Unit III

Java as Object Oriented Programming

Language- Overview

UNIT III

JAVA AS OBJECT ORIENTED PROGRAMMING LANGUAGE- OVERVIEW

Syllabus

Fundamentals of JAVA, Arrays: one dimensional array, multi-dimensional array, alternative array declaration statements, **String Handling:** String class methods, **Classes and Methods**: class fundamentals, declaring objects, assigning object reference variables, adding methods to a class, returning a value, constructors, this keyword, garbage collection, finalize() method, overloading methods, argument passing, object as parameter, returning objects, access control, static, final, nested and inner classes, command line arguments, variable - length arguments.

FUNDAMENTALS OF JAVA

Java programming language was originally developed by Sun Microsystems which was initiated by James Gosling and released in 1995 as core component of Sun Microsystems' Java platform (Java 1.0 [J2SE]).

The latest release of the Java Standard Edition is Java SE 8. With the advancement of Java and its widespread popularity, multiple configurations were built to suit various types of platforms. For example: J2EE for Enterprise Applications, J2ME for Mobile Applications. The new J2 versions were renamed as Java SE, Java EE, and Java ME respectively. Java is guaranteed to be **Write Once, Run Anywhere**.

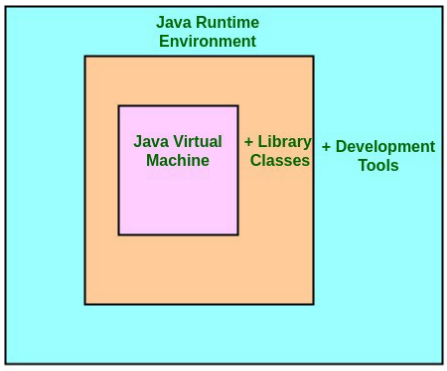
Key Features:

- **Object Oriented**: In Java, everything is an Object. Java can be easily extended since it is based on the Object model.
- **Platform Independent**: Unlike many other programming languages including C and C++, when Java is compiled, it is not compiled into platform specific machine, rather into platform independent byte code. This byte code is distributed over the web and interpreted by the Virtual Machine (JVM) on whichever platform it is being run on.
- **Simple:** Java is designed to be easy to learn. If you understand the basic concept of OOP Java, it would be easy to master.
- **Secure:** With Java's secure feature it enables to develop virus-free, tamper-free systems. Authentication techniques are based on public-key encryption.

- **Architecture-neutral:** Java compiler generates an architecture-neutral object file format, which makes the compiled code executable on many processors, with the presence of Java runtime system.
- **Portable:** Being architecture-neutral and having no implementation dependent aspects of the specification makes Java portable. Compiler in Java is written in ANSI C with a clean portability boundary, which is a POSIX subset.
- **Robust:** Java makes an effort to eliminate error prone situations by emphasizing mainly on compile time error checking and runtime checking.
- **Multithreaded:** With Java's multithreaded feature it is possible to write programs that can perform many tasks simultaneously. This design feature allows the developers to construct interactive applications that can run smoothly.
- **Interpreted:** Java byte code is translated on the fly to native machine instructions and is not stored anywhere. The development process is more rapid and analytical since the linking is an incremental and light-weight process.
- **High Performance:** With the use of Just-In-Time compilers, Java enables high performance.
- **Distributed:** Java is designed for the distributed environment of the internet.
- **Dynamic:** Java is considered to be more dynamic than C or C++ since it is designed to adapt to an evolving environment. Java programs can carry extensive amount of run-time information that can be used to verify and resolve accesses to objects on run-time.

Java Platform

The platform required to develop and execute Java programs consists of three key components that are important to understand — the Java Development Kit (JDK), the Java Runtime Environment (JRE), and the Java Virtual Machine (JVM). The JDK consists of the JRE (and other development tools) which includes the JVM — the graphic below depicts the relationship between the three components.



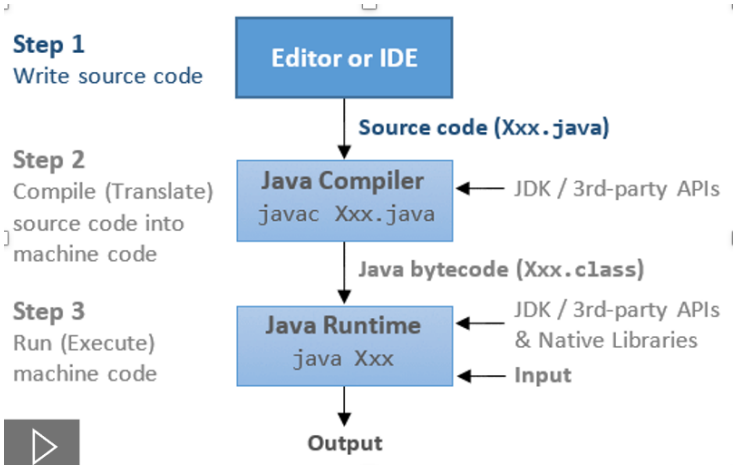
JDK = JRE + Development Tool
JRE = JVM + Library Classes

Breaking down components that make-up the platform —

Java Development Kit (JDK) — Composed of the tools required to compile, document, and package Java programs. This includes the JRE, an interpreter/class loader, a compiler, an archiver, and a documentation generator. Put simply, the JDK provides a complete toolkit for standard Java **development** and program **execution**.

Java Runtime Environment (JRE) — Although it is included in the JDK, it is also available for use as a standalone component. The JRE provides the minimum requirements for **executing** a Java application but doesn't include all features required for development; it consists of the JVM and the some core libraries which enable end-users to run Java applications.

Java Virtual Machine (JVM) — The component responsible for executing the Java program line by line hence it is also known as a run-time **interpreter**. It interprets compiled Java bytecode for a computer's hardware platform so that it can perform a Java program's instructions.



The graphic above depicts the high-level workflow for developers' looking to create and execute Java programs.

Elements of Java Program

When we consider a Java program, it can be defined as a collection of objects that communicate via invoking each other's methods. Let us now briefly look into what do class, object, methods, and instance variables mean.

- **Object** - Objects have states and behaviors. Example: A dog has states - color, name, breed as well as behavior such as wagging their tail, barking, eating. An object is an instance of a class.
- **Class** - A class can be defined as a template/blueprint that describes the behavior/state that the object of its type supports.
- **Methods** - A method is basically a behavior. A class can contain many methods. It is in methods where the logics are written, data is manipulated and all the actions are executed.
- **Instance Variables** - Each object has its unique set of instance variables. An object's state is created by the values assigned to these instance variables.

Java Program Structure:

A typical structure of a Java program contains the following elements:

Documentation Section
Package Declaration
Import Statements
Interface Section
Class Definition
Class Variables and Variables
Main Method Class
Methods and Behaviors

Documentation Section

The documentation section is an important section but optional for a Java program. It includes **basic information** about a Java program. The information includes the **author's name, date of creation, version, program name, company name,** and **description** of the program. It improves the readability of the program. Whatever we write in the documentation section, the Java compiler ignores the statements during the execution of the program.

Package Declaration

The package declaration is optional. It is placed just after the documentation section. In this section, we declare the **package name** in which the class is placed. Note that there can be **only one package** statement in a Java program.

Import Statements

The package contains the many predefined classes and interfaces. If we want to use any class of a particular package, we need to import that class. The import statement represents the class stored in the other package. We use the **import** keyword to import the class. It is written before the class declaration and after the package statement. For example:

1. **import** java.util.Scanner; //it imports the Scanner class only
2. **import** java.util.*; //it imports all the class of the java.util package

Interface Section

It is an optional section. We can create an **interface** in this section if required. We use the **interface** keyword to create an interface. An interface is a slightly different from the class. It contains only **constants** and **method** declarations. Another difference is that it cannot be instantiated. We can use interface in classes by using the **implements** keyword. An interface can also be used with other interfaces by using the **extends** keyword. For example:

```
interface car
{
void start();
void stop();
}
```

Class Definition

In this section, we define the class. It is **vital** part of a Java program. Without the class, we cannot create any Java program. A Java program may conation more than one class definition. We use the **class** keyword to define the class. The class is a blueprint of a Java program. It contains information about user-defined methods, variables, and constants. Every Java program has at least one class that contains the main() method. For example:

```
class Student //class definition
{
}
```

Class Variables and Constants

In this section, we define variables and **constants** that are to be used later in the program. In a Java program, the variables and constants are defined just after the class definition. The variables and constants store values of the parameters. It is used during the execution of the program. We can also decide and define the scope of variables by using the modifiers. It defines the life of the variables. For example:

```
class Student //class definition
{
String sname;  //variable
int id;
double percentage;
}
```

Main Method Class

In this section, we define the **main() method.** It is essential for all Java programs. Because the execution of all Java programs starts from the main() method. In other words, it is an entry point of the class. It must be inside the class. Inside the main method, we create objects and call the methods. We use the following statement to define the main() method:

```
public static void main(String args[])
{
}
```

For example:

```
public class Student //class definition
{
public static void main(String args[])
{
//statements
}
}
```

You can read more about the Java main() method here.

