Introduction to Object Oriented Programming in Java

Notes by Adil Aslam

Programming Techniques

- **a) Unstructured Programming**
  - (Assembly language programming)
- **b) Procedural Programming**
  - (Assembly language, C programming)
- **c) Object Oriented Programming**
  - (C++, Java, Smalltalk, C#, Objective C)

Procedure Oriented Programming

- It means "***a set of procedures***" which is a "***set of subroutines***" or a "***set of functions***".
- functions are called repeatedly in a program to execute tasks performed by them. For **example**, a program may involve collecting data from user (reading), performing some kind of calculations on the collected data (calculation), and finally displaying the result to the user when requested (printing). All the 3 tasks of reading, calculating and printing can be written in a program with the help of 3 different functions which performs these 3 different tasks.

A Real-World Example

- Let's say that you are working for a vehicle parts manufacturer that needs to update it's online inventory system. Your boss tells you to program two similar but separate forms for a website, one form that processes information about cars and one that does the same for trucks.
- **For cars,** we will need to record the following information:
  - Color, Engine Size, Transmission Type, **Number of doors**
- **For trucks**, the information will be similar, but slightly different. We need:
  - Color, Engine Size, Transmission Type, **Cab Size, Towing Capacity**

Scenario 1

- Suppose that we suddenly need to add a bus form, that records the following information:

  Color, Engine Size, Transmission Type,

  Number of passengers

- ***Procedural:*** We need to recreate the entire form, repeating the code for Color, Engine Size, and Transmission Type.
- ***OOP:*** We simply extend the vehicle class with a bus class and add the method,numberOfPassengers.

Scenario 2

- Instead of storing color in a database like we previously did, for some strange reason our client wants the color emailed to him.

- ***Procedural:*** We change three different forms: cars, trucks, and buses to email the color to the client rather than storing it in the database.
- ***OOP****:* We change the color method in the vehicle class and because the car, truck, and bus classes all extend (or inherit from, to put it another way) the vehicle class, they are automatically updated.

Scenario 3

- We want to move from a generic car to specific makes, for example: Nissan and Mazda.

- ***Procedural:*** We create a new form for each make, repeating all of the code for generic car information and adding the code specific to each make.

- ***OOP:*** We extend the car class with a Nissan class and a Mazda class and add methods for each set of unique information for that car make.

Procedural vs. Object-Oriented Programming

POP	OOP
In POP, program is divided into small parts called **functions**.	In OOP, program is divided into parts called **objects**.
POP does not have any proper way for hiding data so it is **less secure**.	OOP provides Data Hiding so provides **more security**.
Example of POP are : C, VB, FORTRAN, Pascal.	Example of OOP are : C++, JAVA, VB.NET, C#.NET.

OOP

- **Object Oriented Methodology** is a certain **process** through which software can be developed.
- The **goals** of this methodology are to achieve Software Systems that are **reliable**, **reusable**, **extensible**; hence, more useful in the long run.
- The methodology achieves its goals by the help of a *collection of objects that communicate by exchanging messages*.

OOP Features

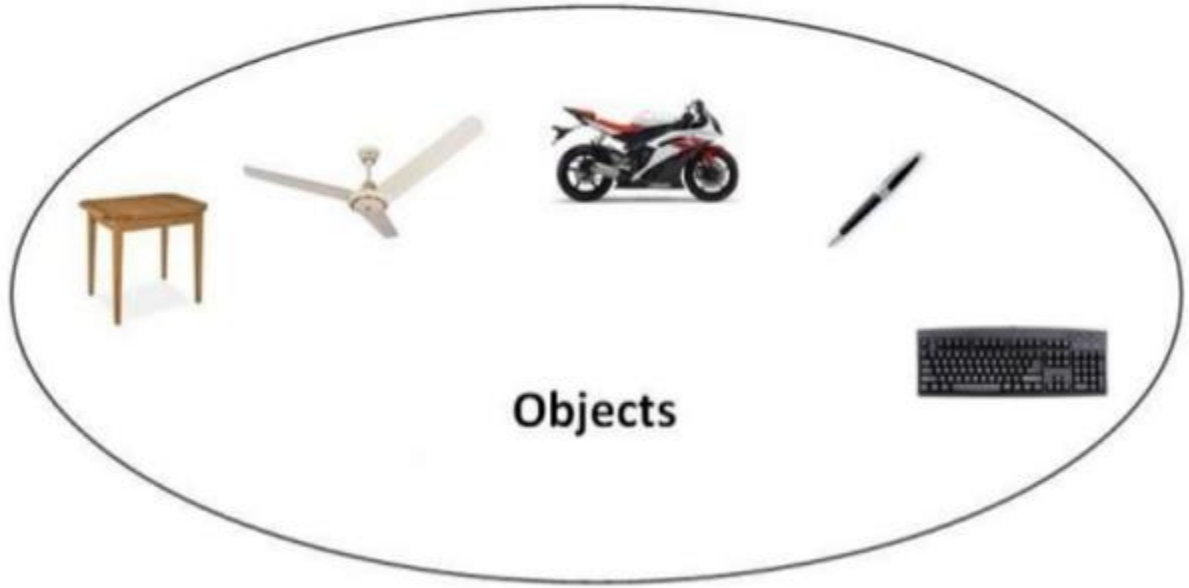
- Java is an Object-Oriented Language. As a language that has the Object-Oriented feature, Java supports the following fundamental concepts –
  - Classes
  - Objects
  - Instance
  - Method
  - Message Parsing
  - Polymorphism
  - Inheritance
  - Encapsulation
  - Abstraction

