Chap2:Classes,Objects and Methods

Java Classes

- A class in Java is a set of objects which shares common characteristics/ behavior and common properties/ attributes.
- It is a user-defined blueprint or prototype from which objects are created.
- For example, Student is a class while a particular student named Ravi is an object.

Properties of Java Classes

1. Class is not a real-world entity. It is just a template or blueprint or prototype from which objects are created.
2. Class does not occupy memory.
3. Class is a group of variables of different data types and a group of methods.
4. A Class in Java can contain:
   - Data member
   - Method
   - Constructor
   - Nested Class
   - Interface

Class Declaration in Java

access_modifier **class** <*class_name*>
{
    data member;
    method;
    constructor;
    nested class;
    interface;

}

Example of Java Class

```
 class Student {
   // data member (also instance variable)
   int id=1;
   // data member (also instance variable)
   String name=”Priya”;

   public static void main(String args[])
   {
      // creating an object of
      // Student
      Student s1 = new Student();
      System.out.println(s1.id);
      System.out.println(s1.name);
   }
}
```

Java Objects

- An object in Java is a basic unit of Object-Oriented Programming and represents real-life entities.
- Objects are the instances of a class that are created to use the attributes and methods of a class.
- A typical Java program creates many objects, which as you know, interact by invoking methods. An object consists of :

1. **State**: It is represented by attributes of an object. It also reflects the properties of an object.
2. **Behavior**: It is represented by the methods of an object. It also reflects the response of an object with other objects.

3. **Identity**: It gives a unique name to an object and enables one object to interact with other objects.
   - The new keyword is used to allocate memory at runtime. All objects get memory in Heap memory area.

```
1. class Rectangle{
2.  int length;
3.  int width;
4.  void insert(int l, int w){
5.   length=l;
6.   width=w;
7.  }
8.  void calculateArea(){System.out.println(length*width);}
9. }
10.       class TestRectangle1{
11.        public static void main(String args[]){
12.         Rectangle r1=new Rectangle();
13.         Rectangle r2=new Rectangle();
14.         r1.insert(11,5);
15.         r2.insert(3,15);
16.         r1.calculateArea();
17.         r2.calculateArea();
18.       }
19.
20.       }
```

O/p

```
55
45
```

Object referencing

- Object referencing in java is basically a address in memory where all methods and variables associated with object resides. When you create an object like this…
- Example a = new Example();
- here a is actually a reference which is pointing to memory assigned using new keyword.
- So if you do some operation with a for eg.
- a.x=5.(consider x is some variable).
- Now x associated with a has value 5.
- And if you declare another object..
- Example b= new Example();
- And simply do…b=a;
- Now b pointing to same memory address as a
- . So b.x will give you 5.This is called object refrencing.

What are Constructors in Java?

- In Java, Constructor is a block of codes similar to the method.
- It is called when an instance of the class is created.
- At the time of calling the constructor, memory for the object is allocated in the memory.
- It is a special type of method that is used to initialize the object.

- Every time an object is created using the new() keyword, at least one constructor is called.

Syntax

Following is the syntax of a constructor −

```
class ClassName {
  ClassName() {
  }
}
```

Java allows two types of constructors namely −

- Default Constructor in Java
- Parameterized Constructors

1) Default Constructor in Java

- A constructor that has no parameters is known as default the constructor.
- A default constructor is invisible.
- And if we write a constructor with no arguments, the compiler does not create a default constructor.
- It is taken out.
- It is being overloaded and called a parameterized constructor.
- The default constructor changed into the parameterized constructor. But Parameterized constructor can't change the default constructor. Example

```
Public class MyClass {
  Int num;
  MyClass() {
    num = 100;
  }
```

```
}
```

You would call constructor to initialize objects as follows

```
public class ConsDemo {
  public static void main(String args[]) {
    MyClass t1 = new MyClass();
    MyClass t2 = new MyClass();
    System.out.println(t1.num + " " + t2.num);
  }
}
```

This would produce the following result

100 100

Parameterized Constructors

- A constructor that has parameters is known as parameterized constructor.
- If we want to initialize fields of the class with our own values, then use a parameterized constructor.

