# Lecture

**NOTE:** FOR FURTHER DETAILS AND MORE COMPREHENSIVE STUDY, PLEASE SEE RECOMMENDED BOOKS OR