Java Arrays

" An array is a finite and ordered collection of homogeneous data elements. It is finite because it contains a limited number of elements. It is ordered because all the elements are stored one by one in a contiguous location of computer memory (heap) in a linear fashion. It is homogeneous because all elements of an array are of the same data type only. We can store either primitive types or object references into it. "

- It is a data structure where we store similar elements.
- We can store only a fixed set of elements in a Java array.
- Array in Java is **index-based,** the first element of the array is stored at the 0th index, 2nd element is stored on 1st index and so on.
- In Java, array is an object of a **dynamically generated class**. Java array inherits the **Object class,** and **implements the Serializable** as well as **Cloneable interfaces**.
- The size of an array must be specified by **int** or **short value** and not long.

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**.

Instantiating an Array in Java

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

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

or

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

Note :

1. 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).
2. 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 to hold the array, using new, and assign it to the array variable.
   Thus, **in Java, all arrays are dynamically allocated.**

Array Literal

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

Arrays of Objects

```
Student[ ] arr = new Student[5];                    // Student is a user-defined class
```

" 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;
    }
}
public class Main
{
    public static void main (String[] args)
    {
        Student[] arr;
        arr = new Student[5];


        arr[0] = new Student(1,"aman");
        arr[1] = new Student(2,"vaibhav");
        arr[2] = new Student(3,"shikar");
        arr[3] = new Student(4,"dharmesh");
        arr[4] = new Student(5,"mohit");
        for (int i = 0; i < arr.length; i++)
            System.out.println("Element at "+i+": "+arr[i].roll_no +" "+arr[i].name);
    }
}
```

Is it possible to declare array size as negative?

" No, it is not possible to declare array size as negative. Still, if we declare the negative size, there will be no compile-time error. But we get the **NegativeArraySizeException** at run-time. "

How to copy an array in Java?

" We can create a copy of an array in two ways, first one is manually by iterating over the array and the second one is by using the **arrayCopy()** method. Using the arrayCopy() method of the System class is the **fastest** way to copy an array and also allows us to copy a part of the array. These two methods are the popular ways to copy an array.
The other two methods to copy an array is to use the **Arrays.copyOf()** method and using **clone()** method. "

What do you understand by the jagged array?

"A jagged array is a multidimensional array in which member arrays are of different sizes. For example, int array[][]=new int[3][]. The statement creates a two-dimensional jagged array. "

Which operations can be performed on an array?

" On an array, we can perform the searching, sorting, traversal, deletion, and insertion operation. "

How many ways to find the duplicate elements in an array?

There are the following five ways to find the duplicate array in Java.

- **Brute Force Method:** In this method, we compare each element of an array with the other elements. If any of the two elements are found equal, we consider them as duplicates. The method has time complexity **$O(n^2)$.**
- **Using HashSet:** We can also use the HashSet class to find the duplicate elements in an array. To find the duplicate elements, iterate over the array elements and insert them into HashSet by invoking add() method of the HashSet class. If the method returns false it means that the element is already present in the Set. It takes **O(n)** time to find the duplicate elements.
- **Using HashMap:** We know that HashMap uses key-value pair to store an element. When we use HashMap to find the duplicate array, we store the elements of the array as keys and the frequency of the elements as values. If the value of any key is greater than 1, the key is a duplicate element. Its time and space complexity is **O(n).** Using this method, we can also find the number of occurrences of duplicates.

What is the difference between Array and ArrayList?

Array: Array is static. It is of fixed size. Its size cannot be changed once it is declared. It contains both primitive data types and objects of a class. Array does not have generic features.
ArrayList: ArrayList is dynamic in size. Its size or capacity automatically grows when we add element into it. It contains only the object entries. It has a generic feature.

Methods in Java Array

```
Arrays.asList(arr);                          // returns a list of array's elements

Arrays.binarySearch(arr, fromIndex, toIndex, key, Comparator)