Example

Here is a simple example that uses a constructor −

```
// A simple constructor.
class MyClass {
  int x;

  // Following is the constructor
  MyClass(int i ) {
    x = i;
  }
}
```

You would call constructor to initialize objects as follows −

```
public class ConsDemo {
  public static void main(String args[]) {
    MyClass t1 = new MyClass( 10 );
    MyClass t2 = new MyClass( 20 );
    System.out.println(t1.x + " " + t2.x);
  }
}
```

This would produce the following result −

```
10 20
```

Constructor Overloading in Java

- In Java, overloaded constructor is called based on the parameters specified when a new is executed.
- Sometimes there is a need of initializing an object in different ways. This can be done using constructor overloading.

Important points to be taken care of while doing Constructor Overloading

- Constructor calling must be the **first** statement of the constructor in Java.
- If we have defined any parameterized constructor, then the compiler will not create a default constructor. and vice versa if we don't define any constructor, the compiler creates the default constructor(also known as no-arg constructor) by default during compilation
- Recursive constructor calling is invalid in Java.

```
class Box {

    double width, height, depth;



    // constructor used when all dimensions specified
```

```
        Box(double w, double h, double d)
        {
            width = w;
            height = h;
            depth = d;
        }
        // constructor used when no dimensions specified
        Box() { width = height = depth = 0; }
        // constructor used when cube is created
        Box(double len) { width = height = depth = len; }
         // compute and return volume
        double volume() { return width * height * depth; }
    }
// Driver code
    public class Test {
        public static void main(String args[])
        {
            // create boxes using the various constructors
            Box mybox1 = new Box(10, 20, 15);
            Box mybox2 = new Box();
            Box mycube = new Box(7);
                double vol;
           // get volume of first box
            vol = mybox1.volume();
            System.out.println("Volume of mybox1 is " + vol);
```

```
                // get volume of second box
                vol = mybox2.volume();
                System.out.println("Volume of mybox2 is " + vol);
            // get volume of cube
                vol = mycube.volume();
                System.out.println("Volume of mycube is " + vol);
        }
    }
```

Op:Volume of mybox1 is 3000.0

Volume of mybox2 is 0.0

Volume of mycube is 343.0

Method Overloading:

- In Java, Method Overloading allows different methods to have the same name, but different signatures where the signature can differ by the number of input parameters or type of input parameters, or a mixture of both.
- Method overloading in Java is also known as ***Compile-time Polymorphism*, *Static Polymorphism, or Early binding***.
- In Method overloading compared to the parent argument, the child argument will get the highest priority.

```
// Java program to demonstrate working of method
// overloading in Java
public class Sum {
    public int sum(int x, int y) { return (x + y); }
        public int sum(int x, int y, int z)
```

```
    {
        return (x + y + z);
    }
    public double sum(double x, double y){return (x + y);}
    public static void main(String args[])
    {
        Sum s = new Sum();
        System.out.println(s.sum(10, 20));
        System.out.println(s.sum(10, 20, 30));
        System.out.println(s.sum(10.5, 20.5));
    }
}
```

Op

```
30
60
31.0
```

Different Ways of Method Overloading in Java

- Changing the Number of Parameters.
- Changing Data Types of the Arguments.
- Changing the Order of the Parameters of Methods

1. Changing the Number of Parameters

Method overloading can be achieved by changing the number of parameters while passing to different methods.

- // Importing required classes

```
import java.io.*;
class Product {
public int multiply(int a, int b)
        {       int prod = a * b;
   return prod;
        }
        public int multiply(int a, int b, int c)
        {
                int prod = a * b * c;
                return prod;
        }
}
class GFG {
                public static void main(String[] args)
        {
                Product ob = new Product();
                int prod1 = ob.multiply(1, 2);
System.out.println("Product of the two integer value :" + prod1);

                int prod2 = ob.multiply(1, 2, 3);
System.out.println("Product of the three integer value :" + prod2);
        }
}
```

O/P

- Product of the two integer value :2
- Product of the three integer value :6

2. Changing Data Types of the Arguments

In many cases, methods can be considered Overloaded if they have the same name but have different parameter types, methods are considered to be overloaded.

Below is the implementation of the above method:
Output

```
import java.io.*;

class Product {

        // Multiplying three integer values

        public int Prod(int a, int b, int c)

        {
```

```
            int prod1 = a * b * c;
            return prod1;
        }

        // Multiplying three double values.
        public double Prod(double a, double b, double c)
        {
            double prod2 = a * b * c;
            return prod2;
        }
}

class GFG {
        public static void main(String[] args)
        {
            Product obj = new Product();
        int prod1 = obj.Prod(1, 2, 3);
        System.out.println("Product of the three integer value :" + prod1);
        double prod2 = obj.Prod(1.0, 2.0, 3.0);
       System.out.println("Product of the three double value :" + prod2);
        }
}
```

op

```
Product of the three integer value :6
Product of the three double value :6.0
```

3. Changing the Order of the Parameters of Methods

Method overloading can also be implemented by rearranging the parameters of two or more overloaded methods.