Methods and behavior

In this section, we define the functionality of the program by using the methods. The methods are the set of instructions that we want to perform. These instructions execute at runtime and perform the specified task. For example:

```
public class Demo //class definition
```

```
{
public static void main(String args[])
{
void display()
{
System.out.println("Welcome to javatpoint");
}
//statements
}
}
```

First Java Program

Let us look at a simple code that will print the words Hello World.

```
public class MyFirstJavaProgram {
       /* This is my first java program.
       * This will print 'Hello World' as the output
       */
 public static void main(String []args) {
 System.out.println("Hello World"); // prints Hello World
 }
}
```

Let's look at how to save the file, compile, and run the program. Please follow the subsequent steps:

- Open notepad and add the code as above.
- Save the file as: MyFirstJavaProgram.java.
- Open a command prompt window and go to the directory where you saved the class. Assume it's C:\.
- Type 'javac MyFirstJavaProgram.java' and press enter to compile your code. If there are no errors in your code, the command prompt will take you to the next line (Assumption : The path variable is set).
- Now, type ' java MyFirstJavaProgram ' to run your program.
- You will be able to see ' Hello World ' printed on the window.

```
C:\> javac MyFirstJavaProgram.java
C:\> java MyFirstJavaProgram
```

```
Hello World
```

Basic Syntax

About Java programs, it is very important to keep in mind the following points.

- **Case Sensitivity** - Java is case sensitive, which means identifier Helloand hello would have different meaning in Java.
- **Class Names** - For all class names the first letter should be in Upper Case. If several words are used to form a name of the class, each inner word's first letter should be in Upper Case.

  Example: `class MyFirstJavaClass`
- **Method Names** - All method names should start with a Lower Case letter. If several words are used to form the name of the method, then each inner word's first letter should be in Upper Case.

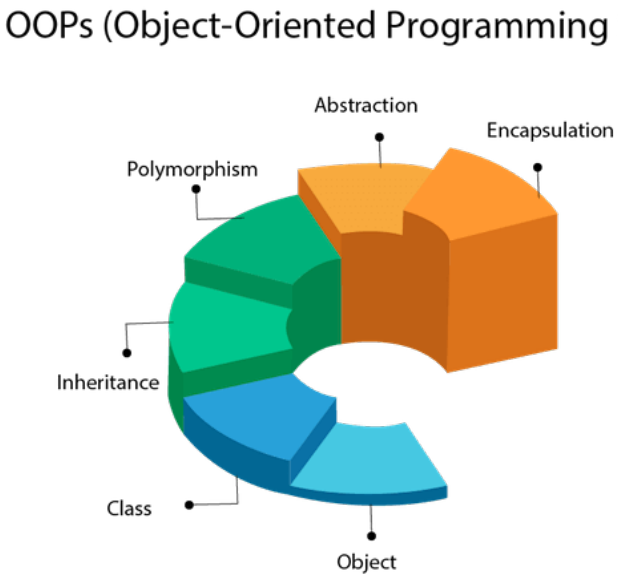
  Example: `public void myMethodName()`
- **Program File Name** - Name of the program file should exactly match the class name. When saving the file, you should save it using the class name (Remember Java is case sensitive) and append '.java' to the end of the name (if the file name and the class name do not match, your program will not compile).

  Example: Assume '**MyFirstJavaProgram**' is the class name. Then the file should be saved as **'MyFirstJavaProgram.java'**
- **public static void main(String args[])** - Java program processing starts from the main() method which is a mandatory part of every Java program.

Java OOPs Concepts:

Java is an Object-Oriented Language.

Object-Oriented Programming is a methodology or paradigm to design a program using classes and objects. It simplifies software development and maintenance by providing some concepts. As a language that has the Object-Oriented feature, Java supports the following fundamental concepts:



1. **Object**

Any entity that has state and behavior is known as an object. For example, a chair, pen, table, keyboard, bike, etc. It can be physical or logical.

An Object can be defined as an instance of a class. An object contains an address and takes up some space in memory. Objects can communicate without knowing the details of each other's data or code. The only necessary thing is the type of message accepted and the type of response returned by the objects.

Example: A dog is an object because it has states like color, name, breed, etc. as well as behaviors like wagging the tail, barking, eating, etc.

2. **Class**

***Collection of objects* is called class**. It is a logical entity. A class can also be defined as a blueprint from which you can create an individual object. Class doesn't consume any space.

3. **Inheritance**

When one object acquires all the properties and behaviors of a parent object, it is known as inheritance. It provides code reusability. It is used to achieve runtime polymorphism.

4. **Polymorphism**

If *one task is performed in different ways*, it is known as polymorphism. For example: to convince the customer differently, to draw something, for example, shape, triangle, rectangle, etc.
In Java, we use method overloading and method overriding to achieve polymorphism.
Another example can be to speak something; for example, a cat speaks meow, dog barks woof, etc.

5. **Abstraction**

Hiding internal details and showing functionality is known as abstraction. For example phone call, we don't know the internal processing. In Java, we use abstract class and interface to achieve abstraction.

6. **Encapsulation**

Binding (or wrapping) code and data together into a single unit are known as encapsulation. For example, a capsule, it is wrapped with different medicines.

Capsule

A java class is the example of encapsulation. Java bean is the fully encapsulated class because all the data members are private here.

Advantage of OOPs over Procedure-oriented programming language

1) OOPs makes development and maintenance easier, whereas, in a procedure-oriented programming language, it is not easy to manage if code grows as project size increases.
2) OOPs provides data hiding, whereas, in a procedure-oriented programming language, global data can be accessed from anywhere.
3) OOPs provides the ability to simulate real-world event much more effectively. We can provide the solution of real word problem if we are using the Object-Oriented Programming language.

Array

An array is a group of similar typed variables that are referred to by a common name. Arrays of any type can be created and may have one or more dimensions. A specific element in an array is accessed by its index. The array is a simple type of data structure which can store primitive variable or objects.

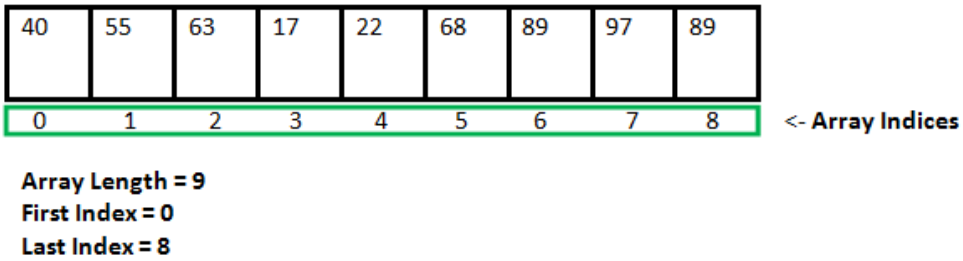
An array in Java is a group of like-typed variables referred to by a common name. Arrays in Java work differently than they do in C/C++. Following are some important points about Java arrays.

In Java, all arrays are dynamically allocated. (discussed below)

Since arrays are objects in Java, we can find their length using the object property length. This is different from C/C++, where we find length using sizeof.

- A Java array variable can also be declared like other variables with [] after the data type.
- The variables in the array are ordered, and each has an index beginning from 0.
- Java array can be also be used as a static field, a local variable, or a method parameter.
- The size of an array must be specified by int or short value and not long.
- The direct superclass of an array type is Object.
- Every array type implements the interfaces Cloneable and java.io.Serializable.

An array can contain primitives (int, char, etc.) and object (or non-primitive) references of a class depending on the definition of the array. In the case of primitive data types, the actual values are stored in contiguous memory locations. In the case of class objects, the actual objects are stored in a heap segment.



One-Dimensional Arrays:

Java provides a data structure, the *array*, which stores a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables.

This tutorial introduces how to declare array variables, create arrays, and process arrays using indexed variables.

Declaring Array Variables

To use an array in a program, you must declare a variable to reference the array, and you must specify the type of array the variable can reference. Here is the syntax for declaring an array variable −

Syntax

```
dataType[] arrayRefVar;   // preferred way.
or
dataType arrayRefVar[];  // works but not preferred way.
```

Note − The style dataType[] arrayRefVar is preferred. The style dataType arrayRefVar[] comes from the C/C++ language and was adopted in Java to accommodate C/C++ programmers.

Example

The following code snippets are examples of this syntax −

```
double[] myList;   // preferred way.
or
double myList[];   // works but not preferred way.
```

Creating Arrays

You can create an array by using the new operator with the following syntax −

Syntax

```
arrayRefVar = new dataType[arraySize];
```

The above statement does two things −

- It creates an array using new dataType[arraySize].
- It assigns the reference of the newly created array to the variable arrayRefVar.

