UNIT-II

Classes and Objects

Class:

Class is a collection of similar type of objects.

Class defines a new data type that can be used for creating objects.

Declaration of class:

In JAVA language, class declaration can be done by using the following syntax.

Syntax:

```
class classname
{
    datatype instance_var1;
    datatype instance_var2;
    .
    .
   datatype instance_varn;
   returntype methodname1(parameter_list)
   {
      //body
   }
   .
   .
  returntype methodnamen(parameter_list)
  {
     //body
   }
}
```

Variables which are declared inside a class are called as instance variables. Because each instance of a class can hold a separate copy of these variables.

Class also contains n number of methods which are used to perform operations on data.

Variables and methods which are defined inside a class are collectively known as members of a class.

Example:

```
class Box

{

    double width, height, depth;

}
```

After defining a class, by using that class we can create any number of objects.

Creation of Object:

Object creation can be done in two steps.

In first step, class variable is created or declared by using the class name.

Syntax:

```
classname class_var;
```

In second step, the memory is allocated for an object at run time using **new** operator.

Syntax:

```
Class_var=new classname();
```

Here parenthesis followed by classname represents a constructor.

Constructor is used to initialize the instance variables.

If the programmer not writing any constructor in a class, then JAVA compiler provides one default constructor.

The above two steps are combined into a single statement as follows.

```
classname class_var=new classname();
```

Example:

Box b1=new Box();

After creating objects, we can access the class members using those objects.

To access instance variables, the following syntax is used.

Syntax:

```
objectname.variblename=value;
```

To call and execute any method in a class, the following syntax is used.

Syntax:

```
objectname.methodname();
```

Write a program to illustrate the use of class and objects.

```
//program to illustrate the use of class and objects.
class Box
{
  double width, height, depth;
}
class BoxDemo
{
  public static void main(String args[])
 {
    Box b1=new Box();
   double vol;
    b1.width=15.35;
    b1.height= 20.56;
    b1.depth=10.34;
   vol=b1.width*b1.height*b1.depth;
   System.out.println("Volume is "+vol);
}
}
```

Defining methods or Adding methods to a class:

General form of a method is as follows

```
returntype method_name(parameter_list)
{
   //body of the method
}
```

Here returntype represents the type of the data returned by the method. This can be any valid type including class types that we create.

If the method does not return a value, its return type must be void.

method_name is any valid identifier.

parameter_list is a sequence of type and identifier pairs separated by commas.

If the method has no parameters, then parameter_list is empty.

Example:

In the above example program, we are calculating volume of a box in BoxDemo class.

Now we are calculating the volume of a box in Box class by adding method to Box class.

```
class Box
{
   double width, height, depth;
   void volume()
  {
     System.out.println("Volume is "+ (width*height*depth));
  }
}
class BoxDemo
{
   public static void main(String args[])
  {
    Box b1=new Box();
   Box b2=new Box();
   b1.width=10.25;
   b1.height=20.35;
   b1.depth=15.45;
   b2.width=3.45;
  b2.height=6.78;
   b2.depth=9.85;
  b1.volume();
  b2.volume();
```

```
}
}
```

Methods returning values:

```
class Box
{
   double width, height, depth;
   double volume()
  {
     return width*height*depth;
  }
}
class BoxDemo
{
   public static void main(String args[])
  {
    Box b1=new Box();
   Box b2=new Box();
   double vol;
   b1.width=10.25;
   b1.height=20.35;
   b1.depth=15.45;
   b2.width=3.45;
  b2.height=6.78;
   b2.depth=9.85;
  vol=b1.volume();
 System.out.println("Volume is "+vol);
  vol=b2.volume();
 System.out.println("Volume is "+vol);
```

```
}
}
```

Adding a method that takes parameters:

Parameters allow a method to be generalized.

Parameterized method can operate on variety of data and/or be used in number of slightly different situations.

```
class Box
{
   double width, height, depth;
   double volume()
   {
      return width* height * depth;
  }
 void setDim(double w, double h, double d)
 {
     width=w;
     height=h;
      depth=d;
  }
}
class BoxDemo
{
    public static void main(String args[])
   {
      Box b1=new Box();
      Box b2=new Box();
      double vol;
       b1.setDim(2.35,4.65,6.78);
```

```
    b2.setDim(5.67, 7.89,8.97);
    vol=b1.volume();
    System.out.println("Volume is "+vol);
    vol=b2.volume();
    System.out.println("Volume is "+vol);
}
}
```

Overloaded Methods:

In JAVA language, it is possible to define two or more methods within a same class with same name but with different number and/or types of arguments.

