Week-3: Classes, Objects and Methods

Introduction:

- Java is a true object-oriented language.
- Anything we wish to represent in a Java program must be encapsulated in a class that defines the state and behaviour of the basic program components known as **objects.**
- Classes create objects and objects use methods to communicate between them.

Defining a class:

- **Class is a user defined data type in which data and methods are put together as a single unit.**
- Once the class type has been defined user can create "variables" of that type using declarations that are similar to the basic type declarations.
- The data are called **data members** or **instance variables** or **data fields**. And methods are called **member functions**.
- **The general form of class definition is**:

```
class  classname[extends superclassname]
  {
        [fields declaration;]
        [methods declaration;]
  }
```

where,

- classname and superclassname are any valid Java identifiers.
- class declaration starts with keyword **class** followed by **class-name**.
- The keyword extends indicates that properties of the superclassname class are extended to the classname class.
- Everything inside the square brackets is optional.
- Class definition includes variable or field declaration and method declaration.
- Variable and methods are enclosed within pair of braces.

Example:

```
class Student
{
        int rollNo;       //variables
        String name;   //variables
void display ()     //methods
  {
        System.out.println ("Student Roll Number is: " + rollNo);
        System.out.println ("Student Name is: " + name);
   }
}
```

Fields Declaration:

- Variables declared inside the class definitions are called **instance variables**. Because, they are created whenever an object of the class is instantiated.
- User can declare the instance variables exactly the same as declare the local variables.

Example:

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

- The class **Rectangle** contains two integer type instance variables.
- It can also be declare in **single line** as : **int length, width;**
- Instance variables are also called "**member variables**". Here variables are only declared and no storage space has been created in the memory.

Methods Declaration:

- A class with only data fields (and without methods that operate on that data) has no life.
- Methods are declared inside the body of the class after the declaration of instance variable.
- The methods are called member **functions**.
- Methods are declared inside the body of the class immediately after the declaration of variables.
- **The general form of a method declaration is :**

```
type methodname (parameter-list)
{
    method- body;
}
```

- **Method declaration have 4 parts:**
  - **The name of the method (method name)**: Method name is a valid identifier.
  - **The type of values the method returns (type):** It could be **void** type, if the method does not return any value. It could be a simple data-type such **int** as well as any class type.
  - **List of parameters (parameter-list):** It is always enclosed in parentheses, this list contains variable names and types of all the values we give as input to the method, variables are separated by commas.
  - **Body of the method**: The body describes operations to be performed on the data.

Example:

```
class Rectangle
  {
    int  length, width;
    void getData ( int x, int y)     // Method declaration
      {
        length = x;
        width = y;
    }
}
```

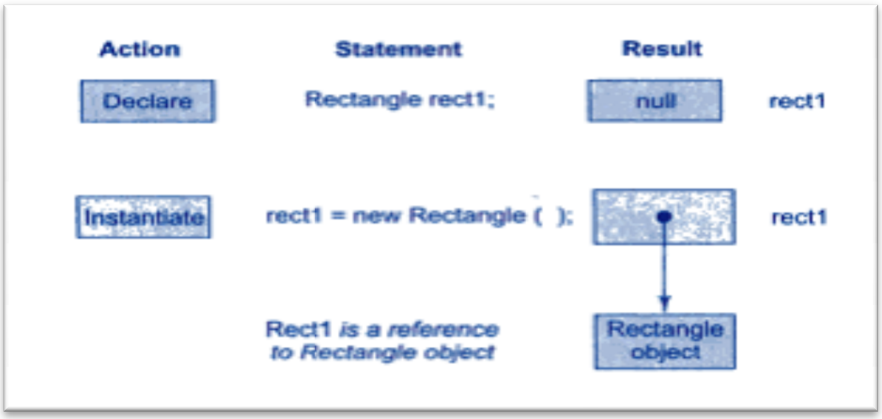
- The method has a return type **void** because it does not return any value.
- We pass two integer values to the method which is then assigned to the instance variables **length** and **width**.
- The **getData** method provides value to the instance variables.

