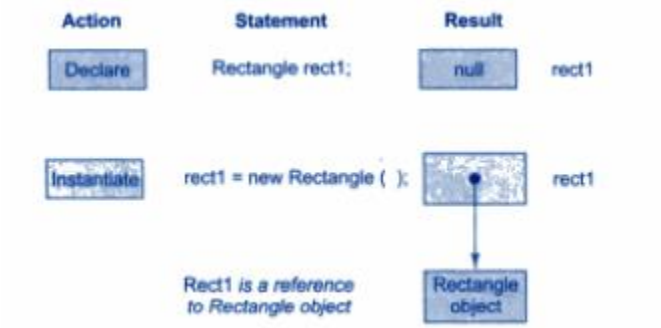
5. Creating Objects

- Instance of a class is called a n Object.

- Creating an object is also referred to as ***instantiating an object.***
- Objects in Java are created using the ***new*** operator.
- The ***new*** operator creates an object of the specified class and returns a reference to that object.

```
Rectangle rect1;                    // declare the object
rect1 = new Rectangle()             // instantiate the object
```

- The first statement declares a variable to hold the object reference and
- The second one actually assigns the object reference to the variable.
- The variable rect1 is now an object of the Rectangle class



- Both statements can be combined into one as
  Rectangle rect1 = new Rectangle()
- It is important to understand that each object has its own copy of the instance variables of its class.
- Means that any changes to the variables of one object have no effect on the variables of another.

6. Accessing Class Members

- Fields and Methods inside the class are called Members of class.
- The Class members can be Accesses using objectName, dot operator and class member(may be variable or method).

```
Objectname.variablename = value;
Objectname.methodname(parameter-list)
```

- Here *objectname* is the name of the object, *variablename i*s the name of the instance variable inside the object that we wish to access.
- *methodname* is the method that we wish to call, and
- *parameter-list* is comma separated list of "actual values" that must match in type and number with the parameter list of the methodname declaraed in the class**.**

```
class Rectangle
{
   int length, width;              //    Declaration of variables
   void getData(int x, int y)      //    Definition of method
   {
      length = x;
      width  = y;
   }
   int rectArea()                  //    Definition of another method
   {
      int area = length * width;
      return (area);
   }
}
class RectArea                     //    Class with main method
{
   public static void main (String args[ ])
   {
      int area1, area2;
      Rectangle rect1 = new Rectangle();      //    Creating objects
```

```
    Rectangle rect2 = new Rectangle();
    Rect1.length = 15;                          //    Accessing variables
    rect1.width  = 10;
    area1 = rect1.length * rect1.width;
    rect2.getData (20,12);                      //    Accessing methods
    area2 = rect2.rectArea();
    System.out.println("Area1 = " + area1);
    System.out.println("Area2 = " + area2);
  }
}
```

Output
Area1 = 150
Area2 = 240

- The first approach is to access the instance variables using the dot operator and compute the area. i.e ***int areal = rect1.lenght*rectl.width:***
- The second approach is to call the method rectArea declared inside the class. That is, ***int areal = rectl.rectArea(): // Calling the method***

7. Constructors

- Java supports a special type of method called a constructor, that enables an object o initialize itself when created.
- Constructors are used to initialize instance variables.

Constructor	Method
Constructor's are used to initialize instance variables	Methods are used to do general purpose calculation
Constructor Name and Class name should be same	Constructor name and Class name may or may not same
Constructor should have neither return type or void	Method should have either return type or void
Constructors are invoked at the time of object creation	Methods are invoked after object is created.

```
class Rectangle
{
    int length, width;
    Rectangle (int x, int y)                    // Defining constructor
    {
        length = x;
        width  = y;
    }
    int rectArea( )
    {
        return (length * width);
    }
}
class RectangleArea
{
    public static void main (string args[ ])
  {
      Rectangle rect1 = new Rectangle(15, 10); // Calling constructor
      int areal = rect1.rectArea( );
      System.out.println("Areal = "+ areal);
  }
}
```