Declaring an array variable, creating an array, and assigning the reference of the array to the variable can be combined in one statement, as shown below −

```
dataType[] arrayRefVar = new dataType[arraySize];
```

Alternatively you can create arrays as follows −

```
dataType[] arrayRefVar = {value0, value1, ..., valuek};
```

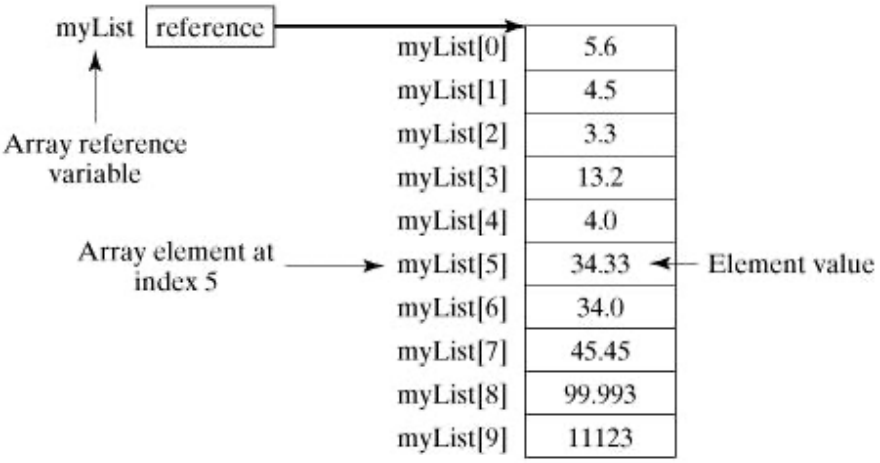
The array elements are accessed through the **index**. Array indices are 0-based; that is, they start from 0 to **arrayRefVar.length-1**.

Example

Following statement declares an array variable, myList, creates an array of 10 elements of double type and assigns its reference to myList −

```
double[] myList = new double[10];
```

Following picture represents array myList. Here, myList holds ten double values and the indices are from 0 to 9.



Processing Arrays

When processing array elements, we often use either **for** loop or **foreach** loop because all of the elements in an array are of the same type and the size of the array is known.

Example

Here is a complete example showing how to create, initialize, and process arrays −

```
public class TestArray {

   public static void main(String[] args) {
      double[] myList = {1.9, 2.9, 3.4, 3.5};

      // Print all the array elements
      for (int i = 0; i < myList.length; i++) {
         System.out.println(myList[i] + " ");
      }

      // Summing all elements
```

```
      double total = 0;
      for (int i = 0; i < myList.length; i++) {
         total += myList[i];
      }
      System.out.println("Total is " + total);

      // Finding the largest element
      double max = myList[0];
      for (int i = 1; i < myList.length; i++) {
         if (myList[i] > max) max = myList[i];
      }
      System.out.println("Max is " + max);
   }
}
```

This will produce the following result −

Output

```
1.9
2.9
3.4
3.5
Total is 11.7
Max is 3.5
```

The foreach Loops

JDK 1.5 introduced a new for loop known as foreach loop or enhanced for loop, which enables you to traverse the complete array sequentially without using an index variable.

Example

The following code displays all the elements in the array myList −

```
public class TestArray {

   public static void main(String[] args) {
      double[] myList = {1.9, 2.9, 3.4, 3.5};

      // Print all the array elements
      for (double element: myList) {
         System.out.println(element);
      }
   }
}
```

This will produce the following result −

Output

```
1.9
2.9
3.4
3.5
```

Passing Arrays to Methods

Just as you can pass primitive type values to methods, you can also pass arrays to methods. For example, the following method displays the elements in an **int** array −

Example

```
public static void printArray(int[] array) {
   for (int i = 0; i < array.length; i++) {
      System.out.print(array[i] + " ");
   }
}
```

You can invoke it by passing an array. For example, the following statement invokes the printArray method to display 3, 1, 2, 6, 4, and 2 −

Example

```
printArray(new int[]{3, 1, 2, 6, 4, 2});
```

Returning an Array from a Method

A method may also return an array. For example, the following method returns an array that is the reversal of another array −

Example

```
public static int[] reverse(int[] list) {
   int[] result = new int[list.length];

   for (int i = 0, j = result.length - 1; i < list.length; i++, j--) {
      result[j] = list[i];
   }
   return result;
}
```

The Arrays Class

The java.util.Arrays class contains various static methods for sorting and searching arrays, comparing arrays, and filling array elements. These methods are overloaded for all primitive types.

Sr.No.	Method & Description
1	**public static int binarySearch(Object[] a, Object key)** Searches the specified array of Object (Byte, Int , double, etc.) for the specified value using the binary search algorithm. The array must be sorted prior to making this call. This returns index of the search key, if it is contained in the list; otherwise, it returns (– (insertion point + 1)).

2	**public static boolean equals(long[] a, long[] a2)** Returns true if the two specified arrays of longs are equal to one another. Two arrays are considered equal if both arrays contain the same number of elements, and all corresponding pairs of elements in the two arrays are equal. This returns true if the two arrays are equal. Same method could be used by all other primitive data types (Byte, short, Int, etc.)
3	**public static void fill(int[] a, int val)** Assigns the specified int value to each element of the specified array of ints. The same method could be used by all other primitive data types (Byte, short, Int, etc.)
4	**public static void sort(Object[] a)** Sorts the specified array of objects into an ascending order, according to the natural ordering of its elements. The same method could be used by all other primitive data types (Byte, short, Int, etc.)

Arrays of Objects

An array of objects is created like an array of primitive type data items in the following way.

```
Student[] arr = new Student[7]; //student is a user-defined class
```

The studentArray contains seven memory spaces each of the size of student class in which the address of seven Student objects can be stored. The Student objects have to be instantiated using the constructor of the Student class, and their references should be assigned to the array elements in the following way.

```
Student[] arr = new Student[5];
```

What happens if we try to access elements outside the array size?

JVM throws **ArrayIndexOutOfBoundsException** to indicate that the array has been accessed with an illegal index. The index is either negative or greater than or equal to the size of an array.

```
public class GFG
{
    public static void main (String[] args)
    {
        int[] arr = new int[2];
        arr[0] = 10;
        arr[1] = 20;

        for (int i = 0; i <= arr.length; i++)
            System.out.println(arr[i]);
```

```
    }
}
```

Runtime error

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 2
    at GFG.main(File.java:12)
```

Output

```
10

20
```

Typical array-processing code.

ArrayExamples.java contains typical examples of using arrays in Java

create an array with random values	`double[] a = new double[n];`
	`for (int i = 0; i < n; i++)`
	`  a[i] = Math.random();`

print the array values, one per line	`for (int i = 0; i < n; i++)` `  System.out.println(a[i]);`

find the maximum of the array values	`double max = Double.NEGATIVE_INFINITY;`
	`for (int i = 0; i < n; i++)`
	`  if (a[i] > max) max = a[i];`

compute the average of the array values	`double sum = 0.0;`
	`for (int i = 0; i < n; i++)`
	`  sum += a[i];`
	`double average = sum / n;`

reverse the values within an array	`for (int i = 0; i < n/2; i++)`
	`{`
	`  double temp = a[i];`
	`  a[i] = a[n-1-i];`
	`  a[n-i-1] = temp;`
	`}`

copy sequence of values to another array	`double[] b = new double[n];`
	`for (int i = 0; i < n; i++)`
	`  b[i] = a[i];`

Sample Programs : The methods used in this notes are as follows:

- Using Standard Method
- Using Scanner
- Using String

1. **Sample Program Using Standard Method**

This program will help to understand initializing and accessing specific array elements.

```
package arrayDemo;
import java.util.Arrays;
public class ResultListDemo {
        public static void main(String[] args) {
                int resultArray[] = new int[6]; //Array Declaration
                //Array Initialization
                                resultArray[0]=69;
                                resultArray[1]=75;
                                resultArray[2]=43;
                                resultArray[3]=55;
                                resultArray[4]=35;
                                resultArray[5]=87;
                //Array elements access
                System.out.println("First Subject- "+ resultArray[0]);
                System.out.println("Second Subject- "+ resultArray[1]);
                System.out.println("Third Subject- "+ resultArray[2]);
                System.out.println("Fourth Subject- "+ resultArray[3]);
                System.out.println("Fifth Subject- "+ resultArray[4]);
                System.out.println("Sixth Subject- "+ resultArray[5]);

        }

}
```

Output:

```
First Subject– 69
Second Subject– 75
Third Subject– 43
Fourth Subject– 55
Fifth Subject– 35
Sixth Subject– 87
```

2. **Sample Program Using Scanner**

1. Read the array length as sc.nextInt() and store it in the variable len and declare an array int[len].
2. To store elements in to the array for i=0 to i<length of an array read the element using sc.nextInt() and store the element at the index a[i].
3. Display the elements of an array for loop iterates from i=0 to i<length of an array print the array element a[i].

