Introduction, Declaration & Initialization of Arrays

In Java, an array is a data structure used to store a collection of elements of the same data type.

Introduction:

- Arrays are objects in Java, meaning they are dynamically created and stored in the heap memory.
- They provide a convenient way to store and access a group of related values.
- Arrays are fixed in size, meaning once created, their size cannot be changed.

Declaration:

- To declare an array in Java, you specify the data type, followed by square brackets "[]", and then the array name.
- Syntax: datatype[] arrayName;
- Ex: int[] numbers;

Initialization:

- An array can be initialized in two ways:

1. Using the "new" keyword:

   Syntax: datatype[] arrayName = new datatype[size];

   Ex: int[] numbers = new int[5];

   This creates an array named "numbers" that can hold 5 integer values. The elements are initialized to their values, which is 0 for integers.

2. Using an array literal:

   Syntax: datatype[] arrayName = {arrayElement_1, arrayElement_2…., arrayElement_n};

   Ex: int[] numbers = {1, 2, 3, 4, 5};

   This creates an array name "numbers" and initializes it with the specified values. The size of the array is automatically determined based on the number of elements provided.

Program:

```
import java.lang.*;
import java.util.*;
class Arrays{
    public static void main(String args[]){
       //Array is declared & initialized
       int arr[] = {10, 20, 30, 40, 50};
       //Displaying array elements
       System.out.println(arr[0]);
       System.out.println(arr[1]);
       System.out.println(arr[2]);
       System.out.println(arr[3]);
       System.out.println(arr[4]);
    }
}
```

Storage of Array in Computer Memory

In Java, arrays are objects, and like all objects, they are stored in the heap memory. Here's a breakdown of how it works:

Memory Allocation:

- When you create an array using the new keyword, Java allocates a contiguous block of memory in the heap to store the array elements.
- The size of this block is determined by the number of elements and the data type of the array.
- For example, an array of 10 integers will require 40 bytes of memory (4 bytes per integer).

Structure:

- The array object itself contains a header that stores metadata, including the array's length and the data type of its elements.

- Following the header, the actual array elements are stored in contiguous memory locations.

Accessing Elements:

- To access an element in the array, you use its index.
- Java calculates the memory address of the desired element based on the starting address of the array, the size of each element, and the index.

Example:

```
int[] numbers = new int[3];

// Create an array of 3 integers

numbers[0] = 10;

numbers[1] = 20;

numbers[2] = 30;
```

Important Points:

Contiguous memory:

The elements of an array are stored in contiguous memory locations, which allows for fast access using index-based operations.

Heap Memory:

Arrays are stored on the heap, which is managed by the garbage collector. This means that you don't need to manually deallocate memory for arrays.

Fixed Size:

Once an array is created, its size cannot be changed. If you need to add or remove elements, you need to create a new array and copy the elements over.

Accessing Java Array Elements

(or)

Review the same topic from - Operations on Array Elements

- Each variable in a Java array is also called an "element". Thus, the example shown earlier created an array with space for 10 elements, and each element is a variable of type int.

- Each element in the array has an index (a number). You can access each element in the array via its index.

Example:

```
intArray[0] = 0;

int firstInt = intArray[0];
```

This example first sets the value of the element (int) with index 0, and second it reads the value of the element with index 0 into an int variable.

You can use the elements in a Java array just like if they were ordinary variables.

You can read their value, assign values to them, use the elements in calculations and pass specific elements as parameters to method calls.

Operations on Array Elements

Arrays are fundamental structures in Java that allow us to store multiple values of the same type in a single variable. They are useful for managing collections of data efficiently. Arrays in Java work differently than they do in C/C++.

Syntax of Array Declaration:

```
data_type  array_name[size1][size2]...[sizeN];
```

Operations on Array:

- Sorting of array elements
- Searching of array elements
- Access/Display of array elements

1. Sorting of array elements:

Insertion sort: Insertion sort is a simple sorting algorithm that works by iteratively inserting each element of an unsorted list into its correct position in a sorted portion of the list.