output
Area1 = 150

8. Methods Overloading

- In Java it is possible to define two or more methods within the same class that share the same name, as long as their parameter declarations are different.
- (or) writing two or more methods within Same class with ***difference in type and/or number of arguments.***
- When this is the case, the methods are said to be *overloaded*, and the process is referred to as *method overloading*.
- Method overloading is one of the ways that Java supports ***polymorphism.***
- When an overloaded method is invoked, Java uses the type and/or number of arguments as its guide to determine which version of the overloaded method to actually call.
- Thus, overloaded methods must differ in the type and/or number of their parameters.

```
Class Addition
{
        int add(int x, int y)
        {
                int z = x+y;
                Return z;
        }
        int add(int x, int y,int z)
        {
                int  a = x+y+z;
                Return a;
        }
        double  add(double x, double y)
        {
                double z = x+y;
                Return z;
        }
  }
  Class MethodOverloading
  {
        public static void main(String args[])
        {
                Addition obj1 = new Addition();
                int add2 = obj1.add(2,3);
                int add3 = obj1.add(3,4,5);
                double add2d = obj1.add(2.4,3.5);
                System.out.println("Addition of 2 integer values: "+add2);
                System.out.println("Addition of 3 integer values :"+add3);
                System.out.println("Addition of 2 double values :"+add2d);
        }
  }
```

OUTPUT:

Addition of 2 integer values: 5

Addition of 3 integer values : 12

Addition of 2 double values:5.9

add is the only interface through which we are getting 3 types of output i.e addition of 2 integer values, addition of 3 integer values and addition of 2 double values.

When we call a method in an object, java matches up the method name first and then the number and type of parameters to decide which one of the definitions to execute. This process is known as ***polymorphism***

9. Static Members

Instance Method Vs Static Method

Class or Static Variables	Instance Variables
Class Variables are declared with keyword **static**	Instance Variables are declared without **static** keyword
Static Variables are common to all instances of a class. These variables are shared between the objects of a class	Instance variables are not shared between the objects of a class .Each instance will have their own copy of instance variables
As Static variables are common to all objects of a class, changes made to these variables through one object will reflect in another	As Each object will have its own copy of instance variables, changes made to these variables through one object will not reflect in another object.
Instance variables are created on heap memory.	Class variables are stored on method area.
Static variables can be accessed using either class name or object reference	Instance variable can be accessed only through object reference

- Instance Method can access the instance methods and instance variables directly
- Instance method can access static variables and static methods directly
- Static methods can access the static variables and static methods directly
- Static methods cant access instance methods and instance variables directly. They must use reference to object.
- And static method cant use this keyword as there is no instance for this to refer to.

10. Inheritance

- Reusability is an important concept of OOP paradigm.
- It is always nice if we could reuse something that already exists rather than creating the same all over again.
- Java supports this concept.
- Java classes can be reused in several ways. This is basically done by creating new classes, reusing the properties of existing ones.
- ***The mechanism of deriving a new class from an old class such that the new class acquires all the properties of the old class is called Inheritance.***
- The old class is known as Parent, base or Super class and the new class that is derived is known as child, derived or subclass.
- The Inheritance allows subclasses to inherit all the variables and methods of their parent classes.
- ***Defining a Subclass***
  - A Subclass is defined as follows