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.

Object in Java

- 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.



Example of Object

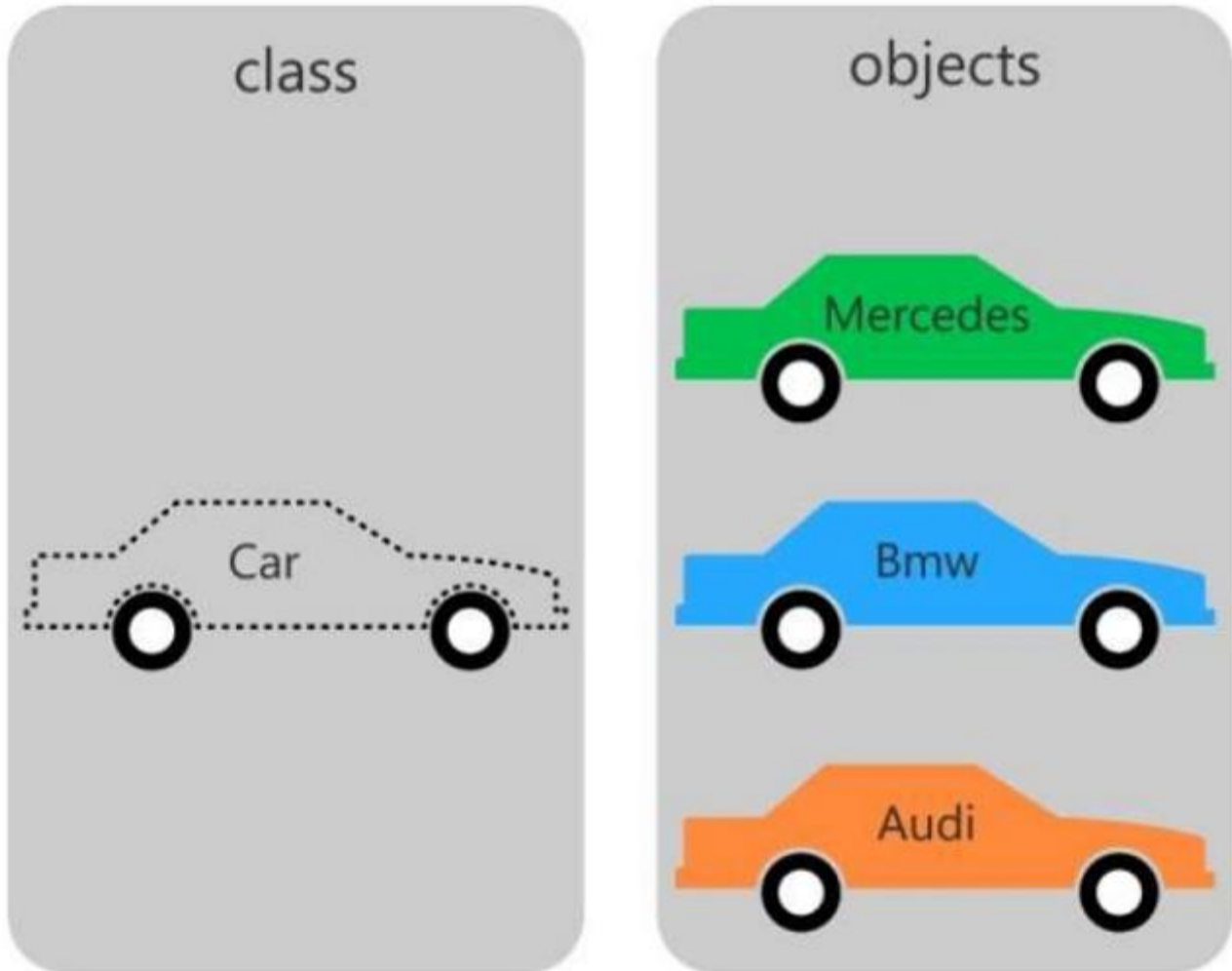
- **Examples-1:**
  Object: House
  State: Current Location, Color, Area of House etc
  Behavior: Close/Open main door.