For example, if the parameters of method 1 are (String name, int roll_no) and the other method is (int roll_no, String name) but both have the same name, then these 2 methods are considered to be overloaded with different sequences of parameters.

Below is the implementation of the above method
Output

```
// Importing required classes

import java.io.*;

class Student {

        // Method 1

        public void StudentId(String name, int roll_no)

        {

        System.out.println("Name :" + name + " "         + "Roll-No :" + roll_no);

        }

 // Method 2

        public void StudentId(int roll_no, String name)

        {

                // Again printing name and id of person

        System.out.println("Roll-No :" + roll_no + " "+ "Name :" + name);

        }

}

class GFG {

        // Main function
```

```
    public static void main(String[] args)
    {
        // Creating object of above class
        Student obj = new Student();

        // Passing name and id
        // Note: Reversing order
        obj.StudentId("riya", 1);
        obj.StudentId(2, "Kamlesh");
    }
}
```

op

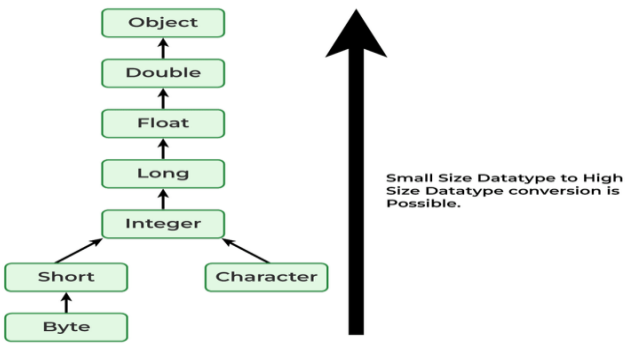
Name : riya Roll-No :1

Roll-No :2 Name :Kamlesh

What if the exact prototype does not match with arguments?

Priority-wise, the compiler takes these steps:

- Type Conversion but to a higher type(in terms of range) in the same family.
- Type conversion to the next higher family(suppose if there is no long data type available for an int data type, then it will search for the float data type).



Let's take an example to clarify the concept:

```
// Demo Class
class Demo {
	public void show(int x)
	{
		System.out.println("In int" + x);
	}
	public void show(String s)
	{
		System.out.println("In String" + s);
	}
	public void show(byte b)
	{
		System.out.println("In byte" + b);
	}
}

class UseDemo {
	public static void main(String[] args)
	{
		byte a = 25;
		Demo obj = new Demo();
		obj.show(a);
		obj.show("hello");
		obj.show(250);
```

// since float datatype is not available and so it's higher datatype, so at this step their will be an error.

```
            obj.show(7.5);

    }

}
```

Output
1 error

finalize() Method in Java

- finalize() method in Java is a method of the Object class that is used to perform cleanup activity before destroying any object.
- It is called by Garbage collector before destroying the objects from memory.
- finalize() method is called by default for every object before its deletion.
- This method helps Garbage Collector to close all the resources used by the object and helps JVM in-memory optimization.
- Ex

```
public class Example

{

   public static void main(String[] args)

   {

      Example ex = new Example(); // Creating object ex of class Example

      ex = null; // Unrefrencing the object ex.

      System.gc(); // Calling garbage collector to destroy ex

      System.out.println("Unreferenced object ex is destroyed successfully!");

   }

   @Override

   protected void finalize()

   {

      System.out.println("Inside finalize method");
```

```
        System.out.println("Release and close connections.");

    }  }
```

Op:

```
Unreferenced object ex is destroyed successfully!
Inside finalize method.
Release and close connections
```

Recursion in Java

- In Java, Recursion is a process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called a recursive function.
- Using a recursive algorithm, certain problems can be solved quite easily.