Arrays.compare(arr1, arr2);                  // compare two arrays, return "1" if true

Arrays.copyOf(arr, newLength);               // returns copy of array of new length

Arrays.copyOf(arr, fromIndex, toIndex);      // returns copy of array from given index range

Arrays.equals(arr1, arr2);                   // returns true if both arrays are equal

Arrays.fill(arr, value);                     // fills array element with given value

Arrays.mismatch(arr1, arr2);                 // returns index of mismatch element

Arrays.parallelSort(arr);                    // returns sorted array

Arrays.sort(arr, (a, b) -> a[0] - b[0]);     // returns column wise sorted 2d array
```

```
Arrays.sort(arr);                          // returns sorted array

Arrays.sort(arr, fromIndex, toIndex);      // returns sorted array in range

Arrays.toString(arr)                       // returns string of array element
```

Sorting a 2D Array according to values in any given column in Java

```
import java.util.*;
class sort2DMatrixbycolumn {
    public static void sortbyColumn(int arr[][], int col)
    {
        Arrays.sort(arr, new Comparator<int[]>() {

          @Override
          // Compare values according to columns
          public int compare(final int[] entry1,
                             final int[] entry2) {
            if (entry1[col] > entry2[col])
                return 1;
            else
                return -1;
          }
        });
    }
    public static void main(String args[])
    {
        int matrix[][] = { { 39, 27, 11, 42 },
                           { 10, 93, 91, 90 },
                           { 54, 78, 56, 89 },
                           { 24, 64, 20, 65 } };
        int col = 3;
        sortbyColumn(matrix, col - 1);
          for (int i = 0; i < matrix.length; i++) {
            for (int j = 0; j < matrix[i].length; j++)
                System.out.print(matrix[i][j] + " ");
            System.out.println();
        }
    }
}
```

Java ArrayList

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

Since ArrayList is a dynamic array and we do not have to specify the size while creating it, the size of the array automatically increases when we dynamically add and remove items.
when the array becomes full and if we try to add an item: It

- Creates a bigger-sized memory on heap memory (for example memory of double size).
- Copies the current memory elements to the new memory.
- New item is added now as there is bigger memory available now.
- Delete the old memory.

Important Features:

- ArrayList inherits **AbstractList** class and implements the **List interface.**
- ArrayList is initialized by the size. However, the size is increased automatically if the collection grows or shrinks if the objects are removed from the collection.
- Java ArrayList allows us to randomly access the list.
- ArrayList cannot be used for primitive types, like int, char, etc. We need a **wrapper class** for such cases.
- ArrayList in Java can be seen as a **vector in C++**.
- ArrayList is not **Synchronized.** Its equivalent synchronized class in Java is **Vector.**

Constructors in the ArrayList

ArrayList(): This constructor is used to build an empty array list.

ArrayList arr = new ArrayList();

ArrayList(Collection c): This constructor is used to build an array list initialized with the elements from the collection c.

ArrayList arr = new ArrayList(c);

ArrayList(int capacity): This constructor is used to build an array list with initial capacity being specified.

ArrayList arr = new ArrayList(N);

Methods in Java ArrayList

