Array

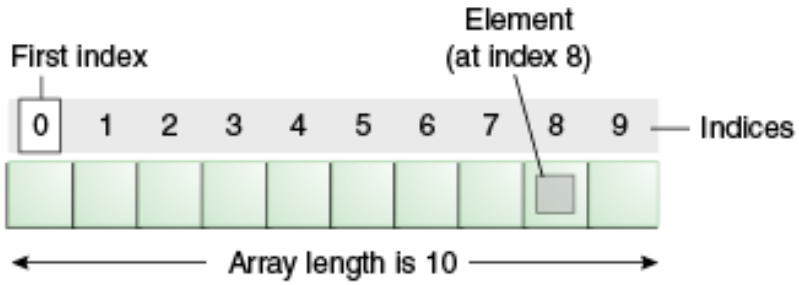
Learning Outcomes:

After successful completion of this lesson, you should be able to:

- Use arrays to implement sort and search algorithms.

Java array is an object which contains elements of a similar data type. Additionally, The elements of an array are stored in a contiguous memory location. It is a data structure where we store similar elements. We can store only a fixed set of elements in a Java array.

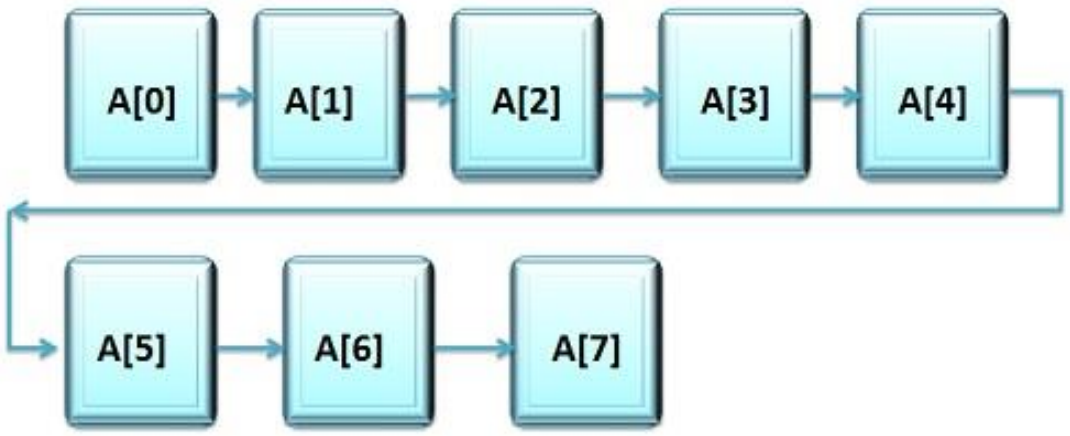
Array in Java is index-based, the first element of the array is stored at the 0th index, 2nd element is stored on 1st index and so on.



Types of Array in Java

- **Single Dimensional Array / One dimensional Array**

Considered as the “list of variables of similar data types”, and each variable can be distinctly accessed by specifying its index in square brackets preceded by the name of that array.



One-Dimensional Array A[8]

Examples:

One dimensional array declaration of variable:

```
import java.io.*;

class GFG {
        public static void main(String[] args)
        {
                int[] a; // valid declaration
                int b[]; // valid declaration
                int[] c; // valid declaration
        }   }
```

We can write it down in any way.

Now if you want to declare your array as below:

```
import java.io.*;

class GFG {
        public static void main(String[] args)
        {
                // invalid declaration -- If we want to assign
                // size of array at the declaration time, it
                // gives compile time error.
                int a[5];

                // valid declaration
                int b[];
        }   }
```

Now, suppose we want to write multiple variable array declaration, then we can use it like this.

```
import java.io.*;

class GFG {
        public static void main(String[] args)
        {
                // valid declaration, both arrays are
                // one dimensional array.
                int a[], b[];

                // invalid declaration
                int c[], [] d;

                // invalid declaration
                int[] e, [] f;
        } }
```

When we declare multiple variables at the same time, we must first write variables and then declare the variable except the first variable declaration. There is no restriction on the first variable.

Now, when we're creating an array, it's mandatory to pass the size of the array; otherwise, we'll get a compile time error.

