Module II

Classes fundamentals, objects, methods, constructors, parameter passing, overloading, access control keywords.

Object Oriented Programming

In object-oriented programming technique, we design a program using objects and classes.

Object is the physical as well as logical entity whereas class is the logical entity only.

Object in Java

An entity that has state and behavior is known as an object e.g. chair, bike, marker, pen, table, car etc.

It can be physical or logical (tangible and intangible). The example of intangible object is banking system.

An object has three characteristics:

state: represents data (value) of an object.

behavior: represents the behavior (functionality) of an object such as deposit, withdraw etc.

identity: Object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. But, it is used internally by the JVM to identify each object uniquely.

For Example: Pen is an object. Its name is Reynolds, color is white etc. known as its state. It is used to write, so writing is its behavior.

Object is an instance of a class. Class is a template or blueprint from which objects are created. So object is the instance (result) of a class.

Object Definitions:

Object is a real world entity.

Object is a run time entity.

Object is an entity which has state and behavior.

Object is an instance of a class.

Class in Java

A class is a group of objects which have common properties. It is a template or blueprint from which objects are created. It is a logical entity. It can't be physical.

A class in Java can contain:

- **fields**
- **methods**

constructors

blocks

nested class and interface

General Form of a class

Data and methods within the class are called members of the class

```
class Classname
{
 type instance_variable1;
 type instance_variable2;
 …..
 . type instance_variableN;
 type methodname1(parameter-list)
 {
 }
 type methodname2(parameter-list)
 {
 }
.
.
type methodnameN(parameter-list)
 {
 }
 .
 .
}
```

Instance variable in Java

A variable which is created inside the class but outside the method is known as instance variable. Instance variable doesn't get memory at compile time. It gets memory at run time when object (instance) is created. That is why, it is known as instance variable.

Example

```
class Box
{
   int breadth;
```

```
    int height;
    int length;
}
```

Here length,height,breadth are **instance variables** of class Box

Creating and accessing Object

Classname objectname= new Classname();

Box b=new Box ();

For accessing object we use dot (.) Operator

b.height=100;

Object and Class Example: main within class

In this example, we have created a **Student** class that have two data members id and name. We are creating the object of the Student class by new keyword and printing the objects value.

Here, we are creating main() method inside the class.

```
class Student
{
 int id;          //field or data member or instance variable
 String name;
   public static void main(String args[])
{
  Student s1=new Student();//creating an object of Student
  System.out.println(s1.id);//accessing member through reference variable
  System.out.println(s1.name);
 }
}
```

Output:

0

NULL

Object and Class Example: main outside class

In real time development, we create classes and use it from another class. It is a better approach than previous one. Let's see a simple example, where we are having main() method in another class.

```
class Student
```

```
{
 int id;
 String name;
}
class TestStudent
{
 public static void main(String args[])
 {
  Student s1=new Student();
  System.out.println(s1.id);
  System.out.println(s1.name);
 }
}
```

Output:

0

NULL

Object and Class Example: Initialization through reference

Initializing object simply means storing data into object. Let's see a simple example where we are going to initialize object through reference variable.

```
class Student
        {
        int id;
        String name;
        }
class TestStudent
        {
         public static void main(String args[])
         {
          Student s1=new Student();
          s1.id=101;
          s1.name="Sonoo";
          System.out.println(s1.id+" "+s1.name);//printing members with a  space
```

```
        }
    }
```

Output:

101 Sonoo

Example Program

```
class Box
{
        float length;
        float height;  // attributes of the class
        float breadth;
}
class Mainbox
{
  public static void main(String args[])
  {
        Box b=new Box();   // object creation
        float v;
        b.length=10;        // accessing object
        b.height=20;
        b.breadth=15;
        v=b.length*b.height*b.breadth;
        System.out.println("Volume is " + v);
  }
}
```

We can also create multiple objects and store information in it through reference variable.

```
class Student
{
 int id;
 String name;
}
class TestStudent
{
 public static void main(String args[])
 {
```

```
//Creating objects
Student s1=new Student();
Student s2=new Student();
//Initializing objects
s1.id=101;
s1.name="Sonoo";
s2.id=102;
s2.name="Amit";
//Printing data
System.out.println(s1.id+" "+s1.name);
System.out.println(s2.id+" "+s2.name);
}
}
```

Output:

101 Sonoo

102 Amit

Method in Java

In java, a method is like function i.e. used to expose behavior of an object.

A Java method is a collection of statements that are grouped together to perform an operation. When you call the System.out.**println()** method, the system actually executes several statements in order to display a message on the console.

Now you will learn how to create your own methods with or without return values, invoke a method with or without parameters, and apply method abstraction in the program design.