Then the methods are said to be overloaded and the process is referred as method overloading.

Method overloading is one of the ways to achieve polymorphism.

Method overloading is an example for compile time polymorphism.

When overloaded method is invoked, JAVA compiler uses the type and/or number of arguments to determine the method which is to be executed.

Write a program to illustrate method overloading.

```
class OverLoad
{
   void display()
  {
     System.out.println("No Parameters");
   }
 void display(int a)
  {
       System.out.println("a= "+a);
   }
void display(int a, int b)
 {
```

```
    System.out.println("a= "+a+" "+"b= "+b);
  }
double display(double a)
{
    System.out.println("double a= "+a);
    return a*a;
 }
}
class OverLoadDemo
{
    public static void main(String args[])
     {
        OverLoad ol=new OverLoad();
        ol.display();
       ol.display(10);
       ol.display(10,20);
      double result=ol.display(3.5);
      System.out.println("Result= "+result);
   }
}
```

In some cases, JAVA's automatic type conversion can play a role in overloaded resolution.

For example, consider the following program

```
class OverLoad
{
   void display()
   {
      System.out.println("No Parameters");
```

```
   }
void display(int a,int b)
 {
    System.out.println("a= "+a+"b= "+b);
}
void display(double a)
{
  System.out.println("double a="+a);
}
}
class OverLoadDemo
{
    public static void main(String args[])
    {
        OverLoad ol=new OverLoad();
        ol.display();
        ol.display(10,20);
        ol.display(10);
        ol.display(15.65);
  }
}
```

Here we are not defined a method display(int) in OverLoad class. Therefore when a displaly() is called with an integer argument in OverLoadDemo, no matching method is found. However, JAVA can automatically converts integer to double and this conversion can be used to resolve the call.

Recursive methods:

A method which is called by itself is called as recursive method.

Process of calling recursive method is known as recursion.

Write a program to illustrate recursion.

```
//program to illustrate recursion
import java.util.*;
class Factorial
{
   int fact(int n)
   {
       if(n==0||n==1)
         return 1;
       else
         return n*fact(n-1);
   }
}
class Recursion
{
     public static void main(String args[])
     {
        int val;
       Factorial f=new Factorial();
        Scanner sc=new Scanner(System.in);
       System.out.println("enter one number");
        val=sc.nextInt();
       System.out.println("Factorial of  "+val+"is "+f.fact(val));
     }
}
```

Access control for class members:

Access modifier is used to determine how a member can be accessed.

JAVA language provides the following access modifiers.

- public
- private
- protected

JAVA also defines default access level.

When a member of a class is specified as public, then that member can be accessed by any other code.

When a member of a class is specified as private, then that member can only be accessed by other members of its class.

To understand public and private access modifiers, consider the following program.

```
// This program demonstrates the difference between public and private.
class Test
{
int a;
public int b;
private int c;
void setc(int i)
{
c = i;
}
int getc()
{
return c;
}
}

class AccessTest

{

public static void main(String args[]) {

Test ob = new Test();
ob.a = 10;
ob.b = 20;
ob.setc(100);
System.out.println("a, b, and c: " + ob.a + " " +ob.b + " " + ob.getc());
}
}
```

Constructors:

Constructor is a special method which is used to initialize the instance variables.

Rules for writing constructors:

1) Constructor name is same as class name in which it resides.
2) Constructor doesn't have any return type not even void also.
3) Constructor cannot be abstract, final, static and synchronized.
4) Access modifiers can be used for declaration of constructors.

Note: When constructor is not defined in a class, JAVA compiler provides one default constructor to a class automatically. The default constructor automatically initializes the instance variables to their default values.

Constructors cannot be called explicitly; Constructor is automatically called when an object is created.

Types of constructors:

There are two types of constructors in JAVA.