Creating objects:

- In Java, an object is an instance of a class i.e. a block of memory that contains space to store all the instance variables. Creating objects is also referred as **instantiating an object.**
- Objects in java are created using **new** operator. The new operator creates an object of the specified class and returns a reference to that object.
- Here is an example of creating object

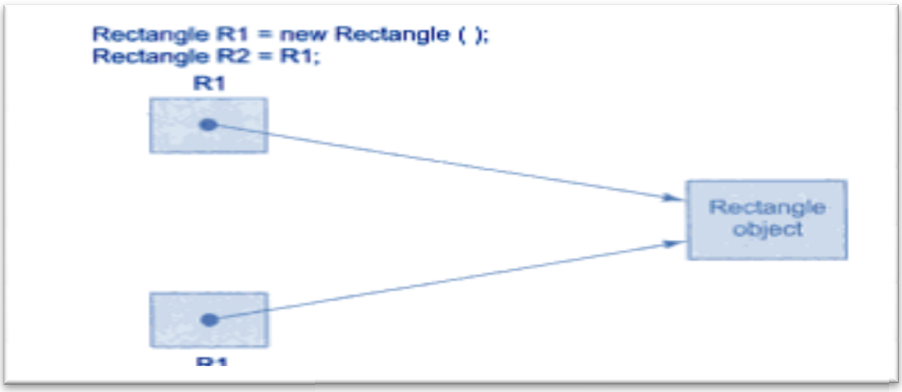
```
Rectangle rect1;          //declare object
rect1 = new Rectangle ();     //instantiate the object
```

- The first statement declares a variable to hold the object reference and the second one actually assign the object reference to the variable.



- Both statements can be combined into one as follows

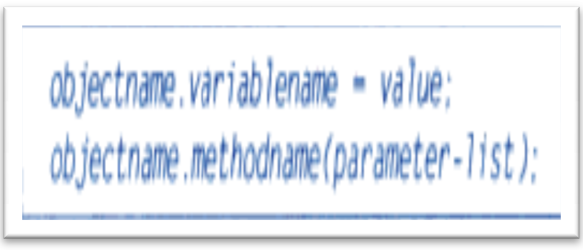
  Rectangle rect1 = **new** Rectangle ();



Accessing class member:

- After creating objects, each containing its own set of variables, we should assign values to these variables in order to use them in our program. All variables must be assigned the values before they are used. There are two ways of accessing class members

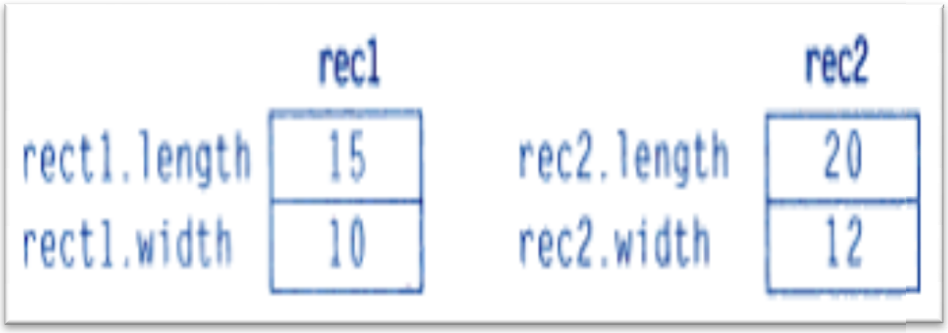
1. **Outside the class**: - Since we are outside the class, we cannot access the instance variables and the method directly. To do this, we must use the concerned object and the dot operator.



- Here objectname is the name of the object and variablename is the instance variable inside the object that we wish to access.
- The methodname is the method that we wish to call ,and parameter-list is a comma separated list of "actual values.
- The instance variables of the rectangle class may be accessed and assigned values as fallows.

```
Rect1.length = 15;
Rec1.width  =  10;
Rect2.length = 20;
Rect2.width = 12;
```

- Note that the two objects rect1 and rect2 store different values as shown below



2. Inside the Class: - Another way and more convenient way of assigning values to the instance variables are to use a method that is declared inside the class.