You can use the new operator to create an array.

```
import java.io.*;

class GFG {
        public static void main(String[] args)
        {
                // invalid, here size of array is not given
                int[] a = new int[];

                // valid, here creating 'b' array of size 5
                int[] b = new int[5];

                // valid
                int[] c = new int[0];

                // gives runtime error
                int[] d = new int[-1];
        }  }
```

Printing Array:

```
/* A complete Java program to demonstrate working
of one dimensional arrays */
class oneDimensionalArray {

        public static void main(String args[])
        {
                int[] a; // one dimensional array declaration
                a = new int[3]; // creating array of size 3
                for (int i = 0; i < 3; i++) {
                        a[i] = 100;
                        System.out.println(a[i]);
                }
        }
}
```

Output:

100
100
100

- **Two Dimensional Array**

The two-dimensional array in the Java programming language is nothing but an array of arrays. In the Java Two Dimensional Array, data stored in rows and columns can be accessed using both the row index and column index (like an Excel file). If the data is linear, the One Dimensional Array can be used. However, in order to work with multi-level data, we need to use the Multi-Dimensional Array.

```
// Java program to demonstrate different ways
// to create two dimensional array.
import java.io.*;

class GFG {
        public static void main(String[] args)
        {
                int a[][]; // valid
                int[][] b; // valid
                int[][] c; // valid
                int[] d[]; // valid
                int[][] e; // valid
                int[] f[]; // valid
                [][] int g; // invalid
                [] int[] h; // invalid
        }  }
```

Now, suppose we want to write multiple declarations of array variable then you can use it like this:

```
// Java program to demonstrate multiple declarations
// of array variable
import java.io.*;

class GFG {
public static void main(String[] args)
        {
                // Here, 'a' is two dimensional array, 'b'
                // is two dimensional array
                int[] a[], b[];
                // Here, 'c' is two dimensional array, 'd'
                // is two dimensional array
                int[] c[], d[];

                // Here, 'e' is two dimensional array, 'f'
                // is three dimensional array
                int[][] e, f[];

                // Here, 'g' is two dimensional array,
                // 'h' is one dimensional array
                int[] g[], h;
        } }
```

Creating a single dimensional array and two dimensional array without a new operator:

```
/* Java program for one and two dimensional arrays.
without new operator*/
class oneTwoDimensionalArray {

        public static void main(String args[])
        {
                int[] a[] = { { 1, 1, 1 }, { 2, 2, 2 },
                                        { 3, 3, 3 } }, b = { 20 };
                // print 1D array
                System.out.println(b[0]);

                // print 2D array
                for (int i = 0; i < 3; i++) {
                        for (int j = 0; j < 3; j++) {
                                a[i][j] = 100;
                                System.out.println(a[i][j]);
                        }
                }
        }
}
```

Output:

```
20
100
100
100
100
100
100
100
100
100
```

Creating one dimensional array and two dimensional array using new operator:

```
/* Java program for one and two dimensional arrays.
using new operator*/
class oneTwoDimensionalArray {

        public static void main(String args[])
        {
           int[] a[], b = { 20 };
           a = new int[3][3];
           b = new int[3];

           // print 1D array
           for (int i = 0; i < 3; i++)
```

```
                System.out.println(b[i]);

            // print 2D array
            for (int i = 0; i < 3; i++) {
                for (int j = 0; j < 3; j++) {
                    a[i][j] = 100;
                    System.out.println(a[i][j]);
                }
            }
        }
}
```

Output:

```
0
0
0
100
100
100
100
100
100
100
100
100
```

How to print array in Java

The Java array is a data structure where the elements of the same data type can be stored. The array elements are stored in a contiguous memory location. So, in an array, we can store a fixed set of elements.

There are following ways to print an array in Java:

- Java for loop
- Java for-each loop
- Java Arrays.toString() method
- Java Arrays.deepToString() method
- Java Arrays.asList() method
- Java Iterator Interface
- Java Stream API

Java for loop is used to execute a set of statements repeatedly until a particular condition is satisfied.