```
import java.util.*;
class OnedimensionalScanner
{
```

```
   public static void main(String args[])
   {
         int len;
         Scanner sc=new Scanner(System.in);
         len=sc.nextInt();
         int a[]=new int[len];//declaration
        for(int i=0; i<len; i++)
        {
           a[i] = sc.nextInt();
        }
        System.out.print("Elements in Array are :\n");
        for(int i=0; i<len; i++)
        {
           System.out.print(a[i] + "  ");
        }
   }
}
```

Output :

```
4
1
2
3
4
Elements in Array are :
1   2 3 4
```

3. **Sample Program Using String**

- We declared one-dimensional string array with the elements strings.
- To print strings from the string array. for i=0 to i<length of the string print string which is at the index str[i].

```
import java.util.*;
class OnedimensionalString
{
    public static void main(String[] args)
    {
        //declare and initialize one dimension array
        String[] str = new String[]{"one", "two", "three", "four"};
        for (int i = 0; i < str.length; i++)
     {
            System.err.println(str[i] + "   ");
        }
   }
}
```

Output

```
one
two
three
four
```

MULTI DIMENSIONAL ARRAYS

Multidimensional Arrays can be defined in simple words as array of arrays. Data in multidimensional arrays are stored in tabular form (in row major order).

Syntax:

```
data_type[1st dimension][2nd dimension][]..[Nth dimension] array_name =
new data_type[size1][size2]….[sizeN];
int[][] a = new int[3][4];
```

Here, we have created a multidimensional array named a. It is a 2-dimensional array, that can hold a maximum of 12 elements,

Remember, Java uses zero-based indexing, that is, indexing of arrays in Java starts with 0 and not 1.

Let's take another example of the multidimensional array. This time we will be creating a 3-dimensional array. For example,

```
String[][][] data = new String[3][4][2];
```

Here, data is a 3d array that can hold a maximum of 24 (3*4*2) elements of type String.

How to initialize a 2d array in Java?

Here is how we can initialize a 2-dimensional array in Java.

```
int[][] a = {
      {1, 2, 3},
      {4, 5, 6, 9},
      {7},
};
```

As we can see, each element of the multidimensional array is an array itself. And also, unlike C/C++, each row of the multidimensional array in Java can be of different lengths.

```
class MultidimensionalArray {
    public static void main(String[] args) {

        // create a 2d array
        int[][] a = {
            {1, 2, 3},
            {4, 5, 6, 9},
            {7},
        };

        // calculate the length of each row
        System.out.println("Length of row 1: " + a[0].length);
        System.out.println("Length of row 2: " + a[1].length);
        System.out.println("Length of row 3: " + a[2].length);
    }
}
```

Output:

```
Length of row 1: 3
Length of row 2: 4
Length of row 3: 1
```

In the above example, we are creating a multidimensional array named a. Since each component of a multidimensional array is also an array (a[0], a[1] and a[2] are also arrays). Here, we are using the *length* attribute to calculate the length of each row.

Multi dimensional array Using *For* Loop

Example: Print all elements of 2d array Using Loop

```
class MultidimensionalArray {
    public static void main(String[] args) {

        int[][] a = {
            {1, -2, 3},
            {-4, -5, 6, 9},
            {7},
        };

        for (int i = 0; i < a.length; ++i) {
            for(int j = 0; j < a[i].length; ++j) {
                System.out.println(a[i][j]);
            }
        }
    }
}
```

Output:

```
1
-2
3
-4
-5
6
9
7
```

Multi dimensional array Using *For Each* Loop

```
class MultidimensionalArray {
    public static void main(String[] args) {

        // create a 2d array
        int[][] a = {
            {1, -2, 3},
            {-4, -5, 6, 9},
```

```
                {7},
            };

            // first for...each loop access the individual array
            // inside the 2d array
            for (int[] innerArray: a) {
                // second for...each loop access each element inside the row
                for(int data: innerArray) {
                    System.out.println(data);
                }
            }
        }
    }
```

Output:

```
1
-2
3
-4
-5
6
9
7
```

In the above example, we are have created a 2d array named a. We then used for loop and for...each loop to access each element of the array.

3 Dimensional Array

Let's see how we can use a 3d array in Java. We can initialize a 3d array similar to the 2d array. For example,

// test is a 3d array

```
int[][][] test = {
        {
          {1, -2, 3},
          {2, 3, 4}
        },
        {
          {-4, -5, 6, 9},
          {1},
          {2, 3}
        }
};
```

Basically, a 3d array is an array of 2d arrays. The rows of a 3d array can also vary in length just like in a 2d array.

Example

```
class ThreeArray {
    public static void main(String[] args) {

        // create a 3d array
        int[][][] test = {
            {
```

```
                    {1, -2, 3},
                    {2, 3, 4}
                },
                {
                    {-4, -5, 6, 9},
                    {1},
                    {2, 3}
                }
            };

            // for..each loop to iterate through elements of 3d array
            for (int[][] array2D: test) {
                for (int[] array1D: array2D) {
                    for(int item: array1D) {
                        System.out.println(item);
                    }
                }
            }
        }
    }
```

Output:

```
1
-2
3
2
3
4
-4
-5
6
9
1
2
3
```

ALTERNATIVE ARRAY DECLARATION SYNTAX

The square brackets follow the type specifier, and not the name of the array variable.

```
type[] var-name;
```

For example, the following two declarations are equivalent:

```
int al[] = new int[3];
int[] a2 = new int[3];
```

The following declarations are also equivalent:

```
char d1[][] = new char[3][4];
char[][] d2 = new char[3][4];
```

This alternative declaration form offers convenience when declaring several arrays at the same time. For example,

```
int[] e1, e2, e3; // create three arrays
```

STRING HANDLING

Strings, which are widely used in Java programming, are a sequence of characters. In Java programming language, strings are treated as objects.

The Java platform provides the String class to create and manipulate strings.

Creating Strings

The most direct way to create a string is to write −

```
String greeting = "Hello world!";
```

Whenever it encounters a string literal in your code, the compiler creates a String object with its value in this case, "Hello world!'.

As with any other object, you can create String objects by using the new keyword and a constructor. The String class has 11 constructors that allow you to provide the initial value of the string using different sources, such as an array of characters.

Example

```
public class StringDemo {

   public static void main(String args[]) {
      char[] helloArray = { 'h', 'e', 'l', 'l', 'o', '.' };
      String helloString = new String(helloArray);
      System.out.println( helloString );
   }
}
```

This will produce the following result −

Output

```
hello.
```

Note − The String class is immutable, so that once it is created a String object cannot be changed. If there is a necessity to make a lot of modifications to Strings of characters, then you should use String Buffer & String Builder Classes.

String Length

Methods used to obtain information about an object are known as **accessor** methods. One accessor method that you can use with strings is the length() method, which returns the number of characters contained in the string object.

The following program is an example of **length(),** method String class.

Example

```
public class StringDemo {

   public static void main(String args[]) {
      String palindrome = "Dot saw I was Tod";
      int len = palindrome.length();
      System.out.println( "String Length is : " + len );
```

```
   }
}
```

This will produce the following result −

Output

```
String Length is : 17
```

Concatenating Strings

The String class includes a method for concatenating two strings −

```
string1.concat(string2);
```

This returns a new string that is string1 with string2 added to it at the end. You can also use the concat() method with string literals, as in −

```
"My name is ".concat("Zara");
```

Strings are more commonly concatenated with the + operator, as in −

```
"Hello," + " world" + "!"
```

which results in −

```
"Hello, world!"
```

Let us look at the following example −

Example

```
public class StringDemo {

   public static void main(String args[]) {
      String string1 = "saw I was ";
      System.out.println("Dot " + string1 + "Tod");
   }
}
```

This will produce the following result −

Output

```
Dot saw I was Tod
```

Creating Format Strings

You have printf() and format() methods to print output with formatted numbers. The String class has an equivalent class method, format(), that returns a String object rather than a PrintStream object.

Using String's static format() method allows you to create a formatted string that you can reuse, as opposed to a one-time print statement. For example, instead of −

Example

```
System.out.printf("The value of the float variable is " +
                  "%f, while the value of the integer " +
                  "variable is %d, and the string " +
                  "is %s", floatVar, intVar, stringVar);
```

You can write –