Example:

```
// Java Program to implement
// Factorial using recursion
class GFG {

        // recursive method
        int fact(int n)
        {
                int result;

                if (n == 1)
                        return 1;
                result = fact(n - 1) * n;
                return result;
        }
}

// Driver Class
class Recursion {

        // Main function
        public static void main(String[] args)
        {
```

```
        GFG f = new GFG();

        System.out.println("Factorial of 3 is " + f.fact(3));
        System.out.println("Factorial of 4 is "+ f.fact(4));
        System.out.println("Factorial of 5 is "+ f.fact(5));
        }
}
```

Op:

```
Factorial of 3 is 6
Factorial of 4 is 24
Factorial of 5 is 120
```

Java Object as Parameter

- Objects, like primitive types, can be passed as parameters to methods in Java.
- When passing an object as a parameter to a method, a reference to the object is passed rather than a copy of the object itself.
- This means that any modifications made to the object within the method will have an impact on the original object.
- Example:
- class Add {
- int a;
- int b;
-
- Add(int x, int y) // parametrized constructor
- {
- a = x;
- b = y;
- }
- void sum(Add A1) // object 'A1' passed as parameter in function 'sum'
- {
- int sum1 = A1.a + A1.b;
- System.out.println("Sum of a and b :" + sum1);

```
  }
}

public class Main {
  public static void main(String arg[]) {
    Add A = new Add(5, 8);
    /* Calls  the parametrized constructor
    with set of parameters*/
    A.sum(A);
  }
}
```

Output:
Sum of a and b :13

Returning Objects

In java, a method can return any type of data, including objects.

Example

```
import java.util.Scanner;

class TwoNum {
  private int a, b;
  Scanner kb = new Scanner(System.in);

  void getValues()
  {
    System.out.print("Enter a: ");
    a = kb.nextInt();
    System.out.print("Enter b: ");
    b = kb.nextInt();
  }

  void putValues()  {
    System.out.println(a + " " + b);
  }

  TwoNum add(TwoNum B) /*class type function add() takeobject 'B' as parameter*/ {
    TwoNum D = new TwoNum(); //object D act as instance variable
    D.a = a + B.a;
    D.b = b + B.b;
    return (D); //returning object D
  }
}

public class Main {
  public static void main(String arg[]) {
    TwoNum A = new TwoNum();
    A.getValues();
    A.putValues();

    TwoNum B = new TwoNum();
    B.getValues();
    B.putValues();
```

```
    TwoNum C;
    /*object A calls add() passing object B
    as parameter and result are return at C*/
    C = A.add(B);

    C.putValues();
  }
}
```

New Operator in Java:

The new operator in java is used to create new objects of a class. A request will be sent to the Heap Memory for object creation. If enough memory is available, the operator new initializes the memory and returns the address of the newly allocated and initialized memory to a the object variable.

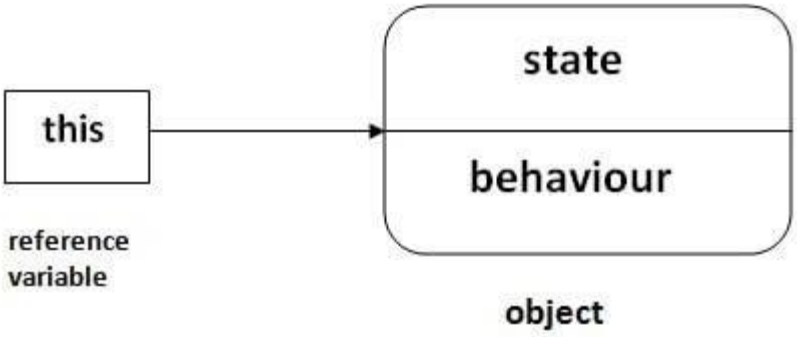
Syntax

```
1. NewExample obj=new NewExample();
```

```
2.  public class NewExample1 {
3.
4.      void display()
5.      {
6.          System.out.println("Invoking Method");
7.      }
8.
9.      public static void main(String[] args) {
10.         NewExample1 obj=new NewExample1();
11.         obj.display();
12.     }
13.
14. }
```

this keyword in Java

There can be a lot of usage of **Java this keyword**. In Java, this is a **reference variable** that refers to the current object.



Usage of Java this keyword

Here is given the 6 usage of java this keyword.

1. this can be used to refer current class instance variable.
2. this can be used to invoke current class method (implicitly)
3. this() can be used to invoke current class constructor.
4. this can be passed as an argument in the method call.
5. this can be passed as argument in the constructor call.
6. this can be used to return the current class instance from the method.

Java static keyword

- The **static keyword** in Java is used for memory management mainly.
- We can apply static keyword with variables, methods, blocks and nested classes.
- The static keyword belongs to the class than an instance of the class.

The static can be:

1. Variable (also known as a class variable)
2. Method (also known as a class method)
3. Block
4. Nested class

1) Java static variable

If you declare any variable as static, it is known as a static variable.