1) Zero-argument constructor(Default constructor)
2) Parameterized constructor

Zero-argument constructor:

A constructor that has no parameters is called as zero-argument constructor.

Syntax:

```
classname()
{
  //body  of the constructor
}
```

Example:

```
class Box
{
   double width, height, depth;
   Box()
   {
      System.out.println("Constructing Box....");
     width=10.25;
      height=12.35;
```

```
      depth=15.45;
    }
  double volume()
  {
    return width*height*depth;
   }
}
class ConstructorDemo
{
  public static void main(String args[])
  {
      Box b1=new Box();
      Box b2=new Box();
     double vol;
      vol=b1.volume();
      System.out.println("Volume is "+vol);
      vol=b2.volume();
     System.out.println("Volume is "+vol);
}
}
```

Parameterized Constructor:

While the Box() constructor in the previous program does initialize a Box object, it is not very useful because all the boxes have same dimensions.

To construct Box objects of various dimensions, we have to add parameters to constructor.

A constructor that has parameters is called as parameterized constructor.

Syntax:

```
classname(parameter_list)
{
   //body of the constructor
}
```

Example:

```
class Box
{
    double width, height, depth;
    Box(double w, double h, double d)
    {
        System.out.println("Constructing Box....");
       width=w;
        height=h;
      depth=d;
    }
 double volume()
 {
   return width*height*depth;
  }
}
class ConstructorDemo
{
  public static void main(String args[])
  {
      Box b1=new Box(10.25,13.35,15.45);
      Box b2=new Box(23.45,34.56,56.78);
     double vol;
```

```
        vol=b1.volume();
        System.out.println("Volume is "+vol);
        vol=b2.volume();
       System.out.println("Volume is "+vol);
}
}
```

Constructor overloading:

Like methods, constructors can also be overloaded.

Constructor overloading means writing two or more constructors with different types or number of parameters.

Example:

```
class A
{
     int a, b;
    A()
    {
        a=0;
       b=0;
     }
A(int x)
{
   a=x;
   b=5;
}
A(int x, int y)
{
   a=x;
    b=y;
```

```
}
void display()
{
   System.out.println(a+" "+b);
}
}
class ConstructorOverloadDemo
{
  public static void main(String args[])
  {
       A a1=new A();
      A a2=new A(3);
      A a3=new A(10,20);
      a1.display();
      a2.display();
     a3.display();
}
}
```

static keyword in JAVA:

When a member is declared static, it can be accessed before any objects of its class are created and without reference to any object.

We can declare both methods and variables to be static.

Instance variables declared as static are essentially global variables. When objects of its class are declared, no copy of static variable is made. Instead, all objects of the class share the same static variable.

Methods declared as static have several restrictions.

They can only call other static methods.

Then can only directly access static variables.

They cannot refer to this or super in any way.

If we want to do computation in order to initialize static variables, we can declare a static block that gets executed only once when the class is first loaded.

Write a program to illustrate static variables, methods and blocks.

```
//Program to illustrate static variables, methods and blocks

class StaticDemo

{

    static int a=3;

   static int b;

   static void display(int x)

   {

       System.out.println("x="+x);

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

      System.out.println("b="+b);

    }

 static

{

System.out.println("Static block is initialized");

 b=a*4;

}

public static void main(String args[])

{

  display(5);

}

}
```

Outside of the class in which they are defined, static methods and variables can be used independently of any object.

To call a static method from outside its class, we use class name and dot operator as follows.

```
classname.method();
```

Here classname is the name of the class in which static method is declared.

In the same way, static variable can be accessed by using its class name and dot operator as follows.

```
classname.variable_name;
```

Example:

//Program to illustrate how to call and access static variables and methods from outside class.

```
class StaticDemo
{
    static int a=20;
    static int b=30;
     static void display()
     {
       System.out.println("a="+a);
      }
}
class StaticByName
{
   public static void main(String args[])
   {
      StaticDemo.display();
     System.out.println("b="+StaticDemo.b);
   }
}
```

this keyword:

In Java, this is a keyword which is **used to refer current object** of a class. We can use it to refer any member of the class. It means we can access any instance variable and method by using **this** keyword.

The main purpose of using **this** keyword is to solve the confusion when we have same variable name for instance and local variables.

