JAVA PROGRAMMING LANGUAGE

PLACEMENT PREPARATION [EXCLUSIVE NOTES]

SAVE AND SHARE

Curated By- HIMANSHU KUMAR(LINKEDIN)

TOPICS COVERED-

PART -2 :-

- Loops in Java
- Methods in Java
- Sample Problems|(Decision, Loops,Arrays & Strings)
- Arrays in Java
- Strings in Java
- Immutable Strings in Java
- BigInteger Class in Java
- ArrayList in Java

PART-1 LINK :-

https://www.linkedin.com/posts/himanshukumarmahuri_java-programming-language-activity-6941257412381134848-rF1_?utm_source=linkedin_share&utm_medium=android_app

HIMANSHU KUMAR(LINKEDIN)

https://www.linkedin.com/in/himanshukumarmahuri

Part-2 :-

1.Loops in Java-

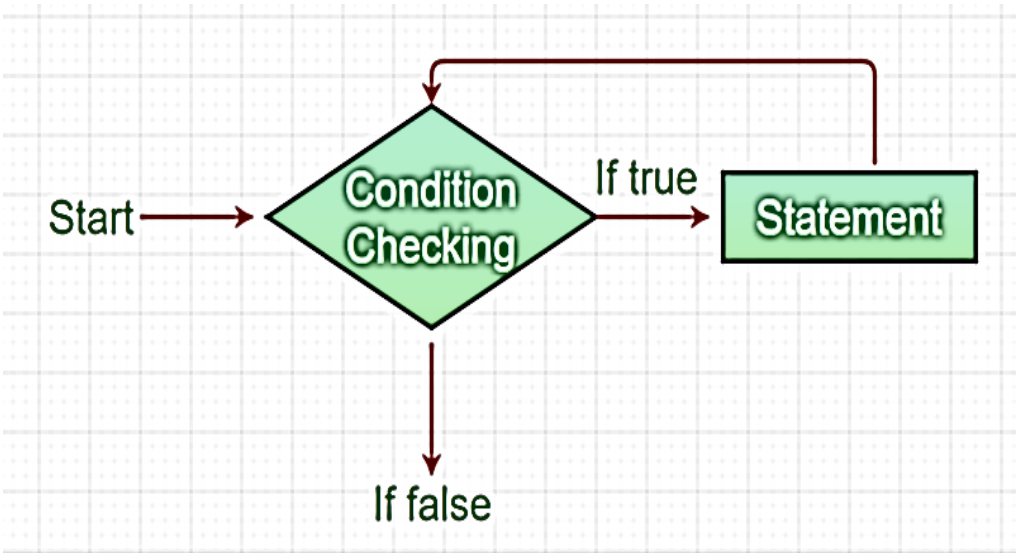
Looping in programming languages is a feature that facilitates the execution of a set of instructions/functions repeatedly while some condition evaluates to true.

Java provides three ways of executing the loops. While all the ways provide similar basic functionality, they differ in their syntax and condition checking the time.

1. **while loop:** A while loop is a control flow statement that allows code to be executed repeatedly based on a given boolean condition. The while loop can be thought of as a repeating if statement.
   Syntax :

```
while (boolean condition)
{
   loop statements...
}
```

Flowchart:



- While loop starts with the checking of condition. If it evaluated to true, then the loop body statements are executed otherwise the first statement following the loop is executed. For this reason, it is also called **Entry control loop**
- Once the condition is evaluated to true, the statements in the loop body are executed. Normally, the statements contain an update value for the variable being processed for the next iteration.
- When the condition becomes false, the loop terminates which marks the end of its life cycle.

// Java program to illustrate while loop

```
class whileLoopDemo

{

   public static void main(String args[])

   {

      int x = 1;

      // Exit when x becomes greater than 4

      while (x <= 4)

      {

         System.out.println("Value of x:" + x);

         // Increment the value of x for

         // next iteration

         x++;

      }

   }

}
```

Output:

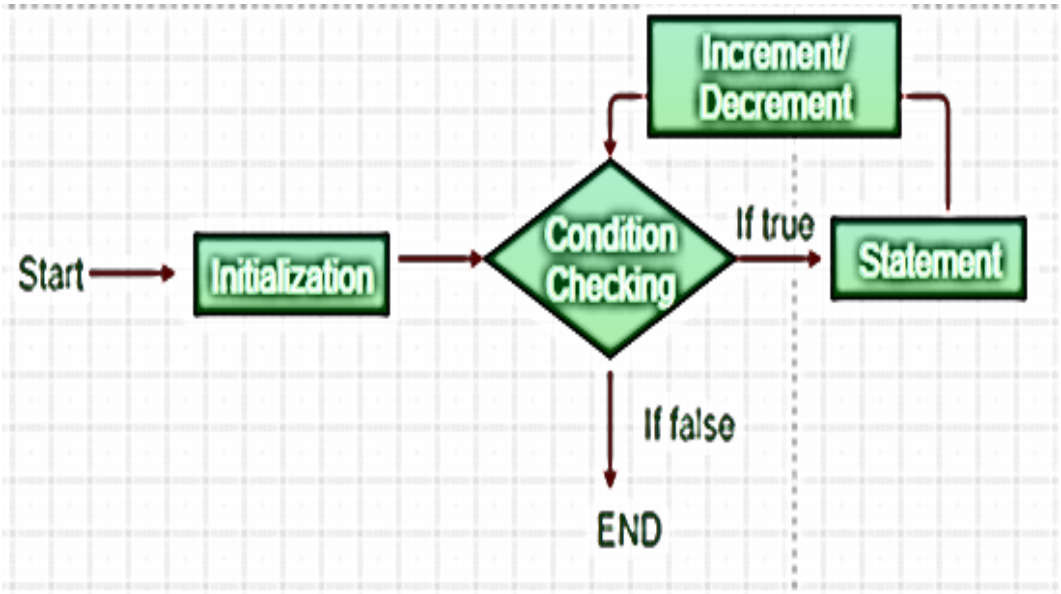
```
Value of x:1
Value of x:2
Value of x:3
Value of x:4
```

for loop: for loop provides a concise way of writing the loop structure. Unlike a while loop, a for statement consumes the initialization, condition, and increment/ decrement in one line, thereby providing a shorter and a easy way to debug the structure of the loop.

Syntax:

```
for (initialization condition; testing condition;
                                increment/decrement)
{
    statement(s)
}
```

Flowchart:



Initialization condition: Here, we initialize the variable in use. It marks the start of a for loop. An already declared variable can be used or a variable can be declared, local to loop only.

Testing Condition: It is used for testing the exit condition for a loop. It must return a boolean value. It is also an **Entry Control Loop** as the condition is checked prior to the execution of the loop statements.

Statement execution: Once the condition is evaluated to true, the statements in the loop body are executed.

Increment/ Decrement: It is used for updating the variable for next iteration.

Loop termination: When the condition becomes false, the loop terminates marking the end of its life cycle.