- **Examples-2:**
  Object: – Car
  State: Color, Make
  Behavior: Climb Uphill, Accelerate, SlowDown etc

Class in Java

- A class is a group of objects that has common properties. It is a template or blueprint from which objects are created.
- A class in java can contain:
  - **Data Member**
  - **Method**
  - **Constructor**
  - **Block**
  - **Class and Interface**

Object and Class in Java



Ways to Create Objects in Java?

There are five different ways to create objects in java:

- Using new keyword
- Using Class.forName():
- Using clone():
- Using Object Deserialization:
- Using newIntance() method

Creating an Object (using new keyword)

- As mentioned previously, a class provides the blueprints for objects. So basically, an object is created from a class. In Java, the new keyword is used to create new objects.
- There are three steps when creating an object from a class :
  - **Declaration** – A variable declaration with a variable name with an object type.
  - **Instantiation** – The 'new' keyword is used to create the object.
  - **Initialization** – The 'new' keyword is followed by a call to a constructor. This call initializes the new object.

Ways to Create Objects in Java?

- **Using new keyword:**
- This is the most common way to create an object in java. Almost 99% of objects are created in this way.

- **Syntax:**

```
class_name object_name = new class_name();
```

- **Example**

Here Rectangle show default constructor

```
Rectangle obj = new Rectangle();
```

Variable Types

- Local variables
- Instance variables
- Class/Static variables

Variable Types

- **Local Variables**
  - Local variables are declared in methods, constructors, or blocks.
  - Local variables are created when the method, constructor or block is entered and the variable will be destroyed once it exits the method, constructor, or block.
  - Access modifiers cannot be used for local variables.
  - Local variables are visible only within the declared method, constructor, or block.
  - Local variables are implemented at stack level internally.
  - There is no default value for local variables, so local variables should be declared and an initial value should be assigned before the first use.

Variable Types

- Here's a method that declares a local variable named i, and then initializes the variable before using it:

```
public static void main(String[] args)
   {
      int i;   //declare local variable
      i = 0;  //initializing the local variable
      System.out.println("i is " + i);
   }
```

OR

```
public static void main(String[] args)
   {
    int i=0;//declare and initializing local variable in single line
     System.out.println("i is " + i);
   }
```

Instance Variables-1

- Instance variables are declared in a class, but outside a method, constructor or any block.
- When a space is allocated for an object in the heap, a slot for each instance variable value is created.
- Instance variables are created when an object is created with the use of the keyword 'new' and destroyed when the object is destroyed.
- Instance variables hold values that must be referenced by more than one method, constructor or block, or essential parts of an object's state that must be present throughout the class.
- Instance variables can be declared in class level before or after use.

Instance Variables-2

- Access modifiers can be given for instance variables.
- The instance variables are visible for all methods, constructors and block in the class. Normally, it is recommended to make these variables private (access level). However, visibility for subclasses can be given for these variables with the use of access modifiers.
- Instance variables have default values. For numbers, the default value is 0, for Booleans it is false, and for object references it is null. Values can be assigned during the declaration or within the constructor.
- Instance variables can be accessed directly by calling the variable name inside the class.