Program:

```
//Java program for implementation of Insertion Sort

class InsertionSort{

/* Function to sort array using insertion sort */

void sort(int arr[]){
```

```
          int n = arr.length;
          for (int i = 1; i < n; ++i) {
                    int key = arr[i];
                    int j = i - 1;
                    /* Move elements of arr[0..i-1], that are greater than key, to one
                    position ahead of their current position */
                    while (j >= 0 && arr[j] > key){
                    arr[j + 1] = arr[j];
                    j = j - 1;
            }
            arr[j + 1] = key;
        }
    }
    /* A utility function to print array of size n */
static void printArray(int arr[])
    {
        int n = arr.length;
          for (int i = 0; i < n; ++i)
             System.out.print(arr[i] + " ");
        System.out.println();
    }
public static void main(String args[])
    {
        int arr[] = { 12, 11, 13, 5, 6 };
        InsertionSort ob = new InsertionSort();
        ob.sort(arr);
        printArray(arr);
    }
```

}

Selection sort: Selection Sort is a comparison-based sorting algorithm. It sorts an array by repeatedly selecting the smallest (or largest) element from the unsorted portion and swapping it with the first unsorted element. This process continues until the entire array is sorted.

Program:

```
//Java program for implementation of Selection Sort
import java.util.Arrays;
class Selectionsort{
   static void selectionSort(int[] arr){
        int n = arr.length;
        for (int i = 0; i < n - 1; i++) {
         // Assume the current position hold the minimum element
         int min_idx = i;
         // Iterate through the unsorted portion to find the actual minimum
         for (int j = i + 1; j < n; j++){
                if (arr[j] < arr[min_idx]){
                // Update min_idx if a smaller element is found
                min_idx = j;
             }
         }
         // Move minimum element to its correct position
        int temp = arr[i];
        arr[i] = arr[min_idx];
        arr[min_idx] = temp;
      }
}
   static void printArray(int[] arr){
        for (int val : arr){
         System.out.print(val + " ");
      }
      System.out.println();
   }
   public static void main(String[] args){
         int[] arr = { 64, 25, 12, 22, 11 };
         System.out.print("Original array: ");
        printArray(arr);
        selectionSort(arr);
        System.out.print("Sorted array: ");
        printArray(arr);
   }
}
```

<u>Merge sort</u>: Merge sort is a sorting algorithm that follows the divide-and-conquer approach. It works by recursively dividing the input array into smaller subarrays and sorting those subarrays then merging them back together to obtain the sorted array**.**

<u>Program</u>:

```
public class MergeSort{
    // Merge two subarrays L and M into array
    void merge(int[] array, int left, int mid, int right){
    // Sizes of two subarrays to be merged
    int n1 = mid - left + 1;
    int n2 = right - mid;
    // Temporary arrays
    int[] L = new int[n1];
    int[] M = new int[n2];
    i++;
    } else {
      // Copy data to temporary arrays

      for (int i = 0; i < n1; i++)

          L[i] = array[left + i];

      for (int j = 0; j < n2; j++)
          M[j] = array[mid + 1 + j];
    // Initial indexes of first and second subarrays
     int i = 0, j = 0;
     // Initial index of merged subarray array
     int k = left;
     while (i < n1 && j < n2) {
         if (L[i] <= M[j]) {
             array[k] = L[i];
            array[k] = M[j];
             j++;
         }
      k++;
    }
    // Copy remaining elements of L[] if any
     while (i < n1) {
         array[k] = L[i];
         i++;
         k++;
    }
     // Copy remaining elements of M[] if any
while (j < n2) {
          array[k] = M[j];
          j++;
```

```
            k++;
        }
    }
    // Divide the array into two subarrays, sort them and merge them
    void mergeSort(int[] array, int left, int right) {
        if (left < right) {
           // Find the middle point
            int mid = (left + right) / 2;
           // Sort first and second halves
            mergeSort(array, left, mid);
            mergeSort(array, mid + 1, right);
            // Merge the sorted halves
            merge(array, left, mid, right);
        }
    }
    public static void main(String[] args) {
        int[] array = { 6, 5, 12, 10, 9, 1 };
        MergeSort ob = new MergeSort();
        ob.mergeSort(array, 0, array.length - 1);
        System.out.println("Sorted array:");
       System.out.println(Arrays.toString(array));
    }
}
```

Quick sort: Quick Sort is a sorting algorithm based on the divide and conquer that picks an element as a pivot and partitions the given array around the picked pivot by placing the pivot in its correct position in the sorted array.