```
String fs;
fs = String.format("The value of the float variable is " +
                   "%f, while the value of the integer " +
                   "variable is %d, and the string " +
                   "is %s", floatVar, intVar, stringVar);
System.out.println(fs);
```

All String Methods

The String class has a set of built-in methods that you can use on strings.

Method	Description	Return Type
charAt()	Returns the character at the specified index (position)	char
codePointAt()	Returns the Unicode of the character at the specified index	int
codePointBefore()	Returns the Unicode of the character before the specified index	int
codePointCount()	Returns the number of Unicode values found in a string.	int
compareTo()	Compares two strings lexicographically	int
compareToIgnoreCase()	Compares two strings lexicographically, ignoring case differences	int
concat()	Appends a string to the end of another string	String
contains()	Checks whether a string contains a sequence of characters	boolean
contentEquals()	Checks whether a string contains the exact same sequence of characters of the specified CharSequence or StringBuffer	boolean
copyValueOf()	Returns a String that represents the characters of the character array	String
endsWith()	Checks whether a string ends with the specified character(s)	boolean

equals()	Compares two strings. Returns true if the strings are equal, and false if not	boolean
equalsIgnoreCase()	Compares two strings, ignoring case considerations	boolean
format()	Returns a formatted string using the specified locale, format string, and arguments	String
getBytes()	Encodes this String into a sequence of bytes using the named charset, storing the result into a new byte array	byte[]
getChars()	Copies characters from a string to an array of chars	void
hashCode()	Returns the hash code of a string	int
indexOf()	Returns the position of the first found occurrence of specified characters in a string	int
intern()	Returns the canonical representation for the string object	String
isEmpty()	Checks whether a string is empty or not	boolean
lastIndexOf()	Returns the position of the last found occurrence of specified characters in a string	int
length()	Returns the length of a specified string	int
matches()	Searches a string for a match against a regular expression, and returns the matches	boolean
offsetByCodePoints()	Returns the index within this String that is offset from the given index by codePointOffset code points	int
regionMatches()	Tests if two string regions are equal	boolean
replace()	Searches a string for a specified value, and returns a new string where the specified values are replaced	String
replaceFirst()	Replaces the first occurrence of a substring that matches the given regular expression with the given replacement	String
replaceAll()	Replaces each substring of this string that matches the given regular expression with the given replacement	String

split()	Splits a string into an array of substrings	String[]
startsWith()	Checks whether a string starts with specified characters	boolean
subSequence()	Returns a new character sequence that is a subsequence of this sequence	CharSequence
substring()	Returns a new string which is the substring of a specified string	String
toCharArray()	Converts this string to a new character array	char[]
toLowerCase()	Converts a string to lower case letters	String
toString()	Returns the value of a String object	String
toUpperCase()	Converts a string to upper case letters	String
trim()	Removes whitespace from both ends of a string	String
valueOf()	Returns the string representation of the specified value	String

CLASSES AND METHODS:

Class

Everything in Java is associated with classes and objects, along with its attributes and methods. For example: in real life, a car is an object. The car has attributes, such as weight and color, and methods, such as drive and brake.

A Class is like an object constructor, or a "blueprint" for creating objects.

- A *class* is a template or blueprint for how to build an object.
    - A class is a prototype that defines state placeholders and behavior common to all objects of its kind.
    - Each object is a member of a single class — there is no multiple inheritance in Java.

A class is a group of objects which have common properties. It is a logical entity. It can't be physical.

A class in Java can contain:

- Fields
- Methods
- Constructors
- Blocks
- Nested class and interface

Syntax to declare a class:

```
class <class_name>{
    field;
    method;
}
```

E.g. To create a class, use the keyword class:

Main.java

Create a class named "Main" with a variable x: Now you can see how a class Student is defined class Student

```
public class Student {
String name;
String course
int roll_no;
}
```

The class Student is not yet complete because in this class only data members are there. Still a member function to display basic information is needed.

Complete definition of class Student will have three data members-name, course and age and one member function `display_info( ).`

```
//class definition
class Student
       {
       String name;
       String course ;
       int age;
       void display_info( ) // function for displaying basic information
       {
       System.out.println(" Student Information");
       System.out.println("Name:"+name);
       System.out.println("Course:"+course);
       System.out.println("Age:"+age);
       }
       }
       // end of Student class
```

An object of Student class can be created as follows:

```
       Student student_1 = new Student( );
```

As this statement is executed an object student 1 of Student class is created. You will see a detailed discussion of this type of statements in a later section of this unit. Each time you create an object of a class a copy of each instance variables defined in the class is created. In other words you can say that each object of a class has its own copy of data members defined in the class. Member functions have only one copy and shared by all the objects of that class. All the objects may have their own value of instance variables. As given in Figure1 every object of class Student will have its own copy of name, course, and age but only one copy of method display_info( ) for all the objects. Now let us see how objects are declared and used inside a program.

Object

Objects have states and behaviors.

An *object* is an instance of a particular class.

- There are typically many object instances for any one given class (or type).

- Each object of a given class has the same built-in behavior but possibly a different state (data).
- Objects are *instantiated* (created).

An object of a class can be created by performing these two steps.

1. Declare a object of class.
2. Acquire space for and bind it to object.

In the first step only a reference to an variable is created, no actual object's exists. For actual existence of an object memory is needed. That is acquired by using **new** operator.

The **new** operator in Java dynamically allocates memory for an object returns a reference to object and binds the reference with the object. Here reference to object means address of object. So it is essential in Java that all the objects must be dynamically allocated.

Declaring Student object

Step 1:

```
Student Student1; // declaring reference to object
```

Step 2:

```
Student1 = new Student( ); // allocating memory
```

In normal practice these two steps are written in a single statement as

```
Student student1 = new Student( );
```

Now you can see a complete Java program for displaying basic information of students program

```
class Student
     {
     String name;
     String course ;
     int age;
     void display_info( ) // function for displaying basic
     information {
     System.out.println(" Student Information");
     System.out.println("Name:"+name);
     System.out.println("Course:"+course);
     System.out.println("Age:"+age);
     }
     }
     // end of Student class
     class Display_Test {
     public static void main( String para[])
     {
     Student student1;
     student1 = new Student();
     sudent1.name = "Mr.Raj";
     student1.course = "BE";
```

```
student1.age = 23;
student1.display_info();
}
}
```

The output of this program is:

```
Name:Mr.Raj
Course:BE
Age:23
```

Two approaches can be used for assigning value to the instance variables of the objects. In first approval create an object and initialize instance variables one by one, as done in the above program. Initialization of instance variable is done by using dot(.) operator. Dot(.) operator is used to access members (data and method both) of the objects.

```
object_name.variable_name = value;
```

But this method of initialization of objects is not convenient because all instance variables of the object must be initialized carefully, and this exercise should be done for all objects before they are used. In this method of initialization there is a chance of skipping some variables unassigned where large number of objects are to be initialized.

To overcome this problem second approach of object initialization with the help of **constructors**, can be used. We will discuss how constructors are used in the next section of this unit.

ASSIGNING OBJECT REFERENCE VARIABLES

One object can be assigned to another object, of same type but it works very different than a normal assignment. Let us see it with the help of a program.

```
class Person
{
String name;
int age ;
String address;
void Display( )
{
System.out.println("Person Information:"+name +" ("+age
+")"+"\n"+address); }
}
class Reference_Test
{
public static void main(String[] args)
{
Person p = new Person();
```

```
Person q = new Person();
p.name= "Mr.Naveen Kumar";
p.age= 24;
p.address = "248,Sector 22, Pune";
p.Display();
q = p;                  // q refer to p
q.name = "Mr.Suresh";
q.address = "22,Mahanadi ,Maidan Garhi";
p.Display();
q.Display();
}
}
```

Output of this program is:

```
Person Information:Mr.Naveen Kumar (24) 248,Sector 22, Pune
Person Information:Mr.Suresh (24) 22,Mahanadi ,Maidan Garhi Person
Information:Mr.Suresh (24) 22,Mahanadi ,Maidan Garhi
```

In this program two objects p and q are created. Object p is initialized with some values, then Display method is called through object p. It displays the information of object p. Further, object p is assigned to object q as reference variable. You can see in the figure both object p and object q are referring to the same object. Thus any change made in object p or q changes will be reflect the both p and q. You can see in the program that changes made in name and address by q are reflected in p also. Whenever in a program this type of referencing of variables is used care should be taken while changing the values of instance variables of object being referred.

An object is a class instance. The class of an object determines what it is and how it can be manipulated. A class encapsulates methods, data, and implementation of methods. This encapsulation is like a contract between the implementer of the class and the user of that class.

METHODS TO CLASS

A Java class is a group of values with a set of operations. The user of a class manipulates object of that class only through the methods of that class.

General form of a method is:

```
type name_of_method (argument_list) {
// body of method
}
```

type specifies the type of data that will be returned by the method. This can be any data type including class types. In case a method does not return any value, its return type must be void.

The argument–list is a list of type and identifier pairs separated by commas. Arguments are basically variables that receive the values at the time of method invocation. If a method has return type other than void, it returns a value to the calling point using the following form of statement.

```
return value;// value is the value to be returned by function.
```

Suppose we define a class to represent complex numbers. The complex class definition shown in the program given below illustrates how this can be done. Two variables real and imag, are declared. These represent the real and imaginary parts of a complex number (respectively). The program also defines three methods, assignReal and assign Imag() and showComplexl()that can be used to assign values to the real is () part imaginary part of a complex number, and to show the real number respectively.