Accessing Instance Variables and Methods

- Instance variables and methods are accessed via created objects. To access an instance variable, following is the fully qualified path –

```
/* First create an object */
Rectangle ObjectReference = new Rectangle();

/* Now call a variable as follows */
ObjectReference.variableName;

/* Now you can call a class method as follows */
ObjectReference.MethodName(ParameterList);
```

Accessing Instance Variables and Methods

```
public class Example {
   int myCount = 0;
   void increment ( ) {
      myCount = myCount + 1;  }
    void print ( ) {
      System.out.println ("count = " + myCount);  }
   public static void main(String[] args) {
    Example c1 = new Example ( );
      c1.increment ( );  // c1's myCount is now 1
      c1.increment ( );  // c1's myCount is now 2
      c1.print();
      c1.myCount = 0;    // effectively resets the c1 counter
      c1.print();
   }
}
```

Instance Variables

```
class Rectangle {
 //instance variables
  double length;
  double breadth;
// This class declares an object of type Rectangle.
  public static void main(String args[]) {
    Example myrect = new Example();
    double area;
    // assign values to myrect1's instance variables
    myrect.length = 10;
    myrect.breadth = 10;
    // Compute Area of Rectangle
    area =  myrect.length * myrect.breadth ;
    System.out.println("Area of Rectangle  : " + area);
  }
}
```

Class/Static Variables-1

- Class variables also known as static variables are declared with the static keyword in a class, but outside a method, constructor or a block.
- There would only be one copy of each class variable per class, regardless of how many objects are created from it.
- Static variables are rarely used other than being declared as constants. Constants are variables that are declared as public/private, final, and static. Constant variables never change from their initial value.
- Static variables are stored in the static memory. It is rare to use static variables other than declared final and used as either public or private constants.

Class/Static Variables-2

- Static variables are created when the program starts and destroyed when the program stops.
- Visibility is similar to instance variables. However, most static variables are declared public since they must be available for users of the class.
- Default values are same as instance variables. For numbers, the default value is 0; for Booleans, it is false; and for object references, it is null. Values can be assigned during the declaration or within the constructor. Additionally, values can be assigned in special static initializer blocks.
- Static variables can be accessed by calling with the class name*ClassName.VariableName*.

Static Variables Example

```
public class Employee {
   // salary  variable is a private static variable
   private static double salary;
   // DEPARTMENT is a constant
   public static final String DEPARTMENT = "Development ";
   public static void main(String args[]) {
      salary = 1000;
      System.out.println(DEPARTMENT + "average salary:" + salary);
   }
}
```

Note: If the Variables are Accessed from an Outside class, the Constant should be Accessed as Employee.DEPARTMENT

Characteristic	Local variable	Instance variable	Class variable
Where declared	In a method, constructor, or block.	In a class, but outside a method. Typically private.	In a class, but outside a method. Must be declared static. Typically also final.
Use	Local variables hold values used in computations in a method.	Instance variables hold values that must be referenced by more than one method	Class variables are mostly used for *constants*, variables that never change from their initial value
Lifetime	Created when method or constructor is entered. Destroyed on exit.	Created when instance of class is created with new. Destroyed when there are no more references to enclosing object (made available for	Created when the program starts. Destroyed when the program stops.

Characteristic	Local variable	Instance variable	Class variable
Declaration	Declare before use anywhere in a method or block.	Declare anywhere at class level (before or after use).	Declare anywhere at class level with static.
Initial value	None. Must be assigned a value before the first use.	Zero for numbers, false for booleans, or null for object references. May be assigned value at declaration or in constructor.	Same as instance variable, and it addition can be assigned value in special static *initializer block*.
Name syntax	Standard rules.	Standard rules, but are often prefixed to clarify difference from local variables, eg withmy, m, or m_ (for member) as in myLength, or this as inthis.length.	static public final variables (constants) are all uppercase, otherwise normal naming conventions.

Characteristic	Local variable	Instance variable	Class variable
Access from outside	Impossible. Local variable names are known only within the method.	Instance variables should be declared private to promote information hiding, so should not be accessed from outside a class. However, in the few cases where there are accessed from outside the class, they must be qualified by an object (eg, myPoint.x).	Class variables are qualified with the class name (e.g. Color. BLUE). They can also be qualified with an object, but this is a deceptive style.

Example to Understand the Types of Variables in Java

```
class A
{
    int data=50;   //instance variable
    static int m=100;  //static variable
    void method()
    {
        int n=90;  //local variable
    } //end of method
} //end of class
```

Constructor in Java

Constructor in Java

- **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.