Program:

```
class QuickSort {
```

```
// Method to find the partition position
 static int partition(int array[], int low, int high){
     int pivot = array[high];
     int i = (low - 1);
    for (int j = low; j < high; j++){
         if (array[j] <= pivot){
             i++;
             int temp = array[i];
             array[i] = array[j];
             array[j] = temp;
        }
    }
    int temp = array[i + 1];
    array[i + 1] = array[high];
    array[high] = temp;
    return (i + 1);
}
quickSort(int array[], int low, int high) {
     if (low < high) {
        int pi = partition(array, low, high);
        quickSort(array, low, pi - 1);
        quickSort(array, pi + 1, high);
    }
}
 public static void main(String args[]) {
     int[] data = { 8, 7, 2, 1, 0, 9, 6 };
    System.out.println("Unsorted Array");
    System.out.println(Arrays.toString(data)); int size = data.length;
```

```
        quickSort(data, 0, size - 1);
        System.out.println("Sorted Array in Ascending Order");
        System.out.println(Arrays.toString(data));
    }
}
```

2. Searching of array elements:

Linear search: Linear search is a method for searching for an element in a collection of elements. In linear search, each element of the collection is visited one by one in a sequential fashion to find the desired element. Linear search is also known as “sequential search”.

Program:

```
class LinearSearch {
    // This function returns index of element x in arr[]
      static int search(int arr[], int n, int x)
    {
        for (int i = 0; i < n; i++) {
            // Return the index of the element if the element is found
              if (arr[i] == x)
                  return i;
        }
        // return -1 if the element is not found
        return -1;
    }
    public static void main(String[] args)
    {
        int[] arr = { 3, 4, 1, 7, 5 };
          int n = arr.length;
          int x = 4;
          int index = search(arr, n, x);
          if (index == -1)
            System.out.println("Element is not present in the array");
          else
            System.out.println("Element found at position " + index);
    }
}
```

Binary search: Binary search is one of the searching techniques applied when the input is sorted here we are focusing on finding the middle element that acts as a reference frame whether to go left or right to it as the elements are already sorted.

Program:

```
// Java implementation of iterative Binary Search
class BinarySearch {
        // Returns index of x if it is present in arr[l....r], else return -1

        int binarySearch(int arr[], int l, int r, int x){

                while (l <= r) {
                        int mid = (l + r) / 2;
                        // If the element is present at the middle itself
                        if (arr[mid] == x) {
                                return mid;
                        // If element is smaller than mid, then it can only be present
                        //in left subarray so we decrease our r pointer to mid - 1
                        }
                        else if (arr[mid] > x) {
                                r = mid - 1;
                        // Else the element can only be present in right subarray so
                        //we increase our l pointer to mid + 1
```

```
                    } else {
                    l = mid + 1;
                    }
                }

            // We reach here when element is not present in array
                return -1;
            }
            // Driver method to test above
            public static void main(String args[])
            {
                BinarySearch ob = new BinarySearch();
                int arr[] = { 2, 3, 4, 10, 40 };
                int n = arr.length;
                int x = 10;
                int result = ob.binarySearch(arr, 0, n - 1, x);
                if (result == -1)
                    System.out.println("Element not present");
                else
                  System.out.println("Element found at index " + result);
            }
        }
```

3. <u>Access/Display of array elements</u>:

Accessing array elements in Java refers to the process of retrieving or modifying values stored at specific positions within an array. Each element in an array is identified by an index, starting from 0, and can be accessed using the array's name followed by the index in square brackets [].