```
class Complex
{
double real;
double imag;
void assignReal( double r)
{
real = r;
}
void assignImag( double i)
{
imag= i;
}
void showComplex ( )
{
System.out.println("The Complex Number is :"+ real +"+i"+imag);
}
}
class Complex_Test
{
public static void main(String[] args)
{
Complex R1 = new Complex();
R1.assignReal(5);
R1.assignImag(2);
R1.showComplex();
}
}
```

Output of this program is:

```
The Complex Number is :5.0+i2.0
```

Sometimes we need to have the same name of different methods in a class. All these methods perform different operations of a similar nature. For example of two add methods in a class one is used for adding two integer variables and the other for adding two double variables. Methods of the same name in a class are called overloaded methods, and the process of

defining this type of methods is called method overloading. The concept of overloading we cover in more detail in the next section of this unit. The concept of overloading can also be used in case of defining constructors. So far we have seen that the initialization of instance variables is done after creating objects. Now we will use constructors for initialization of instance variables.

- **Static Methods**

Methods and variables that are not declared as static are known as instance methods and instance variables or in other words you can say that they belong to objects of the Class and Objects.

A static method is a characteristic of a class, not of the objects it has created.

Static variables and methods are also known as class variables or class methods since each class variable and each class method occurs once per class. Instance methods and variables occur once per instance of a class or you can say every object is having its own copy of instance variables.

One very important point to note here is that a program can execute a static method without creating an object of the class. All other methods must be invoked through an object, and, therefore an object must exist before they can be used. You have seen that every Java application program has one main()method. This method is always static because Java starts execution from this main()method, and at that time no object is created.

Let us see one example program:

```
import Java.util.Date;
     class DateApp
     {
     public static void main(String args[]) {
     Date today = new Date();
     System.out.println(today);
     } }
```

The last line of the main() method uses the System class from the Java.lang package to display the current date and time. See the line of code that invokes the println () method.

```
     System.out.println(today);
```

Now look at the details of the argument passed to it.

System.out refers to the out variable of the System class. You already know that, to refer to static variables and methods of a class, you use a syntax similar to the C and C++ syntax for obtaining the elements in a structure. You join the class's name and the name of the static method or static variable together with a dot (.)

The point which should be noticed here is that the application never instantiated the System class and that out is referred to directly from the class. This is because out is declared as a static variable: a variable associated with the class rather than with an instance of the class. You can also associate methods with a class--static methods-- using static keyword.

Constructors

A constructor initializes object with its creation. It has the same name as the name of its class. Once a constructor is defined, it is automatically called immediately the memory is allocated before the new operation completes. Constructor does not have any return type, it implicit return type is class object. You can see constructor as a class type. Its job is to initialize the instance variables of an object, so that the created object is usable just after creation.

In program Complex_Test you have seen that for initializing imaginary and real parts, two methods have been used. Both the methods have been invoked on the object one by one. In place of methods if you use constructor for initialization you have to make changes in Complex_Test program. Remove methods used for initializing variables and use one method having same name of the class, i.e., Complex with two arguments.

```
class Complex
{
double real;
double imag;
Complex( double p, double q)
{
System.out.println("Constructor in process...");
real = p;
real = p;
imag = q;
}
void showComplex ( )
{
System.out.println("The Complex Number is :"+ real +"+i"+imag); }
}
class Complex_Test
{
public static void main(String[] args)
{
Complex R1 = new Complex(5,2);
R1.showComplex();
}
}
```

The output of this program is:

```
Constructor in process...
The Complex Number is :5.0+i2.0
```

If you compare this program and the previous Complex_Test program you will find that in this program the instance variable of object R1 is initialized with the help of constructor Complex (5, 2), value 5 has been assigned to the real variable and 2 has been assigned to the imag variable. In the previous program, to initialize these variables, methods assignReal() and assignImag() were used.

Constructors can be defined in two ways. First , a constructor may not have parameter. This type of constructor is known as non-parameterized constructor. The second, a constructor may take parameters. This type of constructor is known as parameterized constructor.

If non-parameterized constructor is used for object creation, instance variables of the object are initialized by fixed values at the time of definition of constructor itself.

You can see this program

```
class Point
{
int x;
int y; Point() {
x= 2; y= 4; }
void Display_Point()
{
System.out.println("The Point is: ("+x+","+y+")"); }
}
class Point_Test
{
public static void main( String args[]) {
Point p1 = new Point();
Point p2 = new Point(); p1.Display_Point(); p2.Display_Point();
}
}
```

Output of this program is: The Point is: (2,4)

```
The Point is: (2,4)
The Point is: (2,4)
```

Constructor Point is a non-parameterized constructor. Both the objects p1 and p2 are created by using Point constructor and are initialized by the same value which is assigned during definition of the constructor. This type of constructors is generally used for the initialization of those objects for which the initial value of instance variables is known. If values of the instance variables are given at the time of object creation, parameterized constructors are used for this. For example, if you want to create Point class object with your initial values of x and y coordinates, you can use parameterized constructor. You have to make the following changes in the previous program.

In place of non-parameterized constructor, define parameterized constructor.

Pass appropriate values as arguments to constructors.

```
class Point
{
int x;
int y;
Point(int a, int b) {
x= a;
y= b;
}
void Display_Point()
{
System.out.println("The Point is: ("+x+","+y+")"); }
}
class Point_1_Test
{
public static void main( String args[]) {
Point p1 = new Point(2,5);
Point p2 = new Point(9,7); p1.Display_Point(); p2.Display_Point();
}
}
```

Output of this program is:

```
The Point is: (2,5)
The Point is: (9,7)
```

In this program you can see that points p1 and p2 are initialized by values of programmer's choice by using parameterized constructor.

Overloading Constructors

You may ask whether there can be more than one constructor in a class. Yes, there may be more than one constructor in a class but all the constructors in a class either will have different types of arguments or different number of arguments passed to it. This is essential because without this it wouldn't be possible to identify which of the constructors is invoked. Having more than one constructor in a single class is known as constructor overloading. In the program given below more than one constructor in class Student are defined.

```
class Student
{
String name;
int roll_no;
String course;
Student( String n, int r, String c) {
name = n;
roll_no = r;
course = c;
}
Student(String n, int r ) {
name = n; roll_no = r; course = "MCA"; }
```

```
void Student_Info( )
{
System.out.println("Name:"+name);
System.out.println("Course:"+course);
System.out.println("Roll Number:"+roll_no);
}
}
class Student_Test
{
public static void main(String[] args)
{
Student s1 = new Student("Ravi Prakash", 987770012,"BCA"); Student
s2 = new Student("Rajeev ", 980070042);
s1.Student_Info();
s2.Student_Info();
}
}
```

Output of this program is:

```
Name:Ravi Prakash
Course:BCA
Roll Number:987770012
Name:Rajeev
Course:MCA
Roll Number:980070042
```

Both the constructors of this program can be used for creating two different types of objects. Here different types are simply related to the values assigned to instance variables of objects during their initialization through constructors. By using a constructor with three arguments all the three instance variables name, roll_no and, course can be initialized. If a constructor with two arguments is used to create object, only instance variables name and roll_no can be given initial value through constructor and the variable course is initialized by a constant value "MCA".

this KEYWORD

If a method wants to refer the object through which it is invoked, it can refer to by using this keyword. You know it is illegal to have two variables of the same name within the same scope in a program. Suppose local variables in a method, or formal parameters of the method, overlap with the name of instance variables of the class, to differentiate between local variables or formal parameters and instance variables, this keyword is used. The reason to use this keyword to resolve any name conflict that might occur between instance variables and local variable is that this keyword can be used to refer to the objects directly. You can see in this

program class Test_This is having one variable named rate and method. Total_Interest is also having one local variable named rate. To avoid conflict between both the rate variables keyword has been used with rate variable of Test_This class.

```
class Test_This {
int rate ;
int amount;
int interest; Test_This( int r, int a) {
rate = r;
amount =a;
}
void Total_Interest( )
{
int rate = 5;
rate = this.rate+rate;
interest = rate*amount/100;
System.out.println("Total Interest on "+amount+" is: "+interest);
}
}
class This_Test
{
public static void main(String[] args)
{
Test_This Ob1 = new This( 5, 5000);
Ob1.Total_Interest();
}
}
```

Output of this program is:

```
Total Interest on 5000 is: 500
```

ARGUMENT PASSING

USING OBJECTS AS PARAMETERS

Objects can be based as parameter and can also be returned from the methods. Let us look at the two aspects one by one.

During problem solving you need to pass arguments through methods. Basically by passing arguments you can generalize a method. During problem solving generalized methods can be used for performing operations on a variety of data.

You can see in this program below that for find the area of square and rectangle. The methods Area_S and Area_R for finding area of square and rectangle respectively are defined.

```
 class Test
{
int Area_S( int i) {
```

```
return i*i; }
int Area_R(int a,int b) {
return a*b; }
}
class Area_Test
{
public static void main(String args[]) {
Test t = new Test();
int area;
area = t.Area_S(5);
System.out.println("Area of Square is : "+area); area =
t.Area_R(5,4);
System.out.println("Area of Rectangle is : "+area); }
}
```

Output of this program is:

```
Area of Square is: 25
Area of Rectangle is: 20
```

Can you tell the way of passing parameters in methods Area_S and Area_R? It is call by value. Right as you have studied about two basic way pass by value and pass by reference of parameter passing in functions.

Now you may ask a question whether parameter passing in Java is by reference or by value. The answer is everything in Java except value, passed by reference.

Pass-by-value means that when you call a method, a copy of the value of each of the actual parameter is passed to the method. You can change that copy inside the method, but this will have no effect on the actual parameters. You can see in the program given below. The method max has two parameters that are passed by value.