```
ArrayList<Integer> List = new ArrayList<Integer>();
List.add(int index, Object element);          // add items to list at specific index
List.add(Object O);                           // add items to end of the list
List.addAll(Collection C);                    // add all element from specific collection
List.addAll(int index, Collection C);         // add all element at specific position
List.clear();                                 // clear all the element from the list
List.clone();                                 // use to create a clone of the same list
List.contains(Object O);                      // return true if element found in the list
List.get(int index);                          // returns the element from specific position
List.indexOf(Object O);                       // returns the index of first occurrence of element
List.isEmpty();                               // returns true if list contains no element
List.remove(int index);                       // remove element form specific index
List.remove(Object O);                        // remove first occurrence of object from the list
List.removeAll(Collection c);                 // remove all element of collection from list
List.removeRange(int fromIndex, int toIndex); // remove elements from list in index range
List.set(int index, E element);               // replace the element from specific position
List.size();                                  // returns the list size
List.subList(int fromIndex, int toIndex);     // returns sub list from specific index range
List.toArray();                               // returns an array of list elements
List.trimToSize();                            // trim the capacity of list to current size
Collections.sort(List);                       // returns sorted ArrayList
Collections.sort(List, Collections.reverseOrder());
```

Java Strings

Strings in Java are Objects that are backed internally by a **char array**. Since arrays are **immutable**(cannot grow), Strings are **immutable** as well. Whenever a **change** to a String is made, an entirely **new String is created.**

Whenever a String Object is created as a literal, the object will be created in **String constant pool.** This allows JVM to optimize the initialization of String literal.

Creating a String

There are two ways to create String object:

1. By string literal
2. By new keyword

String Literal

```
String s1="Welcome";
String s2="Welcome";     //It doesn't create a new instance
```

Each time you create a string literal, the JVM checks the "string constant pool" first. If the string already exists in the pool, a reference to the pooled instance is returned. If the string doesn't exist in the pool, a new string instance is created and placed in the pool.

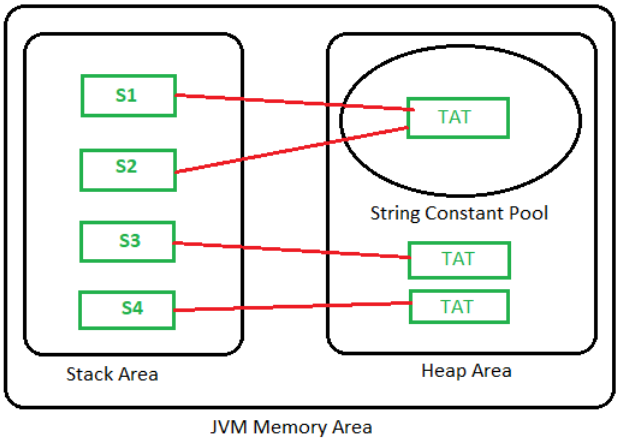
String Object by new keyword

```
String s = new String("Welcome");
```

In such case, JVM will create a new string object in normal (non-pool) heap memory, and the literal "Welcome" will be placed in the string constant pool. The variable s will refer to the object in a heap (non-pool).

Example,

```
String s1 = "TAT";
String s2 = "TAT";
String s3 = new String("TAT");
String s4 = new String("TAT");
```



Strings Immutability

Strings are stored in String Constant Pool, where many Strings instances are pointing the same reference, so if we try to modify a string, it'll reflect the same to all instances. To avoid this problem, strings are made as immutable, any changes in string will allocate new space in memory pointing the new string.

String Pool

String Pool in Java is a special storage space in Java **heap memory**. The other names of string pool are **String Constant Pool** or even **String Intern Pool**. Java strings are created and stored in this area.

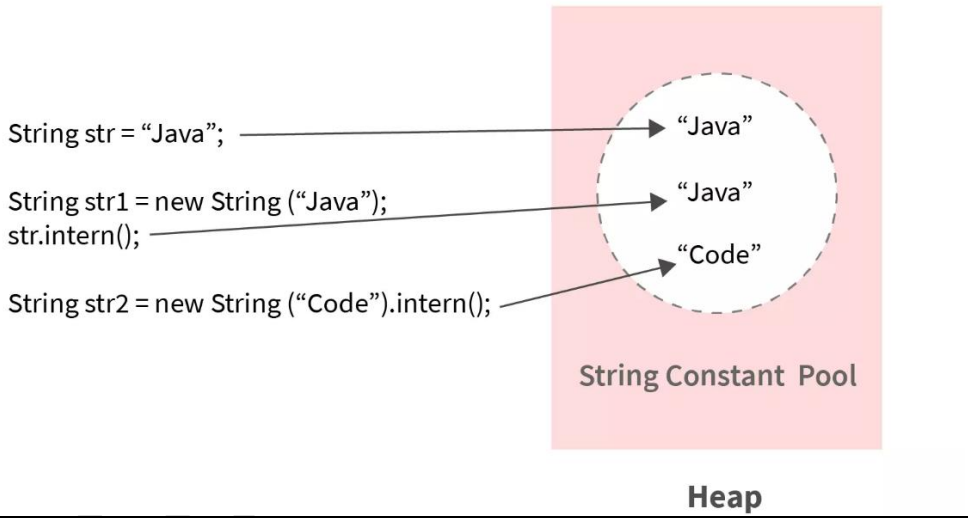
String Interning

String Interning in Java is a process where identical strings are searched in the string pool and if they are present, the same memory is shared with other strings having the same value.

" String Pool is a special storage space in the Heap memory. "

Using String.intern() method

It places a string irrespective of its value(whether the same string is already present) **inside the string constant pool.**