<u>Program</u>:

```
public class Main {
    public static void main(String[] args) {
        // Define an array
        int[] myArray = {10, 20, 30, 40, 50};
         // Access elements by index
        int firstElement = myArray[0];
        // Access the first element
        int secondElement = myArray[1];
       // Access the second element
        int lastElement = myArray[myArray.length - 1];
      // Access the last element
      // Print the elements
        System.out.println("First element: " + firstElement);
        System.out.println("Second element: " + secondElement);
        System.out.println("Last element: " + lastElement);
    }
}
```

Assigning Array to Another Array

1. **Shallow Copy** **(Reference Assignment)**

- In a shallow copy, you simply assign one array to another, meaning both variables refer to the same underlying array. Any modification made to one array is reflected in the other.
- This is useful if you want two variables to refer to the same data.
- When you create the shallowCopyArray object by passing the original Array into the constructor, a shallow copy is made (i.e., the arr inside the MyArray class refers to the same memory location as originalArray)

2. **Deep Copy** **(Element-wise Copy)**

- In a deep copy, a new array is created, and all elements from the original array are copied into this new array. This ensures that the two arrays are independent, meaning changes in one will not affect the other.

- The MyArray (MyArray other) constructor performs a deep copy. It creates a new array and manually copies each element from the other object.
- Changes made to deepCopyArray do not affect originalArray or shallowCopyArray, demonstrating that they are independent after the deep copy.

<u>Sample Program</u>:

```
 class MyArray{
 private int[] arr;
 //Constructor to initialize array
 public MyArray(int[] arr){
       //Shallow copy (reference assignment)
       this.arr = arr;
 }
//Deep copy constructor
public MyArray(MyArray other){
    //Create a new array with the same length as the other array
    this.arr = new int[other.arr.length];
   //Copy each element from the other array to this array
   for(int i = 0; i<other.arr.length; i++){
         this.arr[i] = other.arr[i];
   }
}
//Method to display array elements
public void display(){
  for(int i: arr){
         System.out.print(i + " ");
        }
     System.out.println();
   }
   //Method to modify an element of the array
```

```
     public void setElement(int index, int value){
         if (index >= 0 && index < arr.length){
              arr[index] = value;
        } else {
             System.out.println(“Index out of bounds”);
        }
     }
 }
public class Main
{
    public static void main(String[] args)
   {
    //Create an original array
    int[] originalArray = {1, 2, 3, 4, 5};
    //Creating an object with shallow copy (reference assignment)
    MyArray shallowCopyArray = new MyArray(originalArray);
    //Displaying the original array
    System.out.println(“Original Array (before modification):”);
    shallowCopyArray.display();
    //Modifying the original array
    originalArray[0] = 10;
    //Displaying the original array after modification
    System.out.println(“Original Array (after modification):”);
    shallowCopyArray.display();
    //Creating an object with deep copy
    MyArray deepCopyArray = new MyArray(shallowCopyArray);
    //Displaying the deep copy array
    System.out.println(“Deep Copy Array(before modification):”);
    deepCopyArray.display();
    //Modifying the deep copy array
    deepCopyArray.setElement(0,20);
    //Displaying the deep copy array after modification
    System.out.println("Deep Copy Array(after modification):");
    deepCopyArray.display();
    //Showing that the original array is not affected by changes in the deep copy
    array
    System.out.println("Original Array (after deep copy modification):");
    shallowCopyArray.display();
  }
}
```

Output:

```
Original Array (before modification):
1 2 3 4 5
Original Array (after modification):
10 2 3 4 5
Deep Copy Array (before modification):
10 2 3 4 5
Deep Copy Array (after modification):
20 2 3 4 5
Original Array (after deep copy modification):
10 2 3 4 5
```

Dynamic Change of Array Size

- In Java, arrays have a fixed size once they are created, so you can't directly change their size. However, you can use other dynamic data structures like ArrayList, which is the part of the Java Collections Framework and provides a dynamically resizing array.
- The ArrayList will automatically resize when elements are added or removed.

Methods:

- add(): To add elements.
- remove(): To remove elements.
- set(): To update elements at a specific index.
- get(): To access elements.
- size(): To get all current size of the array.