// Java program to illustrate for loop.

```
class forLoopDemo

{

   public static void main(String args[])

   {

      // for loop begins when x=2

      // and runs till x <=4

      for (int x = 2; x <= 4; x++)

         System.out.println("Value of x:" + x);

   }

}
```

Output:

```
Value of x:2
Value of x:3
Value of x:4
```

Enhanced For loop

Java also includes another version of for loop introduced in Java 5. Enhanced for loop provides a simpler way to iterate through the elements of a collection or an array. It is inflexible and should be used only when there is a need to iterate through the elements in a sequential manner without knowing the index of the currently processed element.

Also, note that the object/variable is immutable when enhanced for loop is used i.e., it ensures that the values in the array can not be modified, so it can be said as a read-only loop where you can't update the values as opposed to other loops where values can be modified.

We recommend using this form of the for statement instead of the general form whenever possible.(as per JAVA doc.)

Syntax:

```
for (T element:Collection obj/array)
{
    statement(s)
}
```

Let's take an example to demonstrate how enhanced for loop can be used to simplify the work. Suppose there is an array of names and we want to print all the names in that array. Let's see the difference between these two examples.

Enhanced for loop simplifies the work as follows-

// Java program to illustrate enhanced for loop

```
public class enhancedforloop
{
   public static void main(String args[])
   {
      String array[] = {"Ron", "Harry",
"Hermoine"};
      //enhanced for loop
      for (String x:array)
      {
         System.out.println(x);
      }
      /* for loop for same function
      for (int i = 0; i < array.length; i++)
      {
         System.out.println(array[i]);
      }
      */
   }
```

```
}
```

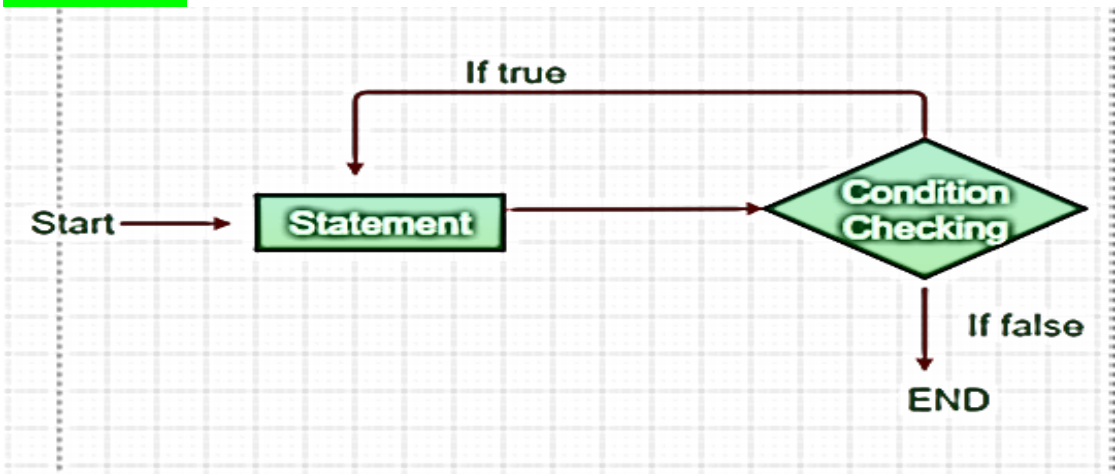
Output:

```
Ron
Harry
Hermoine
```

1. **do while:** do while loop is similar to while loop with only difference that it checks for condition after executing the statements, and therefore is an example of **Exit Control Loop. Syntax:**

```
do
{
    statements..
}
while (condition);
```

Flowchart:



do while loop starts with the execution of the statement(s). There is no checking of any condition for the first time.

After the execution of the statements and the updation of the variable value, the condition is checked for true or false value. If it is evaluated to true, next iteration of loop starts.

When the condition becomes false, the loop terminates which marks the end of its life cycle.

It is important to note that the do-while loop will execute its statements atleast once before any condition is checked, and therefore is an example of exit control loop.

// Java program to illustrate do-while loop

```
class dowhileloopDemo

{

   public static void main(String args[])

   {

      int x = 21;

      do

      {

         // The line will be printed even

         // if the condition is false

         System.out.println("Value of x:" + x);

         x++;

      }

      while (x < 20);

   }

}
```

Output:

```
Value of x: 21
```

For Each Loop: For-each is another array traversing technique such as for loop, while loop, and do-while loop introduced in Java5.

- It starts with the keyword **for** like a normal for-loop.
- Instead of declaring and initializing a loop counter variable, you declare a variable that is of the same type as the base type of the array, followed by a colon, which is then followed by the array name.
- In the loop body, you can use the loop variable you created rather than using an indexed array element.
- It's commonly used to iterate over an array or a Collections class (eg, ArrayList)

Syntax:

```
for (type var : array)
{
    statements using var;
}
```

is equivalent to:

```
for (int i=0; i<arr.length; i++)
{
    type var = arr[i];
    statements using var;
}
```

// Java program to illustrate

// for-each loop

```
class For_Each

{

   public static void main(String[] arg)

   {

      {
```

```
        int[] marks = { 125, 132, 95, 116, 110 };
        int highest_marks = maximum(marks);
        System.out.println("The highest score is " + highest_marks);
    }
  }
  public static int maximum(int[] numbers)
  {
    int maxSoFar = numbers[0];
    // for each loop
    for (int num : numbers)
    {
      if (num > maxSoFar)
      {
        maxSoFar = num;
      }
    }
  return maxSoFar;
  }
}
```

Output:

```
The highest score is 132
```

Pitfalls of Loops

1. **Infinite loop:** One of the most common mistakes while implementing any sort of looping is that the loop may not ever exit, i.e., the loop runs for an infinite time. This happens when the condition fails for some reason.
   Examples:

```
//Java program to illustrate various pitfalls.
public class LooppitfallsDemo
{
   public static void main(String[] args)
   {
      // infinite loop because condition is not apt
      // condition should have been i>0.
      for (int i = 5; i != 0; i -= 2)
      {
         System.out.println(i);
      }
      int x = 5;
      // infinite loop because update statement
      // is not provided.
      while (x == 5)
      {
         System.out.println("In the loop");
      }
   }
}
```

Another pitfall is that you might be adding something into your collection object through the loop, and you run out of memory. If you try and execute the below program, after some time, the out of memory exception will be thrown.