```
class Para_Test {
static int max(int a, int b)
{
if (a > b)
return a;
else
return b;
}
public static void main(String[] args)
{
int num1 = 40, num2 = 50, num3;
num3 = max( num1, num2);
System.out.println("The maximum in "+num1 +" and "+num2+" is :
"+num3);
}
}
```

Output of this program is:

```
The maximum in 40 and 50 is: 50
```

The values of variables are always primitives or references, never objects. In Java, we can pass a reference of the object to the formal parameter. If any changes to the local object that take place inside the method will modify the object that was passed to it as argument. Due to this reason objects passing as parameter in Java are referred to as passing by reference. In the program given below object of the class Marks is passed to method Set_Grade, in which instance variable grade of the object passed is assigned some value. This reflects that the object itself is modified.

```
class Marks
{
String name;
int percentage;
String grade;
Marks(String n, int m)
{
name = n;
percentage = m;
}
void Display()
{
System.out.println("Student Name :"+name);
System.out.println("Percentage Marks:"+percentage);
System.out.println("Grade : "+grade);
}
}
class Object_Pass
{
public static void main(String[] args)
{
Marks ob1 = new Marks("Naveen",75);
Marks ob2 = new Marks("Neeraj",45);
Set_Grade(ob1);
ob1.Display();
Set_Grade(ob2);
ob2.Display();
}
static void Set_Grade(Marks m)
{
if (m.percentage >= 60)
m.grade ="A";
else if( m.percentage >=40)
m.grade = "B";
else
m.grade = "F";
}
}
```

Output of this program is:

```
Student Name :Naveen
Percentage Marks:75
Grade : A
```

```
Student Name :Neeraj
Percentage Marks:45
Grade : B
```

Returning Objects

Like other basic data types, methods can return data of class type, i.e. object of class. An object returned from a method can be stored in any other object of that class like as values of basic type returned are stored in variables of that data type. You can see in the program below where an the object of class Salary is returned by method the incr_Salary.

```
class Salary.
{
int basic ;
String E_id;
Salary( String a, int b) {

E_id = a;
basic = b;
}
Salary incr_Salary ( Salary s ) {

s.basic = basic*110/100; return s;
}
}

class Ob_return_Test
{
public static void main(String[] args) {
Salary s1 = new Salary("I100",5000); Salary s2;// A new salary
object
s1 = s1.incr_Salary( s1);

System.out.println("Current Basic Salary is : "+ s2.basic); }
}
```

Output of this program is:

```
Current Basic Salary is: 5500
```

GARBAGE COLLECTION

One of the key features of Java is its garbage-collected heap, which takes care of freeing dynamically allocated memory that is no longer referenced. It shields the substantial complexity of memory allocation and garbage collection from the developer. Because the heap is garbage-collected, Java programmers don't have to explicitly free the allocated memory.

The JVM's heap stores all the objects created by an executing Java program. You know Java's "new" operator allocate memory to object at the time of creation on the heap at run time. Garbage collection is the process of automatically freeing objects that are no longer referenced by the program. This frees the programmer from having to keep track of when to free allocated memory, thereby preventing many potential bugs and thus reduces the programmer's botheration.

The name "garbage collection" implies that objects that are no longer needed by the program are "garbage" and can be thrown away and collects back into the heap. You can see this process as "memory recycling". When the program no longer references an object, the heap space it occupies must be recycled so that the space is available for subsequent new objects. The garbage collector must somehow determine which objects are no longer referenced by the program and make available the heap space occupied by such unreferenced objects.

Advantages and Disadvantages of garbage collection

Giving the job of garbage collection to the JVM has several advantages. First, it can make programmers more productive. Programming in non-garbage-collected languages, the programmer has to spend a lot of time in keeping track of the memory de-allocation problem. In Java, the programmer can use that time more advantageously in doing other work.

A second advantage of garbage collection is that it ensure program integrity. Garbage collection is an important part of Java's security strategy. Java programmers feel free from the fear of accidental crash of the system because JVM is there to take care of memory allocation and de–allocation.

The major disadvantage of a garbage-collected heap is that it adds an overhead that can adversely affect program performance. The JVM has to keep track of objects that are being referenced by the executing program, and free unreferenced objects on the fly. This activity will likely take more CPU time than would have been required if the program explicitly freed unrefined memory. The second disadvantage is the programmers in a garbage-collected environment have less control over the scheduling of CPU time devoted to freeing objects that are no longer needed.

THE FINALIZE () METHOD

Sometimes it is required to take independent resources from some object before the garbage collector takes the object. To perform this operation, a method named finalize () is used.

Finalized () is called by the garbage collector when it determines no more references to the object exist.

Finalized()method has the following properties:

1. Every class inherits the finalize() method from Java.lang.Object.
2. The garbage collector calls this method when it determines no more references to the object exist.
3. The Object class finalize method performs no actions but it may be overridden by any derived class.
4. Normally it should be overridden to clean-up non-Java resources, i.e. closing a file, taking file handle, etc.

Writing a finalize() Method

Before an object is garbage collected, the runtime system calls its finalize() method. The intent for this to release system resources such as open files or open sockets before object getting collected by garbage collector.

Your class can provide for its finalization simply by defining and implementing a finalize() method in your class. Your finalize() method must be declared as follows:

protected void finalize () throws throwable

This class opens a file when its constructed:

```
class OpenAFile
{
FileInputStream aFile = null; OpenAFile(String filename) {
try
{
a File = new FileInputStream(filename);
}
catch (Java.io.FileNotFoundException e)
{
System.err.println("Could not open file " + filename); }
}
}
```

To avoid accidental modification or other related problem the OpenAFile class should close the file when it is finalized. Implementation of finalize () method for the OpenAFile class:

```
 protected void finalize () throws throwable
{
if (aFile != null) {
a File.close();
a File = null;
}
}
```

The Java programmer must keep in mind that it is the garbage collector that runs finalizers on objects. Because it is not generally possible to predict exactly when unreferenced objects will be garbage collected, and to predict when object finalizers will be run. Java programmers, therefore, should avoid writing code for which program correctness depends upon the timely finalization of objects. For example, a finalize of an unreferenced object may release a resource that is needed again later by the program. The resource will not be made available until after the garbage collector has run the object finalizer. If the program needs the resource before the garbage collector has run the finalizer, the program is out of luck.

METHOD OVERLOADING

If a class has multiple methods having same name but different in parameters, it is known as **Method Overloading**.

If we have to perform only one operation, having same name of the methods increases the readability of the program.

Suppose you have to perform addition of the given numbers but there can be any number of arguments, if you write the method such as a(int,int) for two parameters, and b(int,int,int) for three parameters then it may be difficult for you as well as other programmers to understand the behavior of the method because its name differs.

So, we perform method overloading to figure out the program quickly.

Advantage of method overloading

Method overloading *increases the readability of the program*.

Different ways to overload the method

There are two ways to overload the method in java

1. By changing number of arguments
2. By changing the data type

1) Method Overloading: changing no. of arguments

In this example, we have created two methods, first add() method performs addition of two numbers and second add method performs addition of three numbers.

In this example, we are creating static methods so that we don't need to create instance for calling methods.

```
class Adder{
static int add(int a,int b){return a+b;}
static int add(int a,int b,int c){return a+b+c;}
}
class TestOverloading1{
public static void main(String[] args){
System.out.println(Adder.add(11,11));
System.out.println(Adder.add(11,11,11));
}}
```

Output:

22

33

2) Method Overloading: changing data type of arguments

In this example, we have created two methods that differs in data type. The first add method receives two integer arguments and second add method receives two double arguments.