- **Rules for Creating Java Constructor**
- There are basically two rules defined for the constructor.
  - Constructor name must be same as its class name
  - Constructor must have no explicit return type

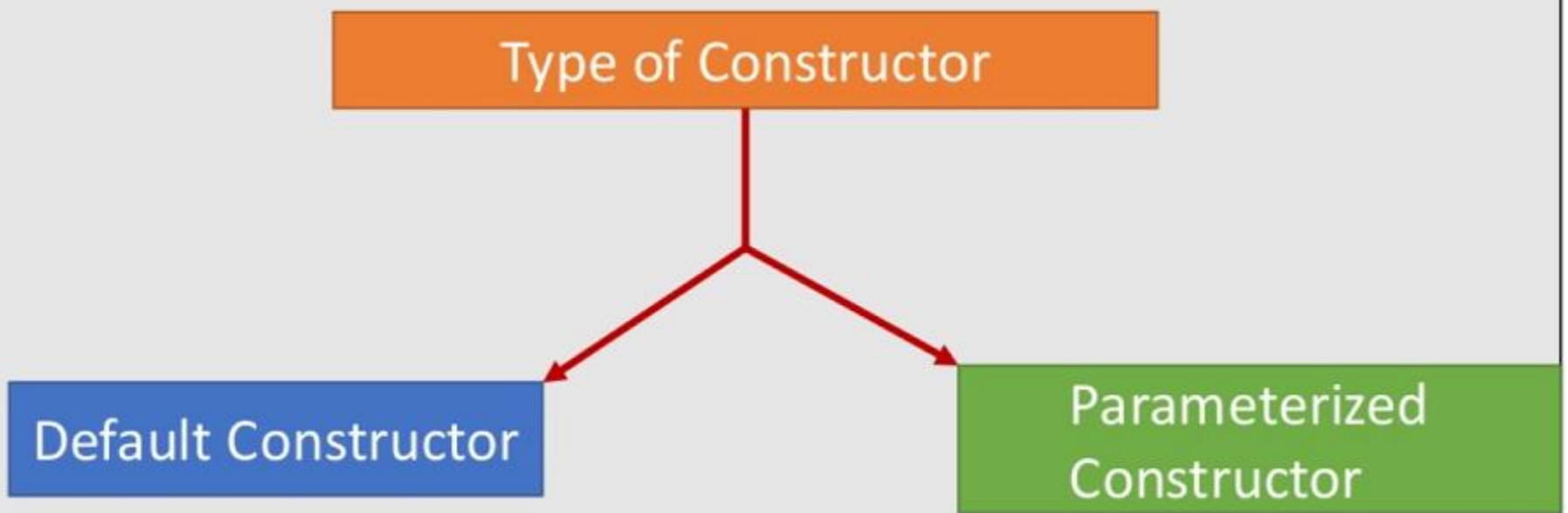
Constructor in Java

- **Some Rules of Using Constructor :**

  - Constructor **Initializes an Object**.
  - Constructor **cannot be called** like methods.
  - Constructors **are called automatically** as soon as object gets created.
  - Constructor **don't have any return Type.** (even Void)
  - Constructor name is same as that of "**Class Name**".
  - Constructor **can accept parameter**.
  - Default constructor **automatically called** when object is created.

Constructor in Java

- **Types of Java Constructors**
- There are two Types of Constructors:
  - Default constructor (no-arg constructor)
  - Parameterized constructor



Constructor in Java

- **Java Default Constructor:**
- A constructor that have no parameter is known as default constructor.
- **Syntax of Default Constructor:**

```
class_name()
{
}
```

- **Example**

```
Rectangle()
{
}
```

Example of Default Constructor

```
public class Rectangle {
   Rectangle()
    {
     System.out.println("Rectangle is created");
    }
    public static void main(String[] args) {
      //Creating new object rec below
       Rectangle rec=new Rectangle();
    }

}
```

Default Constructor

- If there is no constructor in a class, compiler automatically creates a default constructor.
- **What is the Purpose of Default Constructor?**
  - Default constructor provides the default values to the object like 0, null etc. depending on the type.

Example of Default Constructor that Displays the Default Values

```
public class Example{
  int id;
  String name;
  void display() {
    System.out.println(id+“  "+name); }

public static void main(String args[]){
  Example s1=new Example();
  Example s2=new Example();
  s1.display();
  s2.display();
 }
}
```

Constructor in Java

- **Java Parameterized Constructor**
- A constructor that have parameters is known as parameterized constructor.
- **Why use Parameterized Constructor?**
  - Parameterized constructor is used to provide different values to the distinct objects.
- **Syntax of Parameterized Constructor**

```
class_name(parameter_list)
{
}
```

- **Example**

```
Rectangle (int len , int bre)
{
```

Constructor in Java

- **Java Parameterized Constructor**

  - Constructor Can Take Value , Value is Called as – "**Argument**".