//Java program for out of memory exception.

```
import java.util.ArrayList;

public class Integer1

{

   public static void main(String[] args)

   {

      ArrayList<Integer> ar = new ArrayList<>();

      for (int i = 0; i < Integer.MAX_VALUE; i++)

      {

         ar.add(i);

      }

   }

}
```

Output:

```
Exception in thread "main" java.lang.OutOfMemoryError:
Java heap space
at java.util.Arrays.copyOf(Unknown Source)
at java.util.Arrays.copyOf(Unknown Source)
at java.util.ArrayList.grow(Unknown Source)
at java.util.ArrayList.ensureCapacityInternal(Unknown
Source)
at java.util.ArrayList.add(Unknown Source)
```

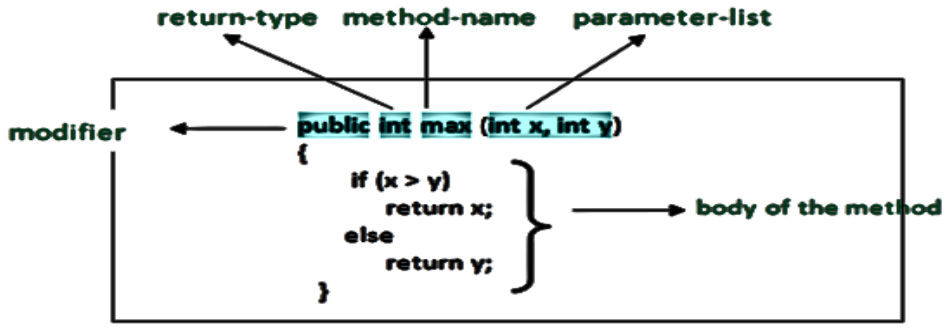
2.Methods in Java-

A method is a collection of statements that perform some specific task and return the result to the caller. A method can perform some specific task without returning anything. Methods allow us to **reuse** the code without retyping the code. In Java, every method must be a part of some class which is different from languages such as C, C++, and Python.
Methods are **time savers** and help us to **reuse** the code without retyping the code.

Method Declaration

In general, a method declaration has six components :

- **Modifier**-: Defines **the access type** of the method i.e., from where it can be accessed in your application. In Java, there are 4 types of access specifiers:
  - public: accessible in all the classes in your application.
  - protected: accessible within the class in which it is defined and in its **subclass(es)**
  - private: accessible only within the class in which it is defined.
  - default (declared/defined without using any modifier): accessible within the same class and the package within which its class is defined.
- **The return type** : The data type of the value returned by the method, or void if does not return a value.
- **Method Name** : The rules for field names apply to method names as well, but the convention is a little different.
- **Parameter list** : Comma-separated list of the input parameters are defined, preceded with their data type, within the enclosed parenthesis. If there are no parameters, you must use empty parentheses ().
- **Exception list** : The exceptions you expect that the method can throw, you can specify these exception(s).
- **Method body** : It is enclosed between the braces. It contains the code that is needed to be executed to perform the intended operations.



Method signature: It consists of the method name and a parameter list (number of parameters, type of the parameters, and order of the parameters). The return type and exceptions are not considered as a part of it.
Method Signature of above function:

```
max(int x, int y)
```

How to name a Method?:

A method name is typically a single word that should be a **verb** in lowercase or multi-word, which begins with a **verb** in lowercase followed by **adjective, noun.....** After the first word, the first letter of each word should be capitalized. For example, findSum, computeMax, setX and getX.

Generally, A method has a unique name within the class in which it is defined but sometimes a method might have the same name as another method's name within the same class as method overloading is allowed in Java.

Calling a method

The method needs to be called for using its functionality. There can be three situations when a method is called:
A method returns to the code that invoked it when:

- It completes all the statements in the method
- It reaches a return statement
- Throws an exception

// Program to illustrate methods in java

```
import java.io.*;

class Addition {

   int sum = 0;

   public int addTwoInt(int a, int b){

      // adding two integer value.

      sum = a + b;

      //returning summation of two values.
```

```
        return sum;
    }
}
class DEMO {
    public static void main (String[] args) {
        // creating an instance of Addition class
        Addition add = new Addition();
        // calling addTwoInt() method to add two integer using instance created
        // in above step.
        int s = add.addTwoInt(1,2);
        System.out.println("Sum of two integer values :"+ s);
    }
}
```

Output :

```
Sum of two integer values :3
```

3.Java Sample Problems | (Decision, Loops, Arrays and Strings)-

The following are some of the basic implementation problems covering the topics discussed until now:

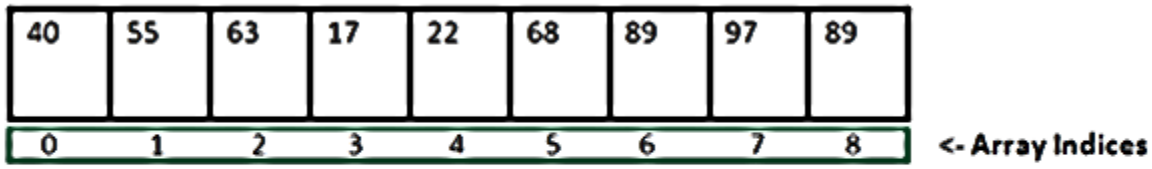
- **Problem 1:** Count the number of digits in a number
- **Problem 2 :** Take as input n numbers and find the average
- **Problem 3 :** Check if a string entered is a palindrome or not. (A palindrome is a string that is the same as the original when reversed)
- **Problem 4:** Convert decimal number input by the user to binary.
- **CHECK IMPLEMENTATION ON GOOGLE**

4. Arrays in Java-

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

- In Java all arrays are dynamically allocated.(discussed below)
- Since arrays are objects in Java, we can find their length using member 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 have 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 an int value and not long or short.

An array can contain primitive data types as well as objects 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 case of objects of a class, the actual objects are stored in heap segment.



Array Length = 9
First Index = 0
Last Index = 8

Creating, Initializing, and Accessing an Array

One-Dimensional Arrays : The general form of a one-dimensional array declaration is

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

An array declaration has two components: the type and the name. *type* declares the element type of the array. The element type determines the data type of each element that comprises the array. Like an array of int type, we can also create an array of other primitive data types such as char, float, double, etc., or user defined data type(objects of a class). Thus, the element type for the array determines what type of data the array will hold.

Example:

```
// both are valid declarations
int intArray[];
or int[] intArray;

byte byteArray[];
short shortsArray[];
boolean booleanArray[];
long longArray[];
float floatArray[];
double doubleArray[];
char charArray[];

// an array of references to objects of
// the class MyClass (a class created by
// user)
MyClass myClassArray[];

Object[]  ao,        // array of Object
Collection[] ca;  // array of Collection
                     // of unknown type
```

Although the above first declaration establishes the fact that intArray is an array variable, **no array actually exists**. It simply tells to the compiler that this(intArray) variable will hold an array of the integer type. To link intArray with an actual, physical array of integers, you must allocate one using **new** and assign it to intArray.

Instantiating an Array in Java

When an array is declared, only a reference of array is created. To actually create or give memory to an array, you should create an array like this: The general form of *new* as it applies to one-dimensional arrays appears as follows:

```
var-name = new type [size];
```

Here, *type* specifies the type of data being allocated, *size* specifies the number of elements in the array, and *var-name* is the name of array variable that is linked to the array. That is, to use *new* to allocate an array, **you must specify the type and number of elements to allocate.**

Example:

```
int intArray[];    //declaring array
intArray = new int[20];  // allocating memory to array
```

OR

```
int[] intArray = new int[20]; // combining both statements
in one
```

Note :

The elements in the array allocated by *new* will automatically be initialized to **zero** (for numeric types), **false** (for boolean), or **null** (for reference types).