Syntax:

```
for(initialization; condition; increment/ decrement)
{
//statements
}
```

Example for loop:

In the following example, we have created an array of lengths of four and initialized elements. We used the loop to get the values from the array. It's the most popular way to print arrays in Java.

```
public class PrintArrayExample1
{
public static void main(String args[])
{
//declaration and instantiation of an array
int arr[]=new int[4];
//initializing elements
arr[0]=10;
arr[1]=20;
arr[2]=70;
arr[3]=40;
//traversing over array using for loop
for(int i=0;i<arr.length;i++)    //length is the property of the array
System.out.println(arr[i]);
}
}
```

Output:

```
10
20
70
40
```

Java for-each loop is also used to pass through an array or collection. It works on the basis of its elements. Returns the elements one by one in a defined variable.

Syntax:

```
for(Type var:array)
```

In the following example, we have generated a string type array with a length of four and initialized elements. We used-each loop to run through the array.

```
public class PrintArrayExample2
{
public static void main(String args[])
{
// declaration and instantiation of an array
String[] city = new String[4];
//initializing elements
city[0] = "Delhi";
city[1] = "Jaipur";
city[2] = "Gujarat";
```

```
city[3] = "Mumbai";
//traversing over array using for-each loop
for (String str : city)
{
System.out.println(str);
}
}
}
```

Output:

Delhi
Jaipur
Gujarat
Mumbai

Java Searching

The simplest type of search is the sequential search. In the sequential search, each element of the array is compared to the key, in the order it appears in the array, until the desired element is found. If you are looking for an element that is near the front of the array, the sequential search will find it quickly. The more data that must be searched, the longer it will take to find the data that matches the key.

```
public static int search(int [ ] numbers, int key)
{
      for (int index = 0; index < numbers.length;
index++)
     {
          if ( numbers[index] = = key )
               return index;  //We found it!!!
     }
    // If we get to the end of the loop, a value has not
yet
    // been returned.  We did not find the key in this
array.
    return -1;
}
```

If the **key** value is found, the index (subscript) of the location is returned. This tells us that the return value x, will be the first integer found such that
numbers [x] = key.

There may be additional **key** locations in this array beyond this location.

Example Program:

```
// Java code to demonstrate search() method
import java.util.*;

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

       // Creating an empty Stack
       Stack<String> STACK = new Stack<String>();
```

```
        // Stacking strings
        STACK.push("Geeks");
        STACK.push("4");
        STACK.push("Geeks");
        STACK.push("Welcomes");
        STACK.push("You");

        // Displaying the Stack
        System.out.println("The stack is: " + STACK);

        // Checking for the element "4"
        System.out.println("Does the stack contains '4'? "
                           + STACK.search("4"));
        // Checking for the element "Hello"
        System.out.println("Does the stack contains 'Hello'? "
                           + STACK.search("Hello"));

        // Checking for the element "Geeks"
        System.out.println("Does the stack contains 'Geeks'? "
                           + STACK.search("Geeks"));
    }
}
```

Output:

```
The stack is: [8, 5, 9, 2, 4]
Does the stack contains '9'? 3
Does the stack contains '10'? -1
Does the stack contains '11'? -1
```

Java Sorting

Sorting is ordering a list of objects. We can distinguish two types of sorting. If the number of objects is small enough to fits into the main memory, sorting is called internal sorting. If the number of objects is so large that some of them reside on external storage during the sort, it is called external sorting.

Bucket sort

Suppose we need to sort an array of positive integers {3,11,2,9,1,5}. A bucket sort works as follows: create an array of size 11. Then, go through the input array and place integer 3 into a second array at index 3, integer 11 at index 11 and so on. We will end up with a sorted list in the second array.

Suppose we are sorting a large number of local phone numbers, for example, all residential phone numbers in the 412 area code region (about 1 million) We sort the numbers without use of comparisons in the following way. Create an a bit array of size 107. It takes about 1Mb. Set all bits to 0. For each phone number turn-on the bit indexed by that phone number. Finally, walk through the array and for each bit 1 record its index, which is a phone number.

We immediately see two drawbacks to this sorting algorithm. Firstly, we must know how to handle duplicates. Secondly, we must know the maximum value in the unsorted array..

Thirdly, we must have enough memory - it may be impossible to declare an array large enough on some systems.

The first problem is solved by using linked lists, attached to each array index. All duplicates for that bucket will be stored in the list. Another possible solution is to have a counter. As an example let us sort 3, 2, 4, 2, 3, 5. We start with an array of 5 counters set to zero.