```
String S1 = "Hello world";
String S2 = "Hello world";
String S3 = new String("Hello world");
S1 == S2          // True
S1 == S3          // False
S1.equals(S3);    // True
```

Here, == operator compare the address of the object,
So, to compare the string by character wise, we should use equals method.

Advantages of String Pool in Java

- ✓ Java String Pool allows caching of string. Caching here is the process of storing data in a cache. Cache improves performance and reduces memory usage.

- ✓ Provides reusability: It saves time to create a new string if there is already a string with the same value present in the pool with the same value. The old string is reused and its reference is returned.

Disadvantages of Using String Objects

- ✓ Strings have a constant value and even if they are altered, instead of reflecting the changes in the original string, a new object is created.

- ✓ This causes a lot of objects to be created in the heap and wastes a lot of memory if the user keeps on updating the value of the string.

- ✓ In order to overcome the drawbacks of the String class, Java provides **StringBuffer** and **StringBuilder** classes. They are used to create **mutable** string objects.

Key Points

- ✓ A string is a **set of characters** that are always enclosed in double-quotes.
- ✓ Strings are **immutable** in nature.
- ✓ This immutability is achieved through **String Pool**.
- ✓ **String Pool** in Java is a **special storage** space in Java **heap memory**. It is also known as **String Constant Pool** or **String Intern Pool.**
- ✓ Whenever a new string is created, JVM first checks the string pool. If it encounters the same string, then instead of creating a new string, it returns the **same instance** of the **found string** to the variable.
- ✓ Java String Pool allows **caching** of string and **reusability.**

Methods in Java Strings

str.length();	// returns the length of string
str.charAt(index);	// returns char at that index
str.toCharArray( );	// returns character array
String(arr, from index, to index);	// return string from given array index range
String.copyValueOf(arr, idx1, idx2);	// return string from array
str.isEmpty( );	// returns true if empty
str.split(" ");	// split the string with given character
str.contains("Hello");	// check whether a string contains substring
str.substring(fromIndex, toIndex);	// returns a substring in given index range
str.substring(index);	// returns a substring from to end index
str.toString( );	// returns the string
str.subSequence(startIndex, endIndex);	// returns the subsequence in given range
str.trim( );	// returns trimmed whitespace from string
String.join(str1, str2, str3);	// joins multiple strings
str.replace('H', 'C');	// replace specified character
str.replaceAll(regex, " ");	// replace all substring matching the regrex
str.replaceFirst('a', 'b');	// replace only first occurrence of character
str.indexOf(string);	// returns the first matching of the string/char
str.lastIndexOf(string);	// returns the last matching index
str1.compareTo(str2);	// compare in dictionary order
str1.equals(str2);	// compare two string
str1.equalsIgnoreCase(str2);	// compare by ignoring case of string
String.format("Language: %s", str);	// returns the formatted string
str.valueOf(argument);	// returns string with passed argument
str.startWith("Hello");	// check whether it starts with given string/char
str.endsWith("World");	// check whether it ends with given string/char
str.intern( );	// allocate the string space in string pool
str.contentEquals(StringBuffer sb);	// compares the contents
str.hashCode( );	// returns the hashcode
str1.concat(str2);	// merge two string

Java StringBuffer & StringBuilder

StringBuffer is a peer class of String that provides much of the functionality of strings. The string represents **fixed-length, immutable** character sequences while StringBuffer represents **growable** and **writable** character sequences. StringBuffer may have characters and substrings inserted in the middle or appended to the end. It will **automatically grow** to make room for such additions and often has more characters preallocated than are actually needed, to allow room for growth.

KeyPoints

- java.lang.StringBuffer extends (or inherits from) Object class.
- All Implemented Interfaces of StringBuffer class: **Serializable, Appendable, CharSequence.**
- public final class StringBuffer extends Object implements Serializable, CharSequence, Appendable.
- String buffers are safe for use by **multiple threads**. The methods can be **synchronized** wherever necessary so that all the operations on any particular instance behave as if they occur in some serial order.
- Whenever an operation occurs involving a source sequence (such as appending or inserting from a source sequence) this class synchronizes only on the string buffer performing the operation, not on the source.
- It inherits some of the methods from the Object class which such as **clone(), equals(), finalize(), getClass(), hashCode(), notifies(), notifyAll().**

Constructors of StringBuffer class

StringBuffer(): It reserves room for 16 characters without reallocation