- In our case the method getData can be used to do this work.
- We can call the getData method on any rectangle object to set values of both length and width.
- Here is the code segment to achieve this.

```
Rectangle rect1 = new rectangle ( );  //creating an object
rect1.getData (15, 10);               //calling the method using the object
```

- The code creates **rect1**object and then passes in the value 15 and 10 for the x and y parameter of the method getData.
- This method then assigns these values to length and width variables respectively as shown below.

```
void getData (int x, int y)
   {
        length= x;   //accessing class members without dot operator
        width=y;
   }
```

Constructors in Java:

- In Java, a 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 constructor, memory for the object is allocated in the memory.
- It is a special type of method which is used to initialize the object.
- Every time an object is created using the new keyword, at least one constructor is called.
- It calls a default constructor if there is no constructor available in the class. In such case, Java compiler provides a default constructor by default.
- There are two types of constructors in Java: Default constructor and Parameterized constructor.
- Rules for creating Java constructor
  - Constructor name must be the same as its class name
  - A Constructor must have no explicit return type
  - A Java constructor cannot be abstract, static, final, and synchronized
- There are two types of constructors in Java:
  1. Default constructor (No-argument constructor)
  2. Parameterized constructor
  3. Copy constructor

1. Default Constructor (No-Argument constructor)

- A constructor is called "Default Constructor" when it doesn't have any parameter.
- The default constructor is used to provide the default values to the object like 0, null, etc., depending on the type.
- Syntax of default constructor:
  <class_name>()
  { }
- In the following example, we are creating the no-argument constructor in the Bike class. It will be invoked at the time of object creation.

```
//Java Program to create and call a default constructor
class Bike
{
   Bike() //creating a default constructor
    {
      System.out.println("Bike is created");
    }
    public static void main(String args[])     //main method
     {
        Bike b=new Bike();            //calling a default constructor
      }
    }
```

Output:Bike is created.

2. Parameterized Constructor

- A constructor which has a specific number of parameters is called a parameterized constructor.
- The parameterized constructor is used to provide different values to distinct objects. However, you can provide the same values also.
- In the following example, we have created the constructor of Student class that has two parameters. We can have any number of parameters in the constructor.

```
//Java Program to demonstrate the use of the parameterized constructor.
class Student
{
   int id;
   String name;
   //creating a parameterized constructor
   Student(int i, String n)
      {
          id = i;
          name = n;
   }
   //method to display the values
   void display()
       {
            System.out.println(id+" "+name);
      }
    public static void main(String args[])
        {
          //creating objects and passing values
           Student s1 = new Student(111,"Karan");
           Student s2 = new Student(222,"Aryan");
         //calling method to display the values of object
          s1.display();
          s2.display();
        }
}
```

Output: 111 Karan
222 Aryan

3. Copy Constructor

- In Java, a copy constructor is a special type of constructor that creates an object using another object of the same Java class. It returns a duplicate copy of an existing object of the class.

Example:

```
public class Fruits
{
   private double price;
   private String name;
 public Fruits(Fruits fruits)  //copy constructor
    {
        //copying each filed
        this.price = fruits.price; //getter
        this.name = fruits.name; //getter
    }
}
```

Constructor Overloading:

- In Java, it is possible to create constructors that have same name but different parameter lists and different definitions. This is called constructor overloading.
- In constructor overloading all constructor definitions have the **same name but with different parameter lists.**
- The difference may either be in the number or type of arguments.
- Here, we are overloading the constructor method Student (). So called constructor overloading.
- Example: (Write a program to illustrate default constructor and parameterized constructor)

```
class Student
{
        String name;
        int regno;
        Student()    // default constructor
          {
                name="Raju";
                regno=12345;
          }
       Student(String n, int  r) // parameterized constructor
          {
            name=n;
            regno=r;
          }
       Student(Student s)   // copy constructor
          {
                name=s.name;
                regno=s.regno;
          }
          void display()
          {
             System.out.println(name + "\t" +regno);
          }
    }
class StudentDemo
     {
   public static void main(String args[])
      {
          Student s1=new Student();
          Student s2=new Student("Ravi",1489);
          Student s3=new Student(s1);
          s1.display();
          s2.display();
          s3.display();
    }
}
```

Java Destructor:

- In Java, when we create an object of the class it occupies some space in the memory (heap). If we do not delete these objects, it remains in the memory and occupies unnecessary space that is not upright from the aspect of programming.
- To resolve this problem, we use the **destructor**. In this section, we will discuss the alternate option to the **destructor in Java**. Also, we will also learn how to use the **finalize()** method as a destructor.
- The **destructor** is the opposite of the constructor. The constructor is used to initialize objects while the destructor is used to delete or destroy the object that releases the resource occupied by the object.
- Remember that **there is no concept of destructor in Java**. In place of the destructor, Java provides the garbage collector that works the same as the destructor.
- The garbage collector is a program (thread) that runs on the JVM. It automatically deletes the unused objects (objects that are no longer used) and free-up the memory. The programmer has no need to manage memory, manually. It can be error-prone, vulnerable, and may lead to a memory leak.
- **What is the destructor in Java?**
  - It is a special method that automatically gets called when an object is no longer used. When an object completes its life-cycle the garbage collector deletes that object and deallocates or releases the memory occupied by the object.
- Advantages of Destructor
  - It releases the resources occupied by the object.
  - No explicit call is required; it is automatically invoked at the end of the program execution.
  - It does not accept any parameter and cannot be overloaded.

- **Java finalize() Method**
  - It is difficult for the programmer to forcefully execute the garbage collector to destroy the object. But Java provides an alternative way to do the same. The Java Object class provides the **finalize()** method that works the same as the destructor. The syntax of the finalize() method is as follows:

- **Syntax:**

```
protected void finalize throws Throwable()
        {
                //resources to be close
        }
```

- **DestructorExample.java**

```
public class DestructorExample
  {
public static void main(String[] args)
   {
        DestructorExample de = new DestructorExample ();
        de.finalize();
        de = null;
```

```
            System.gc();
            System.out.println("Inside the main() method");
    }
    protected void finalize()
    {
        System.out.println("Object is destroyed by the Garbage Collector");
    }
}
```

Output:

Object is destroyed by the Garbage Collector
Inside the main() method
Object is destroyed by the Garbage Collector

Method Overloading:

- In Java, it is possible to create methods that have the same name, but different parameter lists and different definitions. This is called **Methods Overloading**.
- **Methods Overloading** is used when objects are required to perform similar tasks with different input parameters.
- When we call a method in an object, java matches up the method name first and then the number and type of parameters to decide which one of the definitions to execute. This process is known as **polymorphism**.
- To create an overloaded method, all we have to do is to provide several different method definitions in the class, all with the same name, but different parameter lists.
- The difference may be either in the number or type of arguments. That is, each parameter list should be unique.
- Note that the method's return type does not play an important role in this.

- **Example:** (With an example explain, method overloading)

```
class Addition
{
        void add(int a, int b)
        {
                int sum = a + b;
                System.out.println("The sum is " + sum);
        }

        void add(double a, double b)
        {
                double sum = a + b;
                System.out.println("The sum is " + sum);
        }
```

```
        void add(int a, int b, int c)
        {
                int sum = a + b + c;
                System.out.println("The sum is " + sum);
        }
        void add(String s1, String s2)
        {
                String s3 = s1 + s2;
                System.out.println("The string is " + s3);
        }

}

class AddDemo
{
        public static void main(String args[])
        {
                Addition obj = new Addition();
                obj.add(10, 20);
                obj.add(12.2, 25.6);
                obj.add(14, 20, 35);
                obj.add(15, 22.5);
                obj.add("Hello"," World");
        }
}
```

Access modifiers:

- The variables and methods of a class are visible everywhere in the program. However it may be necessary in some situations to restrict the access to certain variables and methods from outside the class.
- This can be achieved by applying **visibility modifiers or access modifiers or access specifiers.**
- Java provides three types of visibility modifiers: public, private and protected.

i) public access:

- To make any variable or method to be visible to all classes in the program, then we may declare as public.