Obtaining an array is a two-step process. First, you must declare a variable of the desired array type. Second, you must allocate the memory that will hold the array, using new, and assign it to the array variable. Thus, **in Java all arrays are dynamically allocated.**

Array Literal

In a situation in which the size of the array and variables of the array are already known, array literals can be used.

```
int[] intArray = new int[]{ 1,2,3,4,5,6,7,8,9,10 };

 // Declaring array literal
```

- The length of this array determines the length of the created array.
- There is no need to write the new int[] part in the latest versions of Java

Accessing Java Array Elements using for Loop

Each element in the array is accessed via its index. The index begins with 0 and ends at (total array size)-1. All the elements of array can be accessed using Java for Loop.

```
// accessing the elements of the specified array

for (int i = 0; i < arr.length; i++)

    System.out.println("Element at index " + i + " : "+ ar
r[i]);
```

Implementation:

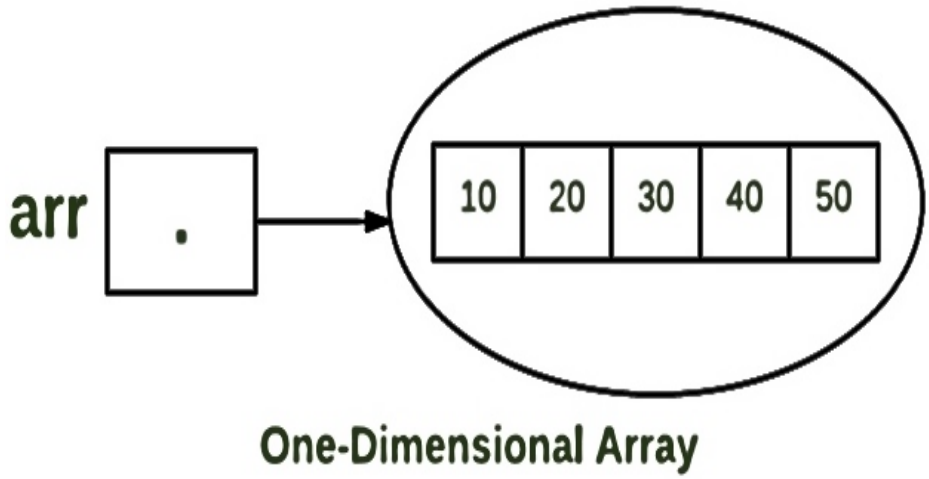
// Java program to illustrate creating an array
// of integers, puts some values in the array,
// and prints each value to standard output.

```
class DEMO
{
   public static void main (String[] args)
   {
     // declares an Array of integers.
     int[] arr;

     // allocating memory for 5 integers.
     arr = new int[5];

     // initialize the first elements of the array
     arr[0] = 10;

     // initialize the second elements of the array
     arr[1] = 20;

     //so on...
     arr[2] = 30;
     arr[3] = 40;
     arr[4] = 50;

     // accessing the elements of the specified array
     for (int i = 0; i < arr.length; i++)
        System.out.println("Element at index " + i +
                           " : "+ arr[i]);
   }
}
```

Output:

```
Element at index 0 : 10
Element at index 1 : 20
Element at index 2 : 30
Element at index 3 : 40
Element at index 4 : 50
```

You can also access java arrays using for each loops



One-Dimensional Array

Arrays of Objects

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

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

The studentArray contains seven memory spaces each of the size of the 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];
```

// Java program to illustrate creating an array of

// objects

```
class Student
{
   public int roll_no;
   public String name;
   Student(int roll_no, String name)
   {
      this.roll_no = roll_no;
      this.name = name;
   }
}
```

```
// Elements of array are objects of a class Student.
public class DEMO
{
   public static void main (String[] args)
   {
      // declares an Array of integers.
      Student[] arr;

      // allocating memory for 5 objects of type Student.
      arr = new Student[5];

      // initialize the first elements of the array
      arr[0] = new Student(1,"aman");

      // initialize the second elements of the array
      arr[1] = new Student(2,"vaibhav");

      // so on...
      arr[2] = new Student(3,"shikar");
      arr[3] = new Student(4,"dharmesh");
      arr[4] = new Student(5,"mohit");

      // accessing the elements of the specified array
      for (int i = 0; i < arr.length; i++)
         System.out.println("Element at " + i + " : " +
                  arr[i].roll_no +" "+ arr[i].name);
   }
}
```

Output:

```
Element at 0 : 1 aman
Element at 1 : 2 vaibhav
Element at 2 : 3 shikar
Element at 3 : 4 dharmesh
Element at 4 : 5 mohit
```

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

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

```
class DEMO
{
   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.ArrayIndexOutOfBounds
Exception:
```

Output:

```
10
20
```

Multidimensional Arrays

Multidimensional arrays are **arrays of arrays** with each element of the array holding the reference of other array. These are also known as JAGGED ARRAY. A multidimensional array is created by appending one set of square brackets ([]) per dimension.

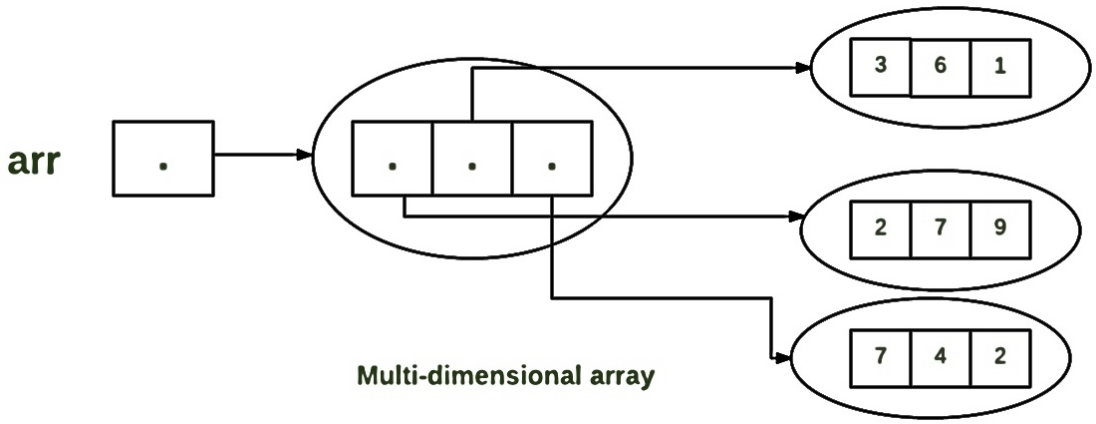
Examples:

```
int[][] intArray = new int[10][20]; //a 2D array or matrix
int[][][] intArray = new int[10][20][10]; //a 3D array
```