  - Argument **can be of any type** i.e Integer, Character, Array or any Object.

  - Constructor **can take any number of Argument**.

Example of Parameterized Constructor

```
class Rectangle {
  int length;
  int breadth;
  Rectangle(int len , int bre)
  {
  length  = len;
  breadth = bre;
  }
  public static void main(String args[]) {
  Rectangle r1 = new Rectangle(20,10);
  System.out.println("Length of Rectangle: " + r1.length);
  System.out.println("Breadth of Rectangle: "+ r1.breadth);
  }
}
```

Point to be Note

```
class Student{
    int var=5;
    public Student(int num) {
        var=num;
    }
    public int getValue()  {
        return var;
    }
  public static void main(String args[]){
  Student ob = new Student();
  System.out.println("value is: "+ob.getValue());
 }
```

Output: It will throw a **compilation error!!.**
The reason is when we don't define any constructor in our class, compiler defines default one for us, however when we declare any constructor (in this example we have already defined a parameterized constructor), compiler doesn't do it for us. Since we have defined a constructor in this code, compiler didn't create default one. While creating object we am invoking default one, which doesn't exist in this code. The code gives an compilation error.

If we remove the parametrized constructor from the code above then the code would run fine because, java would generate the default constructor if it doesn't find any in the code.

Difference Between Constructor and Method

Java Constructor	Java Method
Constructor is used to initialize the state of an object.	Method is used to expose behaviour 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.

Constructor Overloading

- Like methods, a constructor can also be overloaded.
- Overloaded constructors are differentiated on the basis of their type of parameters or number of parameters.
- Constructor overloading is not much different than method overloading.
- In case of method overloading we have multiple methods with same name but different signature, whereas in Constructor overloading we have multiple constructor with different signature but only difference is that Constructor doesn't have return type in Java.
- **Why do we Overload constructors ?**
  - Constructor overloading is done to construct object in different ways.

Constructor Overloading

- **Why Constructor Overloading is Required in Java?**
- Constructor provides a way to create objects implicitly of any class using '***new***' keyword
- So, overloaded constructor serves many ways to create distinct objects using different types of data of same class
- One classical example to discuss about constructor overloading is '***StringBuffer***' class from '*java.lang*' package
- StringBuffer class has four overloaded constructors
- *StringBuffer(String str)* is one of the parametrized constructor which has a initial capacity of 16 plus length of the String supplied
- We can use this constructor if we have initial string value to supply
- Or else, if we don't have any idea about initial String to specify then simply use 1st overloaded constructor which has no argument (default constructor)

Constructor Overloading

- **Different Ways to Overload Constructor in Java, by Changing**
  - Number of input parameters
  - Data-type of input parameters
  - Order/sequence of input parameters, if they are of different data-types
- **Constructor Signature**
- Constructor signature consists of
  - Name of the constructor which should be same as that of class name
  - number of input parameters
  - their data types
  - access modifiers like private, default, protected or public
- Access modifiers are not valid to consider in constructor overloading concept and in fact compiler throws exception if we overload constructor just by changing access modifiers keeping other things in constructor signature same

Constructor Overloading

- **Things to Remember About Constructor Overloading (Constructor Signature)**
  - Compiler checks 3 things when we overload constructor
  1. constructor name (should be same as that of class name)
  2. number of input parameters and
  3. data-type of input parameters
- –> Combination of number of input parameters & their data-type has to be different for successful compilation
- –> Or else compiler throws duplicate error
- **Note:** If name of the constructor differs and doesn't have any return type then compiler threat this as method and throws compile time error

Syntax of Overloaded Constructor

```
class ClassName {
      // Variable declaration
      // Constructor
      ClassName() {
         // body of constructor
      }

      ClassName(Parameter_List) {
         // body of constructor
      }
      :
      :
      ClassName(Parameter_List) {
         // body of constructor
      }
      // Methods
}
```