  Ex: public int number;

```
    public void sum()
     {
        -------
     }
```

- A variable or method declared as public has the widest possible visibility and accessible everywhere.

ii) private access:

- Private fields enjoy the highest degree of protection. They are accessible only with their own class.
- They cannot be inherited by subclass.
- A method declared as private behaves like a method declared as final.

Ex: private int x;
private void login();

iii) protected access:

- The visibility level of a "protected" filed lies in between the public access and friendly access.
- The protected modifier makes the fields visible to all classes and subclasses in the same package and to subclasses in their package.

iv) friendly access:

- When no access modifier is specified, the member defaults to a limited version of public accessibility known as "friendly" level of access.
- The difference between the public access and friendly access is that the public modifier makes fields visible in all classes, while the friendly access makes fields visible only in the same package, but not in other packages.(A package is a group of classes stored separately).

v) private protected access:

- A field can be declared with two keywords private and protected together like:
  Ex:private protected int codeNumber;
- This gives a visibility level in between the protected access and private access.
- This modifier makes the fields visible in all subclasses regardless of what package they are in
- These fields are not accessible by other classes in the same package.
- The below tables summarizes the visibility provided by various access modifiers.

Access modifier Access locations	Public	Protected	Friendly	Private protected	Private
Same Class	Yes	Yes	Yes	Yes	Yes
Sub class in same package	Yes	Yes	Yes	Yes	No
Other classes in same package	Yes	Yes	Yes	No	No
Subclass in other package	Yes	Yes	No	Yes	No
Non subclass in other package	Yes	No	No	No	No

this keyword:

- The this keyword refers to the current object in a method or constructor.
- The most common use of the this keyword is to eliminate the confusion between class attributes and parameters with the same name (because a class attribute is shadowed by a method or constructor parameter.
- Here is given the 6 usage of java this keyword.
  - this can be used to refer current class instance variable.
  - this can be used to invoke current class method (implicitly)
  - this can be used to invoke current class constructor.
  - this can be passed as an argument in the method call.
  - this can be passed as argument in the constructor call.
  - this can be used to return the current class instance from the method.

```
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();
                  }
          }
```

Autoboxing and unboxing:

- **Autoboxing:** The conversion of a primitive value into an object of the corresponding wrapper class is called autoboxing.
- For example, conversions of int type to Integer object or char type to Character object. This conversion is done implicitly by the Java compiler during program execution.
- **Unboxing** on the other hand refers to converting an object of a wrapper type to its corresponding primitive value. For example conversion of Integer to int.
- For example, conversion of Integer type to int type or Byte to byte type etc. Java automatically performs this conversion during program execution.

Wrapper classes:

- For a many of the operations java does not supports primitive data types like int ,float, double etc.
- A primitive data type has to be converted into object types using the wrapper classes.
- Wrapper classes are present in "java.lang" package.
- Each primitive data types contains corresponding wrapper classes type

Simple type	Wrapper class
int	Integer
float	Float
double	Double
long	Long
char	Character
boolean	Boolean

- The wrapper classes have number of methods to handling primitive data types & objects.

Converting primitive numbers to object numbers using constructor methods.

Constructor calling	Conversion Action
Integer IntVal= new Ingeter(i);	Primitive integer to Integer Object
Float FloatVal= new Float(f);	Primitive float to Float Object
Double DoubleVal= new Double(d);	Primitive double to Double Object
Long LongVal= new Long (l);	Primitive long to Long Object