```
class Adder{
static int add(int a, int b){return a+b;}
static double add(double a, double b){return a+b;}
}
class TestOverloading2{
public static void main(String[] args){
System.out.println(Adder.add(11,11));
System.out.println(Adder.add(12.3,12.6));
}}
```

Output:

22

24.9

Why Method Overloading is not possible by changing the return type of method only?

In java, method overloading is not possible by changing the return type of the method only because of ambiguity. Let's see how ambiguity may occur:

```
class Adder{
static int add(int a,int b){return a+b;}
static double add(int a,int b){return a+b;}
```

```
}
class TestOverloading3{
public static void main(String[] args){
System.out.println(Adder.add(11,11));//ambiguity
}}
```

Output:

```
Compile Time Error: method add(int,int) is already defined in class
Adder
```

Can we overload java main() method?

Yes, by method overloading. You can have any number of main methods in a class by method overloading. But JVM calls main() method which receives string array as arguments only. Let's see the simple example:

```
class TestOverloading4{
public static void main(String[] args){
System.out.println("main with String[]");}
public static void main(String args){
System.out.println("main with String");}
public static void main(){
System.out.println("main without args");}
}
```

Output:

```
main with String[]
```

Modifiers in Java

Modifiers are keywords that let us fine-tune access to our class and its members, their scope, and behavior in certain situations. For example, we can control which classes/objects can access certain members of our class, whether a class can be inherited or not, whether we can override a method later, whether we *should* override a method later, etc.

Modifier keywords are written before the variable/method/class (return) type and name, e.g. private int myVar or public String toString().

Modifiers in Java fall into one of two groups - **access** and **non-access**:

- **Access**: public, private, protected, default.
- **Non-access**: static, final, abstract, synchronized, volatile, transient and native.

ACCESS CONTROL

Access control is a mechanism, an attribute of encapsulation which restricts the access of certain members of a class to specific parts of a program. Access to members of a class can be controlled using the access modifiers. There are four access *modifiers* in Java. They are:

1. public
2. protected
3. default
4. private

If the member (variable or method) is not marked as either *public* or *protected* or *private*, the access modifier for that member will be *default*. We can apply access modifiers to classes also.

Among the four access modifiers, *private* is the most restrictive access modifier and *public* is the least restrictive access modifier. Syntax for declaring a access modifier is shown below:

```
access-modifier  data-type  variable-name;
```

Example for declaring a *private* integer variable is shown below:

```
private int side;
```

In a similar way we can apply access modifiers to methods or classes although private classes are less common.

Access Modifiers In Java

Access Modifier	Within the Class	Other Classes [Within the Package]	In Subclasses [Within the package and other packages]	Any Class [In Other Packages]
public	Y	Y	Y	Y
protected	Y	Y	Y	N
default	Y	Y	Same Package – Y Other Packages - N	N
private	Y	N	N	N

Y – Accessible
N – Not Accessible

Example

```
class Employee
{
	int  empid;
	String  empname;
	float  salary;
	//Methods which operate on above data members
}
```

By looking at the above code we can say that the access modifier for all the three data members is *default*. As members with *default* (also known as **Package Private** or **no modifier**) access modifier are accessible throughout the package (in other classes), a programmer might, by mistake, try to make an employee's salary negative as shown below:

```
Employee e1 = new Employee();
e1.salary = -1000.00;
```

Although above code is syntactically correct, it is logically incorrect. To prevent such things to happen, in general, all the data members are declared *private* and are accessible only through *public* methods. So, we can modify our *Employee* class as shown below:

```
class Employee
{
	private int empid;
	private String emp;
	private float salary;
	public void setSalary(float sal)
	{
		if(sal < 0)
		{
			System.out.println("Salary cannot be negative");
		}
		else
		{
			salary = sal;
		}
	}
	//Other methods
}
```

By looking at the above code, we can say that one can access the *salary* field only through *setSalary()* method. Now, we can set the salary of an employee as shown below:

```
Employee e1 = new Employee();
e1.setSalary(-1000.00);  //Gives error as salary cannot be negative
e1.setSalary(2000.00);  //salary of e1 will be assigned 2000.00
```

Non Access Modifiers

These types of modifiers are used to control a variety of things, such as inheritance capabilities, whether all objects of our class share the same member value or have their own values of those members, whether a method can be overridden in a subclass, etc.

A brief overview of these modifiers can be found in the following table:

Modifier Name	Overview
static	The member belongs to the class, not to objects of that class.
final	Variable values can't be changed once assigned, methods can't be overriden, classes can't be inherited.
abstract	If applied to a method - has to be implemented in a subclass, if applied to a class - contains abstract methods
synchronized	Controls thread access to a block/method.
volatile	The variable value is always read from the main memory, not from a specific thread's memory.
transient	The member is skipped when serializing an object.

The static Modifier

The static modifier makes a class member independent of any object of that class. There are a few features to keep in mind here:

- **Variables** declared static are shared among all objects of a class (since the variable essentially belongs to the class itself in this case), i.e. objects don't have their own values for that variable, instead, they all share a single one.
- **Variables and methods** declared static can be accessed via the class name (instead of the usual object reference, e.g. MyClass.staticMethod() or MyClass.staticVariable), *and they can be accessed without the class being instantiated*.
- static methods can only use static variables and call other static methods, and cannot refer to this or super in any way (an object instance might not even exist when we call a static method, so this wouldn't make sense).

Note: It's very important to note that static variables and methods *can't* access non-static (instance) variables and methods. On the other hand, non-static variables and methods *can* access static variables and methods.

This is logical, as static members exist even without an object of that class, whereas *instance* members exist only after a class has been instantiated.

Example

The static modifier is used to create class methods and variables, as in the following example −

```
public class InstanceCounter {
   private static int numInstances = 0;
   protected static int getCount() {
      return numInstances;
   }
   private static void addInstance() {
      numInstances++;
   }
   InstanceCounter() {
      InstanceCounter.addInstance();
   }
   public static void main(String[] arguments) {
      System.out.println("Starting with " + InstanceCounter.getCount() +
" instances");
      for (int i = 0; i < 500; ++i) {
         new InstanceCounter();
      }
      System.out.println("Created  "  +  InstanceCounter.getCount()  +  "
instances");
   }
}
```

This will produce the following result −

Output

```
Started with 0 instances

Created 500 instances
```

The Final Modifier

Final Variables

A final variable can be explicitly initialized only once. A reference variable declared final can never be reassigned to refer to an different object.

However, the data within the object can be changed. So, the state of the object can be changed but not the reference.

With variables, the *final* modifier often is used with *static* to make the constant a class variable.

Example

```
public class Test {
```

```
    final int value = 10;


    // The following are examples of declaring constants:
    public static final int BOXWIDTH = 6;
    static final String TITLE = "Manager";


    public void changeValue() {
       value = 12;   // will give an error
    }
}
```

Final Methods

A final method cannot be overridden by any subclasses. As mentioned previously, the final modifier prevents a method from being modified in a subclass.

The main intention of making a method final would be that the content of the method should not be changed by any outsider.

Example

You declare methods using the final modifier in the class declaration, as in the following example −

```
public class Test {
    public final void changeName() {
       // body of method
    }
}
```

Final Classes

The main purpose of using a class being declared as *final* is to prevent the class from being subclassed. If a class is marked as final then no class can inherit any feature from the final class.

Example

```
public final class Test {
```

```
        // body of class

    }
```

Java Nested and Inner classes

Java inner class or nested class is a class that is declared inside the class or interface.

We use inner classes to logically group classes and interfaces in one place to be more readable and maintainable.

Additionally, it can access all the members of the outer class, including private data members and methods.

Syntax of Inner class

```
class Java_Outer_class{

 //code

 class Java_Inner_class{

  //code

 }

}
```

Advantage of Java inner classes

There are three advantages of inner classes in Java. They are as follows:

- Nested classes represent a particular type of relationship that is it can access all the members (data members and methods) of the outer class, including private.
- Nested classes are used to develop more readable and maintainable code because it logically group classes and interfaces in one place only.
- Code Optimization: It requires less code to write.

Need of Java Inner class

Sometimes users need to program a class in such a way so that no other class can access it. Therefore, it would be better if you include it within other classes.

If all the class objects are a part of the outer object then it is easier to nest that class inside the outer class. That way all the outer class can access all the objects of the inner class.

Difference between nested class and inner class in Java

An inner class is a part of a nested class. Non-static nested classes are known as inner classes.

Types of Nested classes

There are two types of nested classes non-static and static nested classes. The non-static nested classes are also known as inner classes.

- Non-static nested class (inner class)
- Member inner class
- Anonymous inner class
- Local inner class
- Static nested class

	Type	Description
Non-static nested class (inner class)	Member Inner Class	A class created within class and outside method.
	Anonymous Inner Class	A class created for implementing an interface or extending class. The java compiler decides its name.
	Local Inner Class	A class was created within the method.
Static nested class	Static Nested Class	A static class was created within the class.
	Nested Interface	An interface created within class or interface.

command line arguments

The main method (if available) inside a Java class accepts an array of Strings arguments. These arguments come from contents provided when users execute the java command to run the class. For example, given the following CLA.java program:

```
public class CLA {
  public static void main(String[] args) {
    if (args.length == 0) {
      System.out.println("The program is not run with any command
line arguments");
    } else {
      for (int i = 0; i < args.length; i++) {
```

```
            System.out.println("The " + (i+1) + "th argument of args is
    at position " + i + " of the args array and has value: "
    +  args[i]);
          }
        }
      }
    }
```

variable - length arguments

The String[] args parameter of main can potentially accept an unlimited number of arguments. This mechanism, called variable-length arguments, can be applied for other methods as well by using the ... notation for parameters in method declaration.

```
public class VarArgs {
  public static void printArgs(String... values) {
    System.out.println("Run printArgs ... ");
    for (int i = 0; i < values.length; i++) {
      System.out.println(values[i]);
    }
  }
  public static void main(String[] args) {
    printArgs("rams");
    printArgs("golden","rams");
    printArgs("golden","rams","wcupa");
    printArgs("golden","rams","wcupa","university");
  }
}
```

****