```
class multiDimensional

    {
       public static void main(String args[])
       {
          // declaring and initializing 2D array
          int arr[][] = { {2,7,9},{3,6,1},{7,4,2} };

          // printing 2D array
          for (int i=0; i< 3 ; i++)
          {
             for (int j=0; j < 3 ; j++)
                System.out.print(arr[i][j] + " ");

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

Output:

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



Passing Arrays to Methods

Like variables, we can also pass arrays to methods. For example, the below program passes the array to method *sum* for calculating the sum of the array's values.

```java
// Java program to demonstrate
// passing of array to method
class Test
{
    // Driver method
    public static void main(String args[])
    {
        int arr[] = {3, 1, 2, 5, 4};

        // passing array to method m1
        sum(arr);

    }
    public static void sum(int[] arr)
    {
        // getting sum of array values
        int sum = 0;

        for (int i = 0; i < arr.length; i++)
            sum+=arr[i];

        System.out.println("sum of array values : " + sum);
    }
}
```

// Java program to demonstrate
// passing of array to method
class Test
{
   // Driver method
   public static void main(String args[])
   {
      int arr[] = {3, 1, 2, 5, 4};

      // passing array to method m1
      sum(arr);
   }
   public static void sum(int[] arr)
   {
      // getting sum of array values
      int sum = 0;
      for (int i = 0; i < arr.length; i++)
         sum+=arr[i];
      System.out.println("sum of array values : " + sum);
   }
}

Output :

```
sum of array values : 15
```

Returning Arrays from Methods

As usual, a method can also return an array. For example, below program returns an array from method *m1*.

```
// Java program to demonstrate
// return of array from method
class Test
{
   // Driver method
   public static void main(String args[])
   {
      int arr[] = m1();

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

   }
   public static int[] m1()
   {
      // returning  array
      return new int[]{1,2,3};
   }
}
```

Output:

```
1 2 3
```

Cloning of arrays

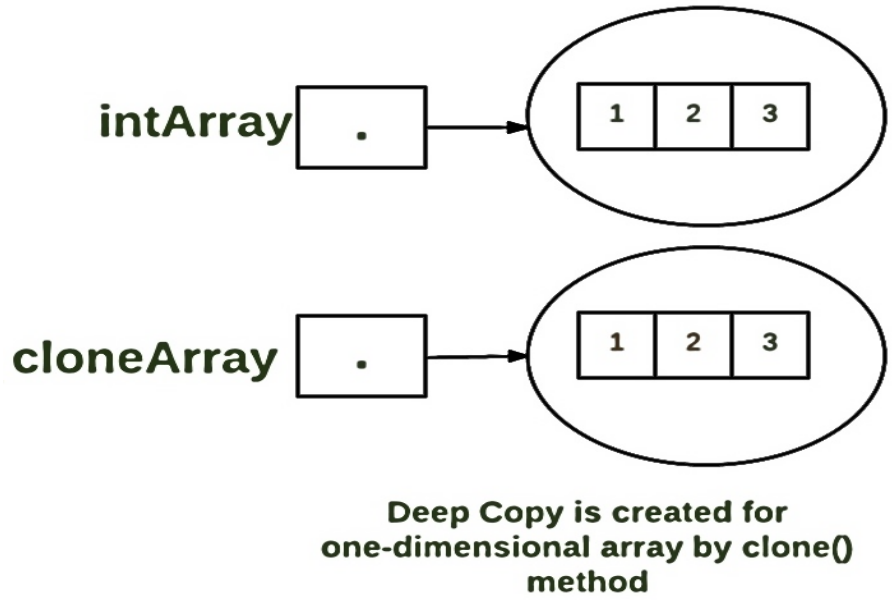
- When you clone a single dimensional array, such as Object[], a "deep copy" is performed with the new array containing copies of the original array's elements as opposed to references.

```
// Java program to demonstrate
// cloning of one-dimensional arrays
```

```
class Test
{
   public static void main(String args[])
   {
      int intArray[] = {1,2,3};

      int cloneArray[] = intArray.clone();

      // will print false as deep copy is created
      // for one-dimensional array
      System.out.println(intArray == cloneArray);

      for (int i = 0; i < cloneArray.length; i++) {
         System.out.print(cloneArray[i]+" "); }}}
```

Output:

```
false
1 2 3
```



Deep Copy is created for one-dimensional array by clone() method

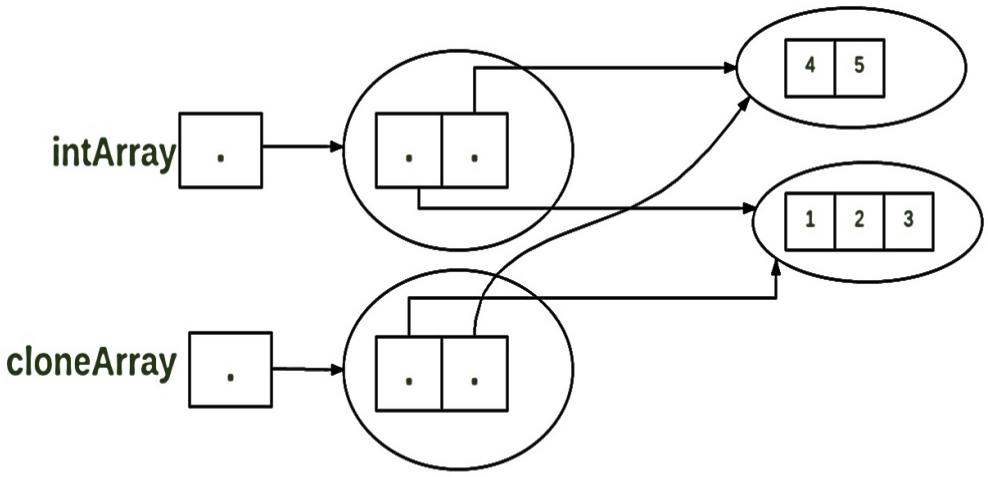
A clone of a multidimensional array (like Object[][]) is a "shallow copy" however, which is to say that it creates only a single new array with each element array as a reference to an original element array but subarrays are shared.

// Java program to demonstrate
// cloning of multi-dimensional arrays

```
class Test
{
   public static void main(String args[])
   {
      int intArray[][] = {{1,2,3},{4,5}};

      int cloneArray[][] = intArray.clone();

      // will print false
      System.out.println(intArray == cloneArray);

      // will print true as shallow copy is created
      // i.e. sub-arrays are shared
      System.out.println(intArray[0] == cloneArray[0]);
      System.out.println(intArray[1] == cloneArray[1]); }
}
```

Output:

```
false
true
true
```



Shallow Copy is created for multi-dimensional array by clone() method

5.Strings in Java-

Strings are defined as an array of characters. The difference between a character array and a string is that the string is terminated with a special character.

Below is the basic syntax for declaring a string in **Java programming** language.