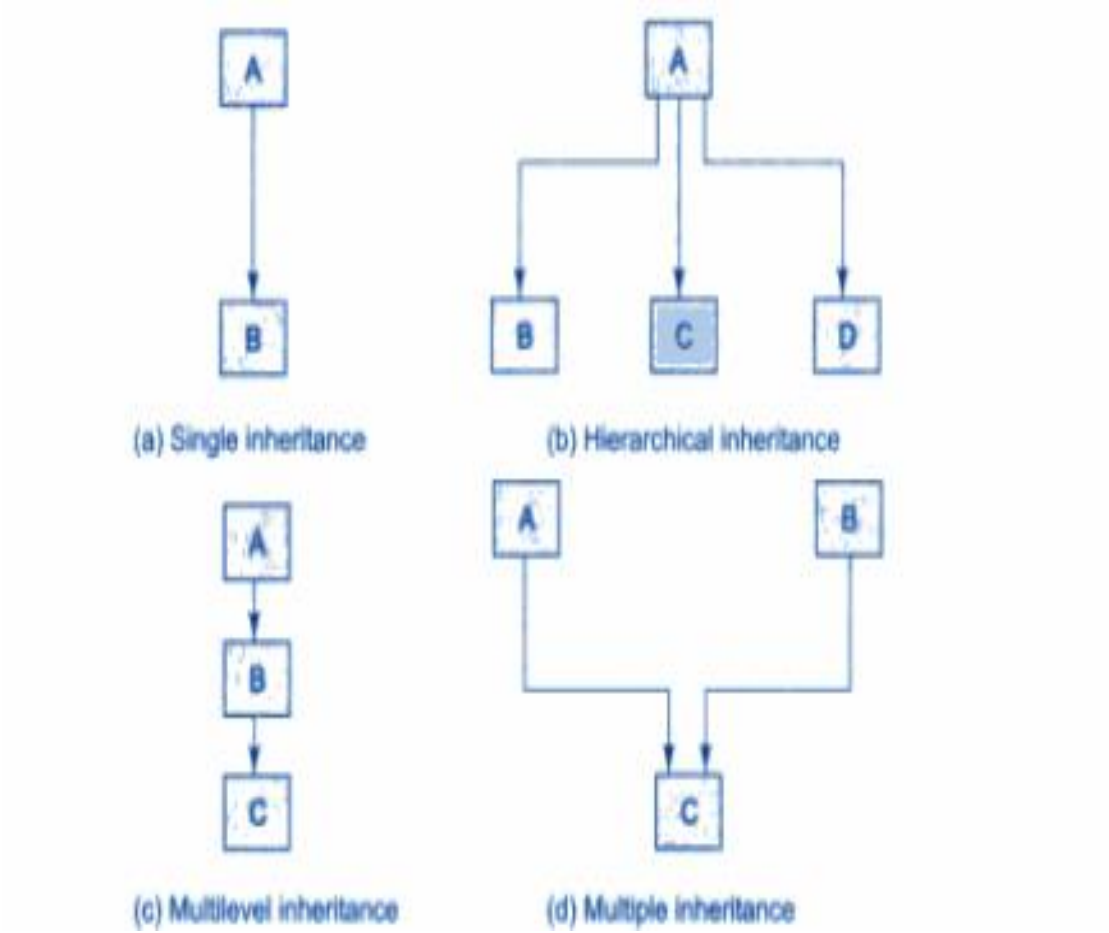
```
Class subclassname extends superclassname
{
        Variables declaration
        Methods declaration
}
```

  - The keyword ***extends*** signifies that the properties of the ***superclassname*** are extended ***subclassname***.
  - The subclass will now contain its own variables and methods as well those superclass.
  - This kind of situation occurs when we want to add some more properties to an existing class without actually modifying it.
- ***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 keyword super is used subject to the following conditions
  - Super may only be used within a subclass constructor method
  - The call to superclass constructor must appear as the first statement within the subclass constructor
  - The parameter in the super call must match the order and type of the instance variable declared in the superclass.

- ***Inheritance may take different types***
  1. Single inheritance
  2. Multilevel Inheritance
  3. Hierarchical Inheritance
  4. Hybrid Inheritance
  5. Multiple Inheritance

  These forms of inheritance are shown as



1. Single Inheritance
   The process of deriving one class from one base class is called single inheritance.

```
class Room
{
      int length;
      int breadth;
      Room(int x, int y)
      {
            length  = x;
            breadth = y;
      }
      int area( )
      {
            return (length * breadth);
      }
}
class BedRoom extends Room                    // Inheriting Room
{
      int height;
      BedRoom(int x, int y, int z)
      {
            super(x, y)                       // pass values to superclass
            height = z;
      }
      int volume( )
      {
            return (length * breadth * height);
      }
}
class InherTest
{
      public static void main(String args[ ])
      {
            BedRoom room1 = new BedRoom(14,12,10);
            int area1 = room1.area( );            // superclass method
            int volume1 = room1.volume( );        // baseclass method
            System.out.println("Area1 = "+ area1);
            System.out.println("Volume = "+ volume);
      }
}
```

Output :

Area1 = 168

Volume1 = 1680

2. Multilevel inheritance

Process of deriving a class from another derived class is called multilevel inheritance

/* PROGRAM TO ILLUSTRATE THE CONCEPT OF MULTILEVEL INHERITANCE */

```
class Employee
{
        int eno;
        double bsal;
        Employee(int e,double b)
        {
                eno=e;
                bsal=b;
        }
        double Hra()
        {
                return (bsal*0.08);
        }
        double Da()
        {
                return (bsal*0.11);
```

```
	}
	double Pf()
	{
		return (bsal*0.07);
	}
	void display()
	{
		System.out.println("Emp number="+eno);
		System.out.println("Basic salary="+bsal);
	}
}
class Gsalary extends Employee
{
	Gsalary(int e,double b)
	{
		super(e,b);
	}
	double gSalary()
	{
		return (bsal+Hra()+Da());
	}
}
class Nsalary extends Gsalary
{
	Nsalary(int e,double b)
	{
		super(e,b);
	}
	double nSalary()
	{
		return (gSalary()-Pf());
	}
}

public class Multilevel
{
	public static void main(String args[])
	{
	double hra,da,pf,gs,ns;
	Nsalary n1=new Nsalary(10,2000);
	hra=n1.Hra();
	da=n1.Da();
	pf=n1.Pf();
	gs=n1.gSalary();
	ns=n1.nSalary();
	n1.display();
	System.out.println("Hra="+hra);
	System.out.println("Da="+da);
	System.out.println("Pf="+pf);
	System.out.println("Gross salary="+gs);
	System.out.println("Net salary="+ns);
	}
}
```