The process of method calling is simple. When a program invokes a method, the program control gets transferred to the called method. This called method then returns control to the caller in two conditions, when –

- the return statement is executed.
- it reaches the method ending closing brace.

The methods returning void is considered as call to a statement.

Syntax

```
access   returntype   methodname(parameters)
{
        //body of the method    }
```

acess – means public, private or protected

Example

```
class Box
{  float length;
   float height;
   float breadth;
   void volume()
  {
          float v;
           v= length*height*breadth;                    // body of the method volume()
          System.out.println("The  volume is " +v);
  }
  }
class Mainbox
{
  public static void main(String args[])
  {
     Box b=new Box();
     b.length=10;
     b.height=20;
     b.breadth=30;
     b.volume();    // Method calling(caller)
     }
}
```

Using return

The return statement is used to explicitly return from a method. That is, it causes program control to transfer back to the caller of the method

At any time in a method the return statement can be used to cause execution to branch back to the caller of the method.

```
class Box
{  float length;
   float height;
   float breadth;
```

```
 float volume()
{
        float v;
       v= length*height*breadth;
      return v;
}

class Mainbox
{
 public static void main(String args[])
 {
  Box b=new Box();
  float vol;
  b.length=10;
  b.height=20;
  b.breadth=30;

  vol=b.volume();
  System.out.println(vol);
}
}
```

Method with parameters

Method parameters make it possible to pass values to the method, which the method can operate on. The method parameters are declared inside the parentheses after the method name.

```
class Box
{  float length;
  float height;
 float breadth;
 float volume(double l, double h, double b)
 {
  length=l;
  height=h;
 breadth=b;
  return length*height*breadth;
```

```
  }
   }
class Mainbox
{
 public static void main(String args[])
 {
  Box b=new Box();
  float vol;
    vol=b.volume(10,20,30);
  System.out.println(vol);
}
}
```

Constructor

Constructor in java is a special type of method that is used to initialize the object.

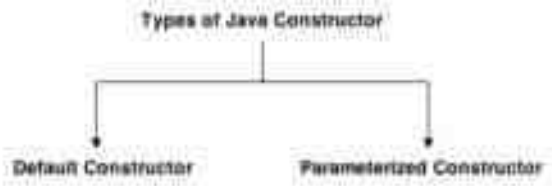
Java constructor is invoked at the time of object creation. It constructs the values i.e. provides data for the object that is why it is known as constructor.

- It has same name of the class and resides same as a method
- Once defined a constructor will be called automatically after an object is created
- There is no return type not even void
- The implicit return type is the type of the class
- It is set a platform to initialize all internal state of an object

Types of java constructors

Default constructor (no-arg constructor)

Parameterized constructor



A constructor that have no parameter is known as default constructor.

Example Program for default constructor

```
class Box
{  double length;
```

```
  double height;
  double breadth;
 Box()
 {
  length=10;
  height=20;
  breadth=30;
 }
 double volume()
  {
 double v;
 v= length*height*braedth;
  return  v;
 }
  }
class Mainbox
{
  public static void main(String args[])
  {
   Box b=new Box();
    double v;
    v= b.volume();
   System.out.println(v);
   }
}
```

Parameterized constructor

- Add parameters to the constructors

Example

```
 class Box
 { double length;
   double height;
   double braedth;
  Box(double l, double h, double b)
  {
```

```
        length=l;              //Parameterized Constructor
         height=h;
         breadth=b;
        }
        double volume()
         {
         return length*height*breadth;
         }
         }
class  Mainbox
  {
   public static void main(String args[])
    {
       Box b=new Box(10,20,30);
        double v;
       v= b.volume();
        System.out.println(v);
    }
       }
```

Overloading Constructors

- Constructors are overloaded here.
- Constructor overloading is a technique in Java in which a class can have any number of constructors that differ in parameter lists
- The compiler differentiates these constructors by taking number of parameters in the list and their type.

```
class Box
{  double length;
  double height;
  double breadth;
 Box()
 {
  length=height=breadth=1;
 }
```

```
 Box(double len)
 {
  length=height=breadth=len;
 }
 Box(double l, double h, double b)
 {
  length=l;
  height=h;
  breadth=b;
 }
 }
class Mainbox
{
 public static void main(String args[])
 {
  Box b1=new Box();
  Box b2=new Box(40);
  Box b3=new Box(100,200,300);
  System.out.println("Volume of first box= "+b1.volume());
  System.out.println("Volume of second box= "+b2.volume());
  System.out.println("Volume of cube is= "+b3.volume());
}
}
```

Java Constructor	Java Method
Constructor is used to initialize the state of an object.	Method is used to expose behavior of an object.
Constructor must not have return type.	Method must have return type.
Constructor is invoked implicitly.	Method is invoked explicitly.
The java compiler provides a default constructor if you don't have any constructor.	Method is not provided by compiler in any case.
Constructor name must be same as the class name.	Method name may or may not be same as class name.