- The static variable can be used to refer to the common property of all objects (which is not unique for each object), for example, the company name of employees, college name of students, etc.
- The static variable gets memory only once in the class area at the time of class loading.

Advantages of static variable

It makes your program **memory efficient** (i.e., it saves memory).

```
1. /Java Program to demonstrate the use of static variable
2. class Student{
3.    int rollno;//instance variable
4.    String name;
5.    static String college ="ITS";//static variable
6.    //constructor
7.    Student(int r, String n){
8.    rollno = r;
9.    name = n;
10.   }
11.   //method to display the values
12.   void display (){System.out.println(rollno+" "+name+" "+college);}
13.}
14.//Test class to show the values of objects
15.public class TestStaticVariable1{
16. public static void main(String args[]){
17. Student s1 = new Student(111,"Karan");
18. Student s2 = new Student(222,"Aryan");
19. //we can change the college of all objects by the single line of code
20. //Student.college="BBDIT";
21. s1.display();
22. s2.display();
23. }
24.}
```

Output:

```
111 Karan ITS
222 Aryan ITS
```

2) Java static method

If you apply static keyword with any method, it is known as static method.

- A static method belongs to the class rather than the object of a class.
- A static method can be invoked without the need for creating an instance of a class.
- A static method can access static data member and can change the value of it.

```
class Calculate{
  static int cube(int x){
  return x*x*x;
  }

  public static void main(String args[]){
  int result=Calculate.cube(5);
  System.out.println(result);
  }
}
```

Restrictions for the static method

There are two main restrictions for the static method. They are:

1. The static method can not use non static data member or call non-static method directly.
2. this and super cannot be used in static context.

Java static block

- Is used to initialize the static data member.
- It is executed before the main method at the time of classloading.

Example of static block

```
1. class A2{
2.   static{System.out.println("static block is invoked");}
3.   public static void main(String args[]){
4.    System.out.println("Hello main");
5.   }
6. }
```

```
Output:static block is invoked
       Hello main
```

Java Nested and Inner Class

In Java, a class can be defined within another class and such classes are known as nested classes. These classes help you to logically group classes that are only used in one place. This increases the use of encapsulation and creates a more readable and maintainable code.

Nested Class in Java

The class written within a class is called the nested class while the class that holds the inner class is called the outer class. Below are some points to remember for nested classes in Java -

- The scope of a nested class is bounded by its enclosing class.
- A nested class has access to the members of the class in which it is nested. But, the enclosing class cannot access the members of the nested class.
- A nested class is its enclosing class member.

- A nested class can be declared public, private, protected, or package-private.

Types of nested classes

Inner/Non-static nested class: In Java, non-static classes are a security mechanism. A class cannot be associated with the access modifier private, but if you have the class as a member of other class, then the non-static class can be made private.

Types of inner classes –

- Inner Class
- Anonymous Inner Class

Inner Class

To create an inner class you just need to write a class within a class. An inner class can be private which cannot be accessed from an object outside the class. Below is a program to create an inner class. In this example, the inner class is made private and is accessed class through a method.

```
lass TestMemberOuter1{

 private int data=30;

 class Inner{

  void msg(){System.out.println("data is "+data);}

 }
```

```
 void display(){
  Inner in=new Inner();
  in.msg();
 }
 public static void main(String args[]){
  TestMemberOuter1 obj=new TestMemberOuter1();
  obj.display();
 }
}
```

Anonymous Inner Class

Anonymous inner class is an inner class declared without a class name. In an anonymous inner class, we declare and instantiate it at the same time. They are generally used when you need to override the method of a class or an interface. The below program shows how to use an anonymous inner class -

```
interface Eatable{

 void eat();

}class TestAnnonymousInner1{

 public static void main(String args[]){

      Eatable e=new Eatable(){

  public void eat(){System.out.println("nice fruits");}

 };

 e.eat();

 }

}
```

Static nested class:

A static class is a nested class that is a static member of the outer class. Unlike inner class, the static nested class cannot access member variables of the outer class because the static nested class doesn't require an instance of the outer class. Hence, there is no reference to the outer class with OuterClass.this. The syntax of a static nested class is –

```
1. class TestOuter1{
2.   static int data=30;
3.   static class Inner{
```

```
4.     void msg(){System.out.println("data is "+data);}
5.    }
6.    public static void main(String args[]){
7.    TestOuter1.Inner obj=new TestOuter1.Inner();
8.    obj.msg();
9.    }
10. }
```

Output:

```
data is 30
```