Example of Constructor Overloading-1

Example on Constructor Overloading
Based on Number of input Parameters

```
class Student{
   int id;
   String name;
   int age;
   Student(int i, String n) {
   id = i;
   name = n;
   }
   Student(int i, String n, int a){
   id = i;
   name = n;
   age=a;
   }
```

Example of Constructor Overloading-1

Example on Constructor Overloading Based on data types of input Parameters

```
public class Employee {
   // member variables
   int employeeId;
   String employeeName;
   // parametrized constructor 1 (String, int)
   Employee(String name, int id) {
      this.employeeId = id;
      this.employeeName = name;
   }
   // parametrized constructor 2 (int, String)
   Employee(int id, String name) {
      this.employeeId = id;
      this.employeeName = name;
   }
```

Example of Constructor Overloading-2

```
// display() method
   void displayEmployeeInfo() {
      System.out.println("Employee details\nId: " + employeeId
      + "\t Name: " + employeeName + "\n");
   }
   // main() method - entry point to JVM
   public static void main(String args[]) {
      Employee emp1 = new Employee("Adil", 23);
      emp1.displayEmployeeInfo();

      Employee emp2 = new Employee(19, "Kashif");
      emp2.displayEmployeeInfo();
   }
}
```

Default Constructor v/s Parametrized Constructor

Sr. No.	Default Constructor	Parametrized Constructor
1	A constructor which takes no arguments is known as default constructor	A constructor which takes one or more arguments is known as parametrized constructor
2	Compiler inserts a default no-arg constructor after compilation, if there is no explicit constructor defined in class	When parametrized constructor are defined in class, then programmer needs to define default no-arg constructor explicitly if required
3	No need to pass any parameters while constructing new objects using default constructor	At least one or more parameters needs to be passed while constructing new objects using argument constructors
4	Default constructor is used to initialize objects with same data	Whereas parametrized constructor are used to create distinct objects with different data

```java
public class Dice {

// Instance variable (data type properties) definitions:
   private int faces; // fixed number of faces of this dice
   private int value; // current top face value of this dice

// Constructor method (data type value builder) definitions:
   /** Construct a n-sided dice with faces numbered from 1 though n, for n >= 2. */
   public Dice(int n) {
      // throw an illegal argument exception if n < 2
      if (n < 2) throw new IllegalArgumentException("Error n < 2");
      this.faces = n;
      this.value = (int)(Math.random() * this.faces) + 1;
   }

   /** Construct a 6-sided dice by invoking the above constructor.  */
   public Dice() {
      this(6); // invoke the above Dice constructor with 6 as parameter
   }

// Instance method (data type operations) definitions:
   /** Return the number of faces on this dice. */
   public int faces() {
      return this.faces;
   }

   /** Return current top face value of this dice. */
   public int value() {
      return this.value;
   }

   /** Simulate rolling this dice by randomly changing its top face value. */
   public void roll() {
      this.value = (int)(Math.random() * this.faces) + 1;
   }

   /** Return a String representation of this dice. */
   @Override public String toString() {
      return "[" + super.toString()   // invoke the inherited toString method
             + " | faces = " + this.faces
             + ", value = " + this.value
             + "]";
   }

} // end Dice class
```

```java
/*
 * Compilation:  javac DiceClientV1.java
 * Dependencies: Dice.java
 * Execution:    java DiceClientV1
 *
 * A test client for the Dice data type.
 */

public class DiceClientV1 {
   public static void main(String[] args) {
      Dice dice1; // declare a name dice1 which can refer to a Dice object
      Dice dice2;
      Dice dice3;

      dice1 = new Dice();    // create a 6-sided Dice object & name it dice1
      dice2 = new Dice(10);  // create a 10-sided Dice object & name it dice2
      dice3 = new Dice(100); // create a 100-sided Dice object & name it dice3

      System.out.println(
         "\n" + "dice1 = " + dice1 + "\n"
              + "dice2 = " + dice2 + "\n"
              + "dice3 = " + dice3 + "\n"
      );

      System.out.println("Invoke the roll operation on each dice!");

      dice1.roll(); // invoke the role operation of the Dice object named dice1
      dice2.roll(); // invoke the role operation of the Dice object named dice2
      dice3.roll(); // invoke the role operation of the Dice object named dice3

      System.out.println(
         "\n" + "dice1 = " + dice1 + "\n"
              + "dice2 = " + dice2 + "\n"
              + "dice3 = " + dice3 + "\n"
      );
   }
} // end DiceClientV1 class
```