```
StringBuffer Sb = new StringBuffer();
```

StringBuffer(int size): It accepts an integer argument that explicitly sets the size of the buffer.

```
StringBuffer Sb = new StringBuffer(20);
```

StringBuffer(String str): It accepts a string argument that sets the initial contents of the StringBuffer object and reserves room for 16 more characters without reallocation.

```
StringBuffer Sb = new StringBuffer("Hello World");
```

StringBuilder is very much similar to the StringBuffer class. However, the StringBuilder class **differs** from the **StringBuffer** class on the basis of **synchronization.** The **StringBuilder** class provides **no guarantee of synchronization** whereas the **StringBuffer class does**. Therefore, this class is designed for use as a drop-in replacement for StringBuffer in places where the StringBuffer was being used by a **single thread** (as is generally the case). Where possible, it is recommended that this class be used in preference to StringBuffer as it will be **faster** under most implementations. **Instances of StringBuilder are not safe** for use by multiple threads. If

such **synchronization is required** then it is recommended that **StringBuffer be used.** String Builder is **not thread-safe** and high in performance compared to String buffer.

Methods in Java StringBuffer/StringBuilder

StringBuffer Sb = new StringBuffer( );

```
Sb.append(string);                          // append the string/char at end
Sb.length( );                               // returns StringBuffer length
Sb.capacity( );                             // returns the total allocated memory
Sb.charAt(index);                           // returns the char at given index
Sb.delete(stirng/char);                     // delete the char/string from StringBuffer
Sb.deleteCharAt(index);                     // delete the string/char at given index
Sb.ensureCapacity( );                       // increase the capacity of StringBuffer object
Sb.insert(index);                           // insert string/char at given index
Sb.reverse( );                              // returns the reversed string
Sb.replace('a', 'b');                       // replace one set of char with another set
Sb.getChars(from, to idx, arr, arrIdx);     // copy chars from StringBuilder to char[ ] arr
Sb.indexOf(string);                         // returns the index of matching string
Sb.lastIndexOf(string);                     // returns the index of last matching string
Sb.setCharAt(index, char);                  // set char at given index
Sb.setLength(int newLength);                // set new length of StringBuffer
Sb.subSequence(start idx, end idx);         // returns sub sequence from given range
Sb.subString(index, index);                 // returns sub string
Sb.toString( );                             // returns string from String Buffer
Sb.trimToSize( );                           // reduce the storage space of String Buffer
```

StringBuffer & StringBuilder have the same methods