We can use this keyword for the following purpose.

- **this** keyword is used to refer to current object.
- **this** is always a reference to the object on which method was invoked.
- **this** can be used to invoke current class constructor.
- **this** can be passed as an argument to another method.

Example:

In this example, we have three instance variables and a constructor that have three parameters with **same name as instance variables**. Now, we will use this to assign values of parameters to instance variables.

```
class Demo
{
 double width, height, depth;
 Demo (double width, double height, double depth)
 {
  this.width = width;
  this.height = height;
  this.depth = depth;
 }
 public static void main(String[] args) {
        Demo d = new Demo(10,20,30);
        System.out.println("width = "+d.width);
        System.out.println("height = "+d.height);
        System.out.println("depth = "+d.depth);
```

```
}
}
```

Calling constructor using this keyword:

We can call a constructor from inside another constructor using this keyword.

```
class Demo
{
 Demo ()
 {
   this("VIT-IT");
 }
 Demo(String str){
            System.out.println(str);
 }
  public static void main(String[] args) {
            Demo d = new Demo();
 }
}
```

Accessing method using this keyword:

```
class Demo
{
            void getName()
            {
             System.out.println("VIT-IT");
            }
            void display()
            {
             this.getName();
            }
```

```
    public static void main(String[] args) {
        Demo d = new Demo();
        d.display();
    }
}
```

Inheritance:

Inheritance is the process by which one object acquires the properties of another object.

(or)

Deriving a class from another class is called as inheritance.

Superclass (or) Parent class (or) base class: A class that is inherited is called as superclass.

Subclass (or) Child class (or) derived class: A class that does inheriting is called as subclass.

Types of inheritance:

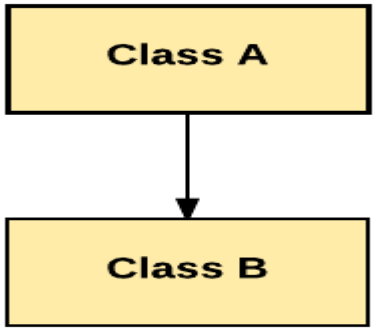
Generally there are 5 types of inheritance

They are

1) Single inheritance
2) Multiple inheritance
3) Multi-level inheritance
4) Hierarchical inheritance
5) Hybrid inheritance

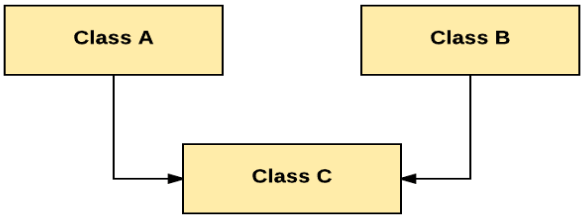
Single inheritance:

Deriving a sub class from single super class is called as single inheritance.



Multiple inheritance:

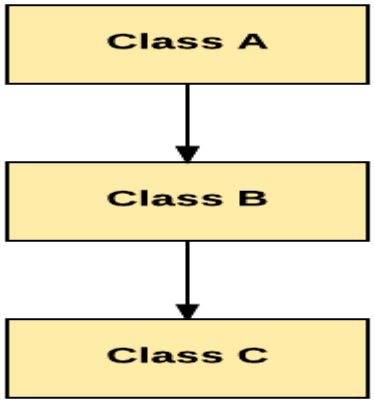
Deriving a sub class from more than one super class is called as multiple inheritance.



Note: JAVA does not support multiple inheritance directly. We can achieve multiple inheritance in JAVA by using interfaces.

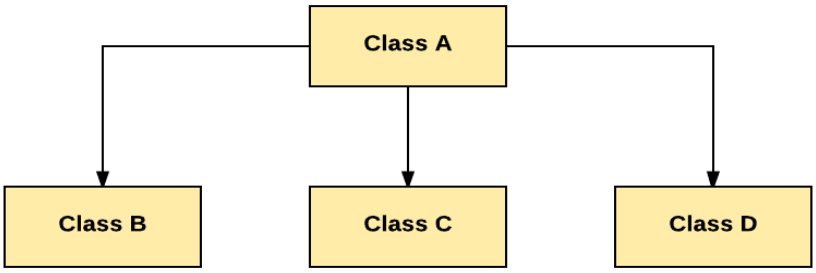
Multi-level inheritance:

In multi-level inheritance, one sub class is derived from another sub class. Hence one sub class becomes super class for a new class.