Overloading Methods

- Two or more methods within the same class that share same name.
- Parameter declaration must be different
- Used to implement polymorphism

Note: - Java uses the type/number of parameters to determine which method to execute

Argument lists could differ in –

1. Number of parameters.
2. Data type of parameters.
3. Sequence of Data type of parameters.

Example 1: Overloading – Different Number of parameters in argument list

```
class Overload
{
 void test()
 {
  System.out.println("No Parameters!");
 }
 void test(int a)
 {
  System.out.println("Parameter is integer"+a);
 }
 void test(int a,int b)
 {
  System.out.println("Parameters are integers  "+a+ " "+b);
 }
 }
class Overloadmain
{
 public static void main(String args[])
 {
   Overload o=new Overload();
   o.test();
   o.test(10);
   o.test(10,20);
       }  }
```

Example 2: Overloading – Difference in data type of arguments

In this example, method disp() is overloaded based on the data type of arguments – Like example 1 here also, we have two definition of method disp(), one with char argument and another with int argument.

```
class DisplayOverload
{
   public void disp(char c)
   {
      System.out.println(c);
   }
   public void disp(int c)
   {
     System.out.println(c );
   }
}
class Sample
{
   public static void main(String args[])
   {
      DisplayOverload obj = new DisplayOverload();
      obj.disp('a');
      obj.disp(5);
   }
}
```

Output:

a

5

Example3: Overloading – Sequence of data type of arguments

Here method disp() is overloaded based on sequence of data type of arguments – Both the methods have different sequence of data type in argument list. First method is having argument list as (char, int) and second is having (int, char). Since the sequence is different, the method can be overloaded without any issues.

```
class DisplayOverload
```

```
{
  public void disp(char c, int num)
  {
     System.out.println("I'm the first definition of method disp");
  }
  public void disp(int num, char c)
  {
     System.out.println("I'm the second definition of method disp" );
  }
}
class Sample3
{
  public static void main(String args[])
  {
     DisplayOverload obj = new DisplayOverload();
     obj.disp('x', 51 );
     obj.disp(52, 'y');
  }
}
```

Output:

I'm the first definition of method disp

I'm the second definition of method disp

this keyword

this keyword can be used to refer current class instance variable. If there is ambiguity between the instance variables and parameters, this keyword resolves the problem of ambiguity.

Understanding the problem without this keyword

```
class Student
{
int rollno;
String name;
float fee;
Student(int rollno,String name,float fee)
```

```
{
rollno=rollno;
name=name;
fee=fee;
}
void display()
{
System.out.println(rollno+" "+name+" "+fee);
}
}
class TestThis
{
public static void main(String args[]){
Student s1=new Student(111,"ankit",5000f);
Student s2=new Student(112,"sumit",6000f);
s1.display();
s2.display();
}}
```

Output:

0 null 0.0

0 null 0.0

In the above example, parameters (formal arguments) and instance variables are same. So, we are using this keyword to distinguish local variable and instance variable.

Solution of the above problem by this keyword

```
class Student
{
int rollno;
String name;
float fee;
Student(int rollno,String name,float fee)
{
this.rollno=rollno;
```

```
this.name=name;
this.fee=fee;
}
void display()
{
System.out.println(rollno+" "+name+" "+fee);
}
}

class TestThis
{
public static void main(String args[])
{
Student s1=new Student(111,"ankit",5000f);
Student s2=new Student(112,"sumit",6000f);
s1.display();
s2.display();
}}
```

Output

111 ankit 5000

112 sumit 6000

Instance variable hiding

- When a ***local variable/formal parameter*** has the same name as ***instance variable*** **,**Local variable/formal parameter ***hides*** the instance variable

```
class Box
{
  double width;
  double height;
  double depth;
Box(double width, double height, double depth)
 {
//width=width;     // not correct
  this.width=width;
```

```
    this.height=height;
    this.depth=depth;
  }
} // use this to overcome instance variable hiding
```

Garbage Collection

- Automatic deallocation is called garbage collection
- When no reference to an object exist it is considered to be no longer needed. Its memory can be destroyed
- Garbage collection occurs at irregular intervals; having no pattern or order in time

finalize() method

- Sometimes an object will need to perform some action when it is destroyed.
- If object holding non java resources like file handle, window character font etc.., make sure these resources are freed before an object is destroyed.
- To handle this situation java provides **finalization mechanism**
- You Can define specific actions that will occur when an object is about to be destroyed by GC
- Finalize is called just prior to GC.

finalize() method

```
protected void finalize()
  {
 // finalization code
 }
```

Protected prevents access to this code by outside class.