Syntax:

String *stringVariableName* = "sequence_of_characters";

Example:

```
String str = "Geeks";
```

Memory allotment of String

Whenever a String Object is created, two objects are created- one in the Heap Area and one in the String constant pool, and the String object reference always points to the heap area object.

For example:

```
String str = "Geeks";
```

Example to illustrate how to declare String:

```
// Java code to illustrate String

import java.io.*;
import java.lang.*;

class Test {
    public static void main(String[] args)
    {
        // Declare String without using new operator
        String s = "himanshukumar";

        // Prints the String.
        System.out.println("String s = " + s);

        // Declare String using new operator
        String s1 = new String("himanshukumar");
```

```
        // Prints the String.
        System.out.println("String s1 = " + s1);
    }
}
```

Output:

```
String s = himanshukumar
String s1 = himanshukumar
```

Interfaces and Classes in Strings in Java

- CharBuffer: This class implements the CharSequence interface. This class is used to allow character buffers to be used in place of CharSequences. An example of such usage is the regular-expression package java.util.regex.
- String: String is a sequence of characters. In java, objects of String are immutable, which means that they are a constant and cannot be changed once created.

Creating a String

There are two ways to create string in Java:

- ***String literal***

```
String s = “himanshukumar”;
```

- **Using *new* keyword**

```
String s = new String (“himanshukumar”);
```

- **StringBuffer: StringBuffer** is a peer class of **String** that provides much of the functionality of strings. String represents fixed-length, immutable character sequences while StringBuffer represents growable and writable character sequences.

Syntax:

```
StringBuffer s = new StringBuffer("GeeksforGeeks");
```

- **StringBuilder**: The **StringBuilder** in Java represents a mutable sequence of characters. Since the String Class in Java creates an immutable sequence of characters, the StringBuilder class provides an alternate to String Class as it creates a mutable sequence of characters.

  Syntax:

```
StringBuilder str = new StringBuilder();
```

```
str.append("GFG");
```

- **StringTokenizer**: StringTokenizer class in Java is used to break a string into tokens.

  Example:
  A StringTokenizer object internally maintains a current position within the string to be tokenized. Some operations advance this current position past the characters processed.
  A token is returned by taking a substring of the string that was used to create the StringTokenizer object.

- **StringJoiner:** StringJoiner is a class in java.util package which is used to construct a sequence of characters(strings) separated by a delimiter and optionally starting with a supplied prefix and ending with a supplied suffix. Though this can also be done with the help of the StringBuilder class, StringJoiner provides an easy way to do so without writing much code.

  Syntax:

```
public StringJoiner(CharSequence delimiter)
```

Compare two Strings in Java

String is a sequence of characters. In Java, objects of String are immutable which means they are constant and cannot be changed once created.

Below are 5 ways to compare two Strings in Java:

1. **Using user-defined function :** Define a function to compare values with following conditions :
   0. if (string1 > string2) it returns a **positive value**.
   1. if both the strings are equal lexicographically i.e.(string1 == string2) it returns **0**.
   2. if (string1 < string2) it returns a **negative value**.

The value is calculated as **(int)str1.charAt(i) - (int)str2.charAt(i)**
Examples:

```
Input 1: GeeksforGeeks
Input 2: Practice
Output: -9

Input 1: Geeks
Input 2: Geeks
Output: 0

Input 1: GeeksforGeeks
Input 2: Geeks
Output: 8
```

Program:

```
// Java program to compare two strings
// lexicographically
```

Must checkout program on google

1. **Using String.equals():** In Java, string equals() method compares the two given strings based on the data/content of the string. If all the contents of both the strings are the same then it returns true. If all characters do not match, then it returns false.
   Syntax:

```
str1.equals(str2);
```

Here str1 and str2 both are the strings that are to be compared.

Examples:

```
Input 1: testoftest
Input 2: Practice
Output: false

Input 1: test
Input 2: test
Output: true

Input 1: test
Input 2: Test
Output: false
```

Program:

Must checkout the program on google

1. **Using String.equalsIgnoreCase() :** The **String.equalsIgnoreCase()** method compares two strings irrespective of the case (lower or upper) of the string. This method returns true if the argument is not null and the contents of both the Strings are same, ignoring case, else it returns false.

Syntax:

```
str2.equalsIgnoreCase(str1);
```

Here str1 and str2 both are the strings that are to be compared.

Examples:

```
Input 1: GeeksforGeeks
Input 2: Practice
Output: false

Input 1: Geeks
Input 2: Geeks
Output: true

Input 1: geeks
Input 2: Geeks
Output: true
```

Program:

Must checkout the program on google

1. method returns true if the arguments are equal to each other and false otherwise. Consequently, if both arguments are null, true is returned and if exactly one argument is null, false is returned. Otherwise, equality is determined by using the equals() method of the first argument.

Syntax:

```
public static boolean equals(Object a, Object b)
```

Here a and b both are the string objects which are to be compared.

Examples:

```
Input 1: GeeksforGeeks
Input 2: Practice
Output: false

Input 1:Test
Input 2: Test
Output: true

Input 1: null
Input 2: null
Output: true
```

1. **Using String.compareTo()** :
   Syntax:

```
int str1.compareTo(String str2)
```

Working: It compares and returns the following values as follows:

0. if (string1 > string2), it returns a **positive value**.
1. if both the strings are equal lexicographically i.e.(string1 == string2), it returns **0**.
2. if (string1 < string2), it returns a **negative value**.

Examples:

```
Input 1: GeeksforGeeks
Input 2: Practice
Output: -9

Input 1: Geeks
Input 2: Geeks
Output: 0

Input 1: GeeksforGeeks
Input 2: Geeks
Output: 8
```

Program:

Must checkout the program on google

Why not to use == for comparison of Strings?

In general both **equals()** and "**==**" operator in Java are used to compare objects to check equality but here are some of the differences between the two:

- Main difference between .equals() method and == operator is that one is a method and other is a operator.
- One can use == operators for reference comparison **(address comparison)** and .equals() method for **content comparison**.

In simple words, **== checks whether both objects point to the same memory location whereas .equals() evaluates the comparison of values in the objects.**
Example:

```
// Java program to understand
// why to avoid == operator

public class Test {
    public static void main(String[] args)
    {
        String s1 = new String("HELLO");
        String s2 = new String("HELLO");
        System.out.println(s1 == s2);
        System.out.println(s1.equals(s2));
    }
}
```

Output:

```
false
true
```

Explanation:
Here two String objects are being created namely s1 and s2.
Both s1 and s2 refers to different objects.
When one uses == operator for s1 and s2 comparison then the result is false as both have different addresses in memory.
Using equals, the result is true because it's only comparing the values given in s1 and s2.

6.Immutable Strings in Java-

Before discussing the concept of string **immutability**, let's just take a look into the String class and its functionality a little before coming to any conclusion.