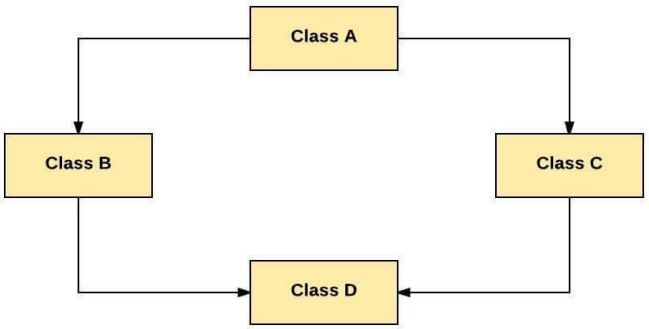
Hierarchical inheritance:

Deriving more than one sub class from a single super class is called as hierarchical inheritance.



Hybrid inheritance:

Hybrid inheritance is a combination of two or more types of inheritances.



Note: JAVA does not support hybrid inheritance.

In JAVA language, in order to derive a class from another class we use **extends** keyword.

General form of deriving a class as follows.

Syntax:

```
class sub_class_name extends super_class_name
{
    //body of a class
}
```

By doing inheritance, all the behaviour of superclass is available in sub class. That means we can access superclass members by using subclass objects.

//Program to illustrate inheritance

```
class A

{

  int i, j;

  void show()
```

```
 {
    System.out.println("i="+i+" "+"j="+j);
  }
}
class B extends A
{
  int k;
  void display()
 {
    System.out.println("k="+k);
  }
void sum()
{
  System.out.println("sum="+(i+j+k));
}
}
class InheritanceDemo
{
 public static void main(String args[])
 {
    A a1=new A();
   a1.i=10;
  a1.j=20;
   a1.show();
   B b1=new B();
  b1.i=3;
 b1.j=5;
 b1.k=7;
```

```
 b1.show();
  b1.display();
  b1.sum();
}
}
```

Note: Although a subclass includes all the members of its superclass, it cannot access the member of the superclass that has been declared as **private**.

super keyword:

Whenever a subclass needs to refer to its immediate superclass, it can do so by use of the keyword **super**.

super keyword has two general forms

1) To call superclass constructor
2) To access member of superclass that has been hidden by member of subclass.

Calling superclass constructor:

A subclass can call a constructor defined by its superclass by using the following syntax.

Syntax:

```
super(arg_list);
```

Her arg_list specifies any arguments needed by the constructor in the superclass.

super() must always be the first statement executed inside a subclass constructor.

//Program to illustrate calling superclass constructor using super keyword

```
class A
{
  int i, j;
 A(int x, int y)
 {
    i=x;
    j=y;
```

```
  }
 void show()
{
   System.out.println("i="+i+" "+"j="+j);
 }
}
class B extends A
{
   int k;
    B(int a, int b, int c)
   {
      super(a, b);
      k=c;
  }
void display()
{
   System.out.println("k="+k);
 }
void sum()
{
   System.out.pritnln("sum="+(i+j+k));
 }
}
class SuperDemo
{
   public static void main(String args[])
  {
        B b1=new B(1,2,3);
```

```
        b1.show();
        b1.display();
       b1.sum();
}
}
```

Accessing superclass members using super keyword:

To access superclass members which are hidden by subclass members, the following syntax is used.

Syntax:

```
super.member;
```

Here member can be either an instance variable or a method name.

//Program to access super class members using super keyword

```
class A
{
   int i;
 }
class B extends A
{
   int i;
   B(int x, int y)
   {
     super.i=x;
     i=y;
   }
 void show()
```

```
{
    System.out.println("i in super class="+super.i);
    System.out.println("i in sub class="+i);
}
}
class SuperDemo
{
 public static void main(String args[])
 {
    B b1=new B(10,20);
     b1.show();
  }
}
```

Method overriding:

In a class hierarchy, when a method in a subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to *override* the method in the super class.
When an overridden method is called from within its subclass, it will always refer to the version of that method defined by the subclass. The version of the method defined by the super class will be hidden.
To call method defined by superclass , the keyword super is used in subclass method.

//Program to illustrate method overriding

```
class A
{
   void show()
   {
       System.out.println("superclass method");
    }
}
```

```
class B extends A
{
   void show()
    {
       super.show();
       System.out.println("subclass method");
     }
}
class OverridingDemo
{
    public static void main(String args[])
    {
         B b1=new B();
         b1.show();
     }
}
```

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.

In Dynamic Method Dispatch, superclass reference variable can refer the subclass objects. By using superclass reference variable, we can call the overridden methods.

```
class A
```

```
{
    void show()
    {
       System.out.println("super class method");
     }
}
class B extends A
{
    void show()
    {
       System.out.println("sub class1 method");
     }
}
class C extends B
{
    void show()
   {
      System.out.println("sub class2 method");
     }
}
Class Demo
{
    public static void main(String args[])
    {
       A a1=new A();
       B b1=new B();
       C c1=new C();
       A a;
```

```
        a=a1;
        a.show();
         a=b1;
        a.show();
          a=c1;
        a.show();
    }
}
```

Abstract classes and methods:

To define generalized behaviour of a class, we use abstract methods and abstract classes.

Abstract method: A method which is not implemented is called as abstract method.

Abstract methods can be declared using the keyword 'abstract' by using the following syntax

```
abstract  returntype method_name(parameter_list);
```

Example:

```
abstract void display();
```

Abstract class: Any class which contains at least one abstract method is called as abstract class.

To declare abstract class, the keyword 'abstract' is used before the class declaration.

Syntax:

```
abstract class classname
{
  ..........
  ..........
   ..........
}
```

In abstract class, in addition to abstract methods, concrete methods are also available.

Abstract methods should be implemented in the sub classes of abstract class. If sub class also not implemented abstract method then make the sub class also as abstract.

We cannot create objects for abstract classes.

We can create only reference variable for abstract classes.

Write a java program for abstract class to find areas of different shapes

```
//Program to find areas of different shapes using abstract class
abstract  class Shape
{
    int dim1,dim2;
   Shape(int  x,int y)
{
     dim1=x;
    dim2=y;
 }
abstract double area();
}
class Triangle extends Shape
{
     Triangle(int a,int b)
     {
          super(a,b);
      }
   double area()
   {
       double a1=(dim1*dim2)/2;
       return a1;
   }
}
class   Rectangle extends Shape
```

```
{
     Rectangle(int i,int j)
     {
        super(i,j);
      }
    double area()
    {
        double a2=dim1*dim2;
       return a2;
    }
}
class AbstractDemo
 {
    public static void main(String args[])
    {
        Shape s;
        Triangle t=new Triangle(10,20);
       s=t;
        System.out.println(s.area());
       Rectangle r=new Rectangle(15,25);
       s=r;
      System.out.println(s.area());
    }
}
```

Final keyword:

The final keyword in java is used to restrict the user

Final is a **non-access modifier** applicable **only to a variable, a method or a class**.

Basically final keyword is used to perform 3 tasks as follows –

1. To create constant variables
2. To prevent method overriding
3. To prevent Inheritance

Final variables:

Once we declare a variable with the final keyword, we can't change its value again. If we attempt to change the value of the final variable, then we will get a compilation error.

You can initialize a final variable when it is declared. A final variable is called **blank final variable**, if it is **not** initialized while declaration.

We can initialize a blank final variable inside the constructor of the class.

Example:

```
final int a=10;

final int b;
```

final methods:

A method which is declared with the keyword final cannot be overridden.

If we try to override final method, then it will give compile time error.

Example:

```
class A
{
   final void show()
    {
       System.out.println("final method");
    }
}
class B extends A
{
   void show()      //This will give compile time error
   {
     System.out.println("sub class method");
```

```
    }
}
```

Final classes:

A class which is declared with the keyword final cannot be inherited.

Example:

```
final class A
{
   void show()
  {
     System.out.println("super class method");
   }
}
class B extends A          //This will give a compile time error
{
    void show()
    {
      System.out.println("sub class method");
     }
}
```