Sample Program:

```
import java.util.ArrayList;
public class Main{
    public static void main(String[] args){
        //Creating an ArrayList
        ArrayList<Integer>dynamicArray = new ArrayList<>();
        //Adding elements
        dynamicArray.add(10);
        dynamicArray.add(20);
        dynamicArray.add(30);
        //Accessing elements
        System.out.println("Elements at index 1: "+ dynamicArray.get(1));
        //Changing elements
        dynamicArray.set(1,50):
        System.out.println("Elements at index 1 after modification: " +
                                                        dynamicArrayget(1));
        //Removing elements
        dynamicArray.remove(2);
        //Size of ArrayList
        System.out.println("Size of ArrayList: " +dynamicArray.size());
        //Ilerating through the ArrayList
        for (int num : dynamicArray){
            System.out.println(num);
        }
    }
}
```

Output:

```
Element at index 1: 20
Element at index 1 after modification: 50
Size of ArrayList: 2
10
50
```

Two-Dimensional Array

- A Two-dimensional array is essentially an array of arrays.
- It can be said as a table with rows and columns.
- Each element in the array is accessed using two indices, one for the row and one for the column.
- Syntax: datatype[][] arrayname = new datatype[rows][columns];
- Ex: int[][] matrix = new int[3][4];

Program with Regular 2D Array

Program:

```
public class Regular2DArray {
    public static void main(String[]args) {
        // Declare a 2D array with fixed rows and columns
        int[][] array = new int[3][3];
       // Initialize the array with values
        array[0][0] = 1;
        array[0][1] = 2;
        array[0][2] = 3;
        array[1][0] = 4;
        array[1][1] = 5;
```

```
         array[1][2] = 6;

         array[2][0] = 7;

         array[2][1] = 8;

         array[2][2] = 9;

        int rows = 3;

        int cols = 3;

        // Print the 2D array

        System.out.println("Regular 2D Array:");

         for (int i = 0; i < rows; i++) {

             for (int j = 0; j < cols; j++) {

                System.out.print(array[i][j] + " ");

            }

            System.out.println();   // Move to the next line after each row

        }

    }

}
```

Output:

```
Regular 2D Array:

1 2 3

4 5 6

7 8 9
```

Program with 2D Array Varying Lengths

Program:

```
public class VaryingLengthArray {

    public static void main(String[] args) {

        // Declare a 2D array with varying row lengths
    }

         int[][] array = new int[3][];

        // Initialize each row with different lengths

         array[0] = new int[2];  // First row has 2 columns

         array[1] = new int[3];  // Second row has 3 columns

         array[2] = new int[1];  // Third row has 1 column

        // Assign values to the array

         array[0][0] = 1;

         array[0][1] = 2;

         array[1][0] = 3;

         array[1][1] = 4;

         array[1][2] = 5;

         array[2][0] = 6;

        int[] rowSizes = {2, 3, 1};

         // Print the 2D array

        System.out.println("2D Array with Varying Lengths:");

        for (int i = 0; i < 3; i++) {

            for (int j = 0; j <rowSizes[i]; j++) {

                System.out.print(array[i][j] + " ");

            }

            System.out.println();  // Move to the next line after each row

        }

    }

}
```

Output:

```
2D Array with Varying Lengths:
1 2
3 4 5
6
```

Three-dimensional Arrays

- A Three-dimensional array is an array of two-dimensional arrays.
- It can be visualized as a cube or a stack of tables.
- Each element is accessed using three indices, one for the table, one for the row, and one for the column.
- Syntax: datatype[][][] arrayName = new datatype[size1][size2][size3];
- Ex: int[][][] myArray = new int[2][3][4];

Program with Regular 3D array

Program:

```
public class ThreeDArray{
    public static void main(String args[]){
        //Create a 3D array
        int[][][] array3D = new int[2][3][4];
        //Initialize the array with values
        int value = 1;
        for(int i=0; i<array3D.length; i++){
            for(int j=0; j<array3D.[i].length; j++){
                for(int k=0; k<array3D.[i][j].length; k++){
                    array3D[i][j][k] = value++;
                }
            }
        }
        //Print the 3D array
        for(int i=0; i<array3D.length; i++){
            for(int j=0; j<array3D.[i].length; j++){
                for(int k=0; k<array3D.[i][j].length; k++){
                        System.out.println(array3D[i][j][k]+ " ");
                }
                System.out.println();
            }
            System.out.println();
        }
    }
}
```