This is how String works:

```
String str = "knowledge";
```

This, as usual, creates a string containing "knowledge" and assigns it a reference str. Simple enough? Lets perform some more functions:

```
// assigns a new reference to the
// same string "knowledge"
String s = str;
```

Let's see how the below statement works:

```
str = str.concat(" base");
```

This appends a string " base" to str. But wait, how is this possible, since String objects are immutable? Well to your surprise, it is.

When the above statement is executed, the VM takes the value of String str, i.e. "knowledge" and appends " base", giving us the value "knowledge base". Now, since Strings are immutable, the VM can't assign this value to str, so it creates a new String object, gives it a value "knowledge base", and gives it a reference str.

An important point to note here is that, while the String object is immutable, **its reference variable is not.** So that's why, in the above example, the reference was made to refer to a newly formed String object.

At this point in the example above, we have two String objects: the first one we created with value "knowledge", pointed to by s, and the second one "knowledge base", pointed to by str. But, technically, we have three String objects, the third one being the literal "base" in the concat statement.

Important Facts about String and Memory usage

What if we didn't have another reference s to "knowledge"? We would have lost that String. However, it still would have existed, but would be considered lost due to having no references.
Look at one more example below

```
/*package whatever // do not write package name here */

import java.io.*;

class DEMO {

   public static void main(String[] args)

   {

      String s1 = "java";

      s1.concat(" rules");

      // Yes, s1 still refers to "java"

      System.out.println("s1 refers to " + s1);

   }

}
```

Output:

```
s1 refers to java
```

What's happening:

1. The first line is pretty straightforward: create a new `String` `"java"` and refer `s1` to it.
2. Next, the VM creates another new `String` `"java rules"`, but nothing refers to it. So, the second `String` is instantly lost. We can't reach it.

The reference variable `s1` still refers to the original `String` `"java"`.

Almost every method, applied to a `String` object in order to modify it, creates new `String` object. So, where do these `String` objects go? Well, *these exist in memory, and one of the key goals of any programming language is to make efficient use of memory.*

As applications grow, *it's very common for `String` literals to occupy large area of memory, which can even cause redundancy.* So, in order to make Java more efficient, **the JVM sets aside a special area of memory called the "String constant pool".**

When the compiler sees a `String` literal, it looks for the `String` in the pool. If a match is found, the reference to the new literal is directed to the existing `String` and no

new String object is created. The existing String simply has one more reference. Here comes the point of making String objects immutable:

In the String constant pool, a String object is likely to have one or more references. *If several references point to the same String without even knowing it, it would be bad if one of the references modify that String value. That's why String objects are immutable.*

Well, now you could say, *what if someone overrides the functionality of String class?* That's the reason that **the String class is marked final** so that nobody can override the behavior of its methods.

7.BigInteger Class in Java-

BigInteger class is used for the mathematical operation which involves very big integer calculations that are outside the limit of all available primitive data types.

For example factorial of 100 contains 158 digits in it so we can't store it in any primitive data type available. We can store as large Integer as we want in it. There is no theoretical limit on the upper bound of the range because memory is allocated dynamically but practically as memory is limited you can store a number which has Integer.MAX_VALUE number of bits in it which should be sufficient to store mostly all large values.

Below is an example Java program that uses BigInteger to compute Factorial.

```
// Java program to find large factorials using BigInteger
import java.math.BigInteger;
import java.util.Scanner;

public class Example
{
   // Returns Factorial of N
   static BigInteger factorial(int N)
   {
      // Initialize result
      BigInteger f = new BigInteger("1"); // Or BigInteger.ONE

      // Multiply f with 2, 3, ...N
      for (int i = 2; i <= N; i++)
         f = f.multiply(BigInteger.valueOf(i));
```

```
        return f;
    }
    // Driver method
    public static void main(String args[]) throws Exception
    {
        int N = 20;
        System.out.println(factorial(N));
    }
}
```

Output:

```
2432902008176640000
```

If we have to write the above program in C++, that would be too large and complex, we can look at Factorial of Large Number.
In this way, the BigInteger class is very handy to use because of its large method library and it is also used a lot in competitive programming.

Now below is given a list of simple statements in primitive arithmetic and its analogous statement in terms of BigInteger objects.

Declaration

```
int a, b;
BigInteger A, B;
```

Initialization:

```
a = 54;
b = 23;

A  = BigInteger.valueOf(54);

B  = BigInteger.valueOf(37);
```

And for Integers available as string you can initialize them as:

```
A  = new BigInteger("54");
B  = new BigInteger("123456789123456789");
```

Some constant are also defined in BigInteger class for ease of initialization :

```
A = BigInteger.ONE;
// Other than this, available constant are BigInteger.ZERO
// and BigInteger.TEN
```

Mathematical operations:

```
int c = a + b;
BigInteger C = A.add(B);
```

Other similar function are subtract() , multiply(), divide(), remainder()
But all these function take BigInteger as their argument so if we want these operation with integers or string convert them to BigInteger before passing them to functions as shown below:

```
String str = "123456789";
BigInteger C = A.add(new BigInteger(str));
int val  = 123456789;
BigInteger C = A.add(BigIntger.valueOf(val));
```

Extraction of value from BigInteger:

```
int x   =  A.intValue();   // value should be in limit of int x
long y   = A.longValue();  // value should be in limit of long y
String z = A.toString();
```

Comparison:

```
if (a < b) {}         // For primitive int
if (A.compareTo(B) < 0)  {} // For BigInteger
```

Actually compareTo returns -1(less than), 0(Equal), 1(greater than) according to values.

For equality we can also use:

```
if (A.equals(B)) {}  // A is equal to B
```

Methods of BigInteger Class:

1. **BigInteger abs():** This method returns a BigInteger whose value is the absolute value of this BigInteger.

2. **BigInteger add(BigInteger val):** This method returns a BigInteger whose value is (this + val).

3. **int bitCount():** This method returns the number of bits in the two's complement representation of this BigInteger that differ from its sign bit.

4. **int bitLength():** This method returns the number of bits in the minimal two's-complement representation of this BigInteger, excluding a sign bit.

5. **int compareTo(BigInteger val):** This method compares this BigInteger with the specified BigInteger.

6. **BigInteger divide(BigInteger val):** This method returns a BigInteger whose value is (this / val).

7. **double doubleValue():** This method converts this BigInteger to a double.

8. **float floatValue():** This method converts this BigInteger to a float.

9. **BigInteger gcd(BigInteger val):** This method returns a BigInteger whose value is the greatest common divisor of abs(this) and abs(val).

10. **int getLowestSetBit():** This method returns the index of the rightmost (lowest-order) one bit in this BigInteger (the number of zero bits to the right of the rightmost one bit).

11. **int intValue():** This method converts this BigInteger to an int.

12. **boolean isProbablePrime(int certainty):** This method returns true if this BigInteger is probably prime, false if it's definitely composite.

13. long longValue(): This method converts this BigInteger to a long.