OUTPUT

```
Z:\java>javac Multilevel.java
Z:\java>java Multilevel
Emp number=10
Basic salary=2000.0
Hra=160.0
Da=220.0
Pf=140.0
Gross salary=2380.0
Net salary=2240.0
```

3. **Hierachical Inheritance**

Process of deriving one or more subclasses from one super class is calledhierarchical inheritance

```
/* PROGRAM TO ILLUSTRATE THE CONCEPT OF HIERARCHICAL INHERITANCE */
class Rectangle
{
        double length,breadth;
        Rectangle(double l,double b)
        {
                length=l;
                breadth=b;
        }
        double areaRect()
        {
                return length*breadth;
        }
}
class Triangle extends Rectangle
{
        Triangle(double l,double b)
        {
                super(l,b);
        }
        double areaTri()
        {
                return (0.5*areaRect());
        }
}
class Volume extends Rectangle
{
        double height;
        Volume(double l,double b,double h)
        {
                super(l,b);
                height=h;
        }
        double volumeRect()
        {
                return (areaRect()*height);
        }
```

```
}

 class Hierarchical
{
        public static void main(String args[])
        {
                double a,b,c;
                Triangle obj1=new Triangle(1.2,2.3);
                Volume obj2=new Volume(1.2,2.3,3.4);
                a=obj1.areaRect();
                b=obj1.areaTri();
                c=obj2.volumeRect();
                System.out.println("Area of Rectangle="+a);
                System.out.println("Area of Triangle="+b);
                System.out.println("Volume of Rectangle="+c);
        }
}
```

OUTPUT

```
Z:\java>javac Hierarchical.java
Z:\java>java Hierarchical
Area of Rectangle=2.76
Area of Triangle=1.38
Volume of Rectangle=9.383999999999999
```

4. Hybrid Inheritance
   Combination of above any inheritance is called hybrid inheritance
5. Multiple inheritance
   Process of deriving a subclass from one or more superclasses is called multiple inheritance. Java does not directly implement multiple inheritance. however, this concept is implemented using a secondary inheritance path in the form of interfaces.

```
Class A
{
}
Class B
{
}
Class C extends A,B// java does not allow this
{
}
```

11. Overriding Methods

- A method defined in a super class is inherited by its subclass and is used by the objects created by the subclass.
- There may be occasions when we want an object to respond to the same method but have different behavior when that method is called means we should override the method defined in the superclass.
- 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 methods is called, the method defined in the subclass is invoked and executed instead of the one in the superclass. This is known as overriding

- ***A method in subclass, whose name , parameter list and return type are same as that of the method in superclass is called overrided methods.***

/*PROGRAM TO ILLUSTRATE THE CONCEPT OVERRIDING */

```
Class Square
{
        int a;
        Square(int side)
        {
                a = side;
        }

        void area()
        {
                int aSquare;
                aSquare = a*a;
                System.out.println("Area of the Square is : " + aSquare);
        }
}

class Cube extends Square
{
        Cube(int s)
        {
                super(s);
        }

        void area()
        {
                int aCube;
                aCube = a*a*a;
                System.out.println("Area of the Cube is : " + aCube);
        }
}

class Overriding
{
        public static void main(String args[])
        {
                Cube CubeObj = new Cube(4);
                CubeObj.area();
        }
}
```

OUTPUT

```
Z:\java pgms>javac Overriding.java
Z:\java pgms>java Overriding
Area of the Cube is : 64
```

12. Final Variables, Methods and Classes
    - Final keyword can be applied to variables, methods and classes
    - Final keyword before variable makes it ***constant***
    - Final keyword before a method ***prevents overriding***