Access Controls Keywords

The access modifiers in java specifies accessibility (scope) of a data member, method, constructor or class.

There are 4 types of java access modifiers:

private

default

protected

public

private access modifier

The private access modifier is accessible only within class.

Simple example of private access modifier

In this example, we have created two classes A and Simple. A class contains private data member and private method. We are accessing these private members from outside the class, so there is compile time error.

```
class A
{
private int data=40;
private void msg()
{
System.out.println("Hello java");
}
}

public class Simple
{
 public static void main(String args[])
{
   A obj=new A();
   System.out.println(obj.data);//Compile Time Error
   obj.msg();//Compile Time Error
   }
}
```

default access modifier

If you don't use any modifier, it is treated as default by default. The default modifier is accessible only within package.

Example of default access modifier

In this example, we have created two packages pack and mypack. We are accessing the A class from outside its package, since A class is not public, so it cannot be accessed from outside the package.

```
//save by A.java
```

```
package pack;
class A
{
  void msg()
{
System.out.println("Hello");
}
}
```

```
//save by B.java
package mypack;
import pack.*;
class B
{
  public static void main(String args[])
{
   A obj = new A();//Compile Time Error
   obj.msg();//Compile Time Error
  }
}
```

protected access modifier

The protected access modifier is accessible within a class and outside the class but through inheritance only.

That is a variable or method is declared as protected , it can be accessed by that class and its derived class

```
public class A
{
protected void msg()
{System.out.println("Hello");
}
}
  class B extends A
{
  public static void main(String args[])
{
   B obj = new B();
```

```
    obj.msg();
   }
}
```

Output:Hello

4) public access modifier

The public access modifier is accessible everywhere. It has the widest scope among all other modifiers.

Example of public access modifier

```
public class A
{
public void msg()
{
System.out.println("Hello");
}
}

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

Output:Hello

Case study

Create two classes Employee and EmployeeTest. The Employee class has four instance variables - name, age, designation and salary. The class has one constructor, which takes parameters. And class contains a function print() that print all the data. Following is the *EmployeeTest* class, which creates two instances of the class Employee(object) and invokes the methods for each object to assign values for each variable.

```
import java.io.*;
public class Employee
 {
```

```
  String name;
  int age;
  String designation;
  double salary;
  // This is the constructor of the class Employee
 Employee(String n, int a, String d, double s)
{
     name=n;
    age=a;
    designation= d;
   salary=s;
  }
  /* Print the Employee details */
void print()
{
     System.out.println("Name:"+ name );
     System.out.println("Age:" + age );
     System.out.println("Designation:" + designation );
     System.out.println("Salary:" + salary);
  }
}
public class EmployeeTest
{
  public static void main(String args[])
 {
     /* Create two objects using constructor */
     Employee emp1 = new Employee("James Smith",25,Clerk, 10000);
     Employee emp2= new Employee("Mary" ,28, Manager,25000);

     // Invoking methods for each object created
   emp1.print();
emp2.print();
        }
}
```

Output

C:\> javac EmployeeTest.java

C:\> java EmployeeTest

Name:James Smith

Age:25

Designation:Clerk

Salary:10000.0

Name:Mary

Age:28

Designation:Manager

Salary:25000.0

Write a program to create a Bankacc class, its instance variables are accno,accname,balance. (Use a constructor to initialize the variables.) . If a bank holder deposits an amount it will add to his account and if uses withdraw operation, balance will be reduced.

```
class Bankacc
 {
String accno;
 String accname;
double balance;
 Bankacc(String accno1, String accname1, balance1)
{
        accno = accno1;
        accname = accname1;
       balance =balance1;
}
void deposit(double amount)
{
        if (amount > 0)
       {
        balance = balance + amount;
       System.out.println("Current balance=" + balance);
       }
       else
```

```
                System.out.println(“ Amount should greater than zero”);
        }
        void withdraw(double amount)
        {
                 if (amount>balance)
                {
                 balance = balance - amount;
                System.out.println(“Current balance=” + balance);
                }
                else
                System.out.println(“ Amount should greater than balance”);
        }
        }
Class BankaccDemo
{
public static void main(String args[])
{
 Bankacc a = new Bankacc("20120", "Abhinav ",5000);
 a.deposit(500);
 a.withdraw(400);
Bankacc b = new Bankacc("20121", "Anu ",8000);
b.deposit(600);
 b.withdraw(1000);
 }
}
```