<u>Program with 3D Array Varying Lengths</u>

Program:

```
class ThreeDVaryingLength{
    public static void main(String args[]){
        int arr[][][] = new int[3][2][];
        // Assign values to the array
        arr[0][0] = new int[3];
        arr[0][1] = new int[4];
        arr[1][0] = new int[2];
        arr[1][1] = new int[3];
        arr[2][0] = new int[5];
        arr[2][1] = new int[4];
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter array elements: ");
        for(int i=0; i<3; i++){
            for(int j=0; j<2; j++){
                for(int k=0; k<array[i][j].length; k++){
                        array[i][j][k] = sc.nextInt();
                }
            }
        }
        //Print the 3D array
        for(int i=0; i<3; i++){
            for(int j=0; j<2; j++){
                for(int k=0; k<array[i][j].length; k++){
                    System.out.println(array[i][j][k]+ " ");
                }
                System.out.println();
            }
            System.out.println();
        }
    }
}
```

Arrays and Vectors

<u>Arrays</u>:

An array is a fundamental data structure available in most programming languages, including Java that stores a collection of elements. These elements are of the same data type and are stored in contiguous memory locations. Arrays are used to organize data so that a related set of values can be easily sorted or searched.

<u>Characteristics of an array</u>:

- Once an array is declared, its size is fixed and cannot be altered. This means you must know the maximum number of elements you plan to store in an array at the time of its declaration.
- All elements in an array must be of the same data type. For example, an integer array can only store integers, and a string array can only store strings.

- Arrays provide random access to elements, which means any element in the array can be accessed directly using its index number. The index of an array usually starts at 0, making the first element accessible at index 0, the second at index 1, and so on.
- In memory, an array's elements are allocated in contiguous memory locations, which enables efficient access to the elements.

Vectors:

A Vector refers to a dynamic array that can grow or shrink in size as needed. It is part of the Java Collections Framework and implements the List interface, providing a way to store elements sequentially, similar to an array, but with the added flexibility of being able to adjust its size dynamically.

Characteristics of vectors:

- Unlike arrays, Vectors can grow or shrink dynamically, accommodating more elements than were initially declared or reducing the storage used based on the elements it currently holds.
- All methods of the Vector class are synchronized. This means a Vector is thread-safe and can be used in concurrent scenarios without additional synchronization code. However, this also means that Vector operations may have more overhead compared to similar unsynchronized collections, like ArrayList.
- Vectors allow duplicate elements and also null values, similar to other list implementations.
- Like arrays, Vectors provide random access to their elements, allowing for fast retrieval of elements based on their index position.

Difference between arrays and vectors

S.No	Arrays	Vectors
1.	Arrays are not synchronised	Vectors are synchronised
2.	Array increments 50% of the current array size if the number of elements exceeds its capacity.	Vector increments 100% means doubles the array size if the total number of elements exceeds its capacity.
3.	Array is not a legacy class. It is introduced in JDK 1.2.	Vector is a legacy class

4.	Array is fast because it is non-synchronized.	Vector is slow because it is synchronized, i.e., in a multithreading environment, it holds the other threads in a runnable or non-runnable state until the current thread releases the lock of the object.
5.	Array uses the Iterator interface to traverse the elements.	A Vector can use the Iterator interface or Enumeration interface to traverse the elements.
6.	Array performance is high	Vector performance is low
7.	Arrays are resizable and dynamic	Vectors are a form of dynamic array
8.	It is not not an inheritance class	It is an inheritance class
9.	It requires creating a new array and copying elements.	It is resizable with methods like add() and remove().

Example Program for Vectors:

```
import java.util.*;

class Abc

{

    void display()
   {
        System.out.print(“HELLO”);

   }
}

class VectorMain {

    public static void main(String[ ] args){

        String s = “CSE”;

        Abc abc1 = new Abc();

        Abc abc2 = new Abc();

        Vector v = new Vector();

        v.addElement(abc1);
        v.addElement(abc2);

        v.addElement(S);

        v.removeElement(abc2);

        Iterator i = v.iterator();

        while(i.hasNext());

        {

                System.out.print(i.next()); //To display

        }
    }
}
```

Sorting of Arrays – Covered in the topic “Operations on Array Elements”
Class Arrays – Arrays of Array **i.e** refer either 2D array-variable names or 3D array-variable names topics
Search for Values in Arrays – Covered in the topic “Operations on Array Elements”