14. BigInteger max(BigInteger val): This method returns the maximum of this BigInteger and val.

15. BigInteger min(BigInteger val): This method returns the minimum of this BigInteger and val.

16. BigInteger mod(BigInteger m): This method returns a BigInteger whose value is (this mod m).

17. BigInteger modInverse(BigInteger m): This method returns a BigInteger whose value is (this-1 mod m).

18. **BigInteger modPow(BigInteger exponent, BigInteger m):** This method returns a BigInteger whose value is (thisexponent mod m).

19. **BigInteger multiply(BigInteger val):** This method returns a BigInteger whose value is (this * val).

20. **BigInteger negate():** This method returns a BigInteger whose value is (-this).

21. **BigInteger or(BigInteger val):** This method returns a BigInteger whose value is (this | val).

22. **BigInteger pow(int exponent):** This method returns a BigInteger whose value is (thisexponent).

23. **BigInteger remainder(BigInteger val):** This method returns a BigInteger whose value is (this % val).

24. **BigInteger setBit(int n):** This method returns a BigInteger whose value is equivalent to this BigInteger with the designated bit set.

25. **BigInteger shiftLeft(int n):** This method returns a BigInteger whose value is (this << n).

26. **BigInteger shiftRight(int n):** This method returns a BigInteger whose value is (this >> n).

27. **BigInteger sqrt():** This method returns the integer square root of this BigInteger.

28. **BigInteger subtract(BigInteger val):** This method returns a BigInteger whose value is (this - val).

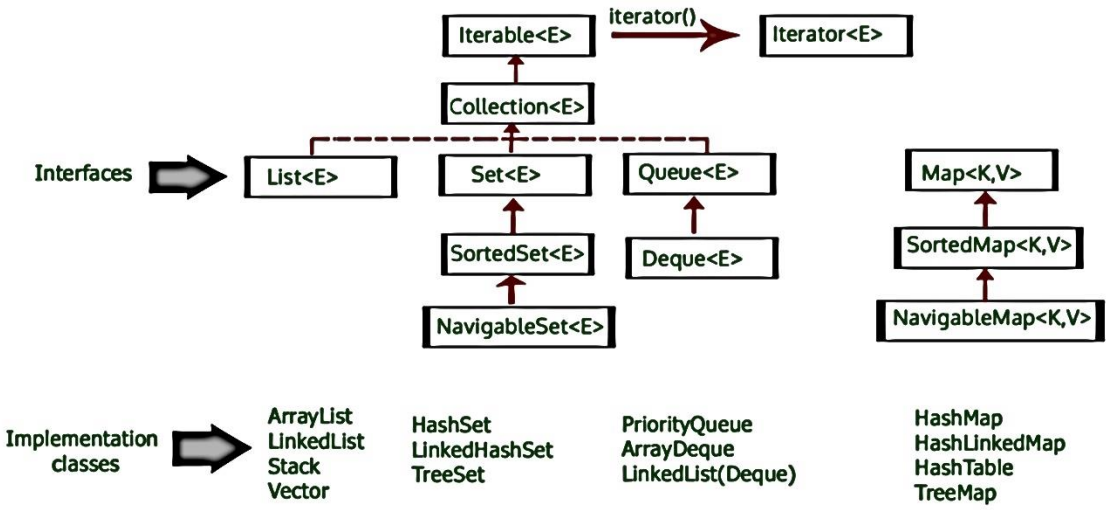
29. **boolean testBit(int n):** This method returns true if and only if the designated bit is set.

30**. String toString():** This method returns the decimal String representation of this BigInteger.

16 ArrayList in Java-

ArrayList is a part of collection framework and is present in java.util package. It provides us dynamic arrays in Java. Though it may be slower than standard arrays, it can be helpful in programs where lots of manipulation in the array is needed.

- ArrayList inherits AbstractList class and implements List interface.
- ArrayList is initialized by a size; however, the size can increase if collection grows or shrink if objects are removed from the collection.
- Java ArrayList allows us to randomly access the list.
- ArrayList cannot be used for primitive types such as int, char, etc. We need a wrapper class for such cases.
- ArrayList in Java can be seen as similar to vector in C++.



Now, primarily the Java ArrayList can constitute of both constructors and methods. Below mentioned is a list of few constructors and methods along with their use and functions.
Constructors in Java ArrayList:

1. ArrayList(): This constructor is used to build an empty array list.
2. ArrayList(Collection c): This constructor is used to build an array list initialized with the elements from collection c.
3. ArrayList(int capacity): This constructor is used to build an array list with initial capacity being specified.

Let us look at the code to create generic ArrayList-

```
// Creating generic integer ArrayList
ArrayList<Integer> arrli = new ArrayList<Integer>();
```

// Java program to demonstrate working of ArrayList in Java

```
import java.io.*;
import java.util.*;
class arrayli
{
   public static void main(String[] args)
                    throws IOException
   {
      // size of ArrayList
      int n = 5;
      //declaring ArrayList with initial size n
      ArrayList<Integer> arrli = new ArrayList<Integer>(n);
      // Appending the new element at the end of the list
      for (int i=1; i<=n; i++)
         arrli.add(i);
      // Printing elements
      System.out.println(arrli);
      // Remove element at index 3
      arrli.remove(3);
      // Displaying ArrayList after deletion
      System.out.println(arrli);
      // Printing elements one by one
      for (int i=0; i<arrli.size(); i++)
         System.out.print(arrli.get(i)+" ");
   }
}
```

Output:

```
[1, 2, 3, 4, 5]
[1, 2, 3, 5]
1 2 3 5
```

Methods in Java ArrayList:

1. **forEach():** Performs the given action for each element of the Iterable until all elements have been processed, or the action throws an exception.

2. **contains(Object o):** Returns true if this list contains the specified element.

3. **remove(int index):** Removes the element at the specified position in this list.

4. remove(Object o): Removes the first occurrence of the specified element from this list, if it is present.

5. **get(int index):** Returns the element at the specified position in this list.

6. **subList(int fromIndex, int toIndex):** Returns a view of the portion of this list between the specified fromIndex, inclusive, and toIndex, exclusive.

7. **set(int index, E element):** Replaces the element at the specified position in this list with the specified element.

8. **size():** Returns the number of elements in this list.

9. **removeAll():** Removes all of the elements that are contained in the specified collection from the list.

10. **isEmpty():** Returns true if this list contains no elements.

11. **void clear():** This method is used to remove all the elements from any list.

12. **void add(int index, Object element):** This method is used to insert a specific element at a specific position index in a list.

13. **int indexOf(Object O):** The index of the first occurrence of a specific element is either returned, or -1 in case the element is not in the list.

14. **int lastIndexOf(Object O):** The index of the last occurrence of a specific element is either returned, or -1 in case the element is not in the list.

HIMANSHU KUMAR(LINKEDIN)

https://www.linkedin.com/in/himanshukumarmahuri

CREDITS- INTERNET

DISCLOSURE- ALL THE DATA AND IMAGES ARE TAKEN FROM GOOGLE AND INTERNET.

CHECKOUT AND DOWNLOAD MY ALL NOTES

LINK- **https://linktr.ee/exclusive_notes**