0	1	2	3	4	5
0	0	0	0	0	0

Moving through the array we increment counters:

0	1	2	3	4	5
0	0	2	2	1	1

Next, we simply read off the number of each occurrence: 2 2 3 3 4 5.

Bubble sort

The algorithm works by comparing each item in the list with the item next to it, and swapping them if required. In other words, the largest element has bubbled to the top of the array. The algorithm repeats this process until it makes a pass all the way through the list without swapping any items.

```
void bubbleSort(int ar[])
{
   for (int i = (ar.length - 1); i >= 0; i--)
   {
      for (int j = 1; j ≤ i; j++)
      {
        if (ar[j-1] > ar[j])
         {
              int temp = ar[j-1];
              ar[j-1] = ar[j];
              ar[j] = temp;
   }   }
 }  }
```

Example.

Here is one step of the algorithm. The largest element - 7 - is bubbled to the top:

7,	**5**,	2,	4,	3,	9
5, **7,**	**2**,	4,	3,		9
5,	2, **7,**	**4**,	3,		9
5,	2,	4, **7,**	**3**,		9
5,	2,	4,	3, **7,**		**9**
5, 2, 4, 3, 7, 9					

The worst-case runtime complexity is O(n2). See explanation below:

```
void bubbleSort(int ar[])
{
   for (int i = (ar.length - 1); i >= 0; i--)
   {
      for (int j = 1; j <= i; j++)
      {
         if (ar[j-1] > ar[j])
         {
               int temp = ar[j-1];
               ar[j-1] = ar[j];
               ar[j] = temp;
   } } } }
```

O(1) O(i) $\sum_{i=0}^{i=n} O(i)$

$$\sum_{i=0}^{i=n} O(i) = 1 + 2 + 3 + \ldots + (n-1) = O(n^2)$$

Selection Sort

The algorithm works by selecting the smallest unsorted item and then swapping it with the item in the next position to be filled.

The selection sort works as follows: you look through the entire array for the smallest element, once you find it you swap it (the smallest element) with the first element of the array. Then you look for the smallest element in the remaining array (an array without the first element) and swap it with the second element. Then you look for the smallest element in the remaining array (an array without first and second elements) and swap it with the third element, and so on.

Here is an example,

```
void selectionSort(int[] ar){
  for (int i = 0; i ‹ ar.length-1; i++)
  {
    int min = i;
    for (int j = i+1; j ‹ ar.length; j++)
        if (ar[j] ‹ ar[min]) min = j;
    int temp = ar[i];
```

```
      ar[i] = ar[min];
      ar[min] = temp;
} }
```

Example.

29,	64,	73,	34, **20**,
20, **64**,	73,		34, **29**,
20,	29, **73**, **34**,		64
20,	29,		34, **73**, **64**
20, 29, 34, 64, 73			

Insertion sort

To sort unordered list of elements, we remove its entries one at a time and then insert each of them into a sorted part (initially empty):

```
void insertionSort(int[] ar)
{
   for (int i=1; i ‹ ar.length; i++)
   {
      int index = ar[i]; int j = i;
      while (j > 0 && ar[j-1] > index)
      {
          ar[j] = ar[j-1];
          j--;
      }
      ar[j] = index;
} }
```

Example. We color a sorted part in green, and an unsorted part in black. Here is an insertion sort step by step. We take an element from unsorted part and compare it with elements in sorted part, moving from right to left.

29,	20,	73,	34,	64
29,	20,	73,	34,	64
20,	**29**,	73,	34,	64
20,	**29,**	**73**,	34,	64
20,	**29,**	**34,**	**73**,	64
20, 29, 34, 64, 73				

Mergesort

Merge-sort is based on the divide-and-conquer paradigm. It involves the following three steps:

1. Divide the array into two (or more) subarrays
2. Sort each subarray (Conquer)
3. Merge them into one (in a smart way!)

Example. Consider the following array of numbers

27 10 12 25 34 16 15 31
divide it into two parts
27 10 12 25 34 16 15 31
divide each part into two parts
27 10 12 25 34 16 15 31
divide each part into two parts
27 10 12 25 34 16 15 31
merge (cleverly-!) parts
10 27 12 25 16 34 15 31
merge parts
10 12 25 27 15 16 31 34
merge parts into one
10 12 15 16 25 27 31 34

How do we merge two sorted subarrays? We define three references at the front of each array