- final keyword before a class ***prevents inheritance***

final variable : makes constant

to convert a variable as a constant, just use final keyword before variable
for eg:

```
final int i =10
int j =20;
i++; //error
j++;
```

because the value of constant cannot be changed during the execution of the program.the main difference between variable and constant is that variable value can be changed during the execution of the program where as the constant value cannot be changed during the execution of the program.

final method : prevents overriding

if we wish to prevent the subclasses from overriding the methods of the superclass, we can declare them as final using the keyword final as a modifier

```
class A
{
    final int m1()
    {
    }
}
Class B extends A
{
    int m1()        // error, trying to override a final method
    {
    }
}
```

Making a method final ensures that the functionality defined in this method will never be altered in any way.

final class : prevents inheritance

sometimes we may like to prevent a class being further subclasses for security reasons. A class that cannot be subclasses is called a final class. Any attempt to inherit final classes will cause an error and the compiler will not allow it.

```
final class A
{
}
class B extend A        //error, cannot inherit a because it is a final class
{
}
```

13 *Abstract Methods and Classes*

- An ***Abstract method*** is a method without method body or a method without implementation.
- An Abstract method is written when the same method has to perform different tasks depending on the object calling it.

  Example:

```
class A                    // Automatically Becomes Abstract Class
{
    void m1();             //    Abstract Method
    void m2()              //    Concrete Method
```

```
        {
            System.out.println("method 2");
        }
    }
```

- A Class that contains one or more Abstract Methods is called ***Abstract Class.***
- An Abstract class is a class that contains 0 or more Abstract Methods.
- Abstract class can contain instance variables and concrete methods in addition to abstract methods.
- Since, abstract class contains incomplete methods, it is not possible to estimate the total memory required to create the object.
- So, JVM can not create objects to an abstract class.
- We should create sub classes and all the abstract methods should be implemented in the sub classes.
- Then, it is possible to create objects to the sub classes since they are complete classes.

Let us make a program where the abstract class MyClass has one abstract method which has got various implementation in sub classes.

```
Abstract class MyClass
{
        abstract void calculate(double x);
}
Class Sub1 extends MyClass
{
        void  calculate(double x)
        {
                System.out.println("Square ="+(x*x));
        }
}
Class Sub3 extends MyClass
{
        void calculate(double x)
        {
                System.out.println("Square Root ="+Math.sqrt(x));
        }
}
Class Different
{
        public static void main(String args[])
        {
                Sub1  obj1 = new Sub1();
                Sub2  obj2 = new Sub2();
                Sub3  obj3 = new Sub3();
                obj1.calculate(3);
                obj2.calculate(4);
                obj3.calculate(5);
```

```
        }

}
```

```
C:\> javac Different.java
C:\> java Different
Square =9.0
Square root =2.0
Cube =125.0
```

Write a program in which abstract class Car contains an instance variable, one concrete method and two abstract methods .

```
Abstract class Car
{
        int regno;
        Car(int r)
        {
                regno =r;
        }
        void openTank()
        {
                System.out.println("Fill the Tank");
        }
        abstract   void steering();
        abstract  void braking( );
}
class Maruti extends Car
{
        Maruti(int regno)
        {
                super(regno);
        }
        void steering()
        {
                System.out.println("This car has ordinary Steering");
        }
        void braking()
        {
                System.out.println("These are hydraulic brakes");
        }
}
class  Santro  extends Car
{
        Santro(int regno)
        {
                super(regno);
        }
        void steering()
        {
```

```
                System.out.println("This car has Power Steering");
        }
        void braking()
        {
                System.out.println("These have gas brakes");
        }
}
class UseCar
{
        public static void main(String args[])
        {
                Maruti m = new Maruti(1001);
                Santro s = new Santro(5005);
                Car ref;
                ref  = m;
                ref.openTank();
                ref.steering();
                ref.braking();
                ref = s;
                ref.openTalk();
                ref.steering();
                ref.braking();
        }
}
```

Output:

Fill the tank
This car has ordinary Steering
These are hydraulic brakes
Fill the tank
This car has Power Steering
These have gas brakes

It is perfectly possible to access all the members of the subclasses by using sub class objects. But , we prefer to use super class reference to access the sub class features because, the reference variable can access only those features of the sub classes which have been already declared in super class. If we write individual method in the subclass, the super class reference cannot access that method. This is to enforce discipline in the programmers not to add any of their own features in the sub classes other than whatever is given in super class. For example, a programmer has added the following method in Maruti class

```
void wings()
{
        System.out.println(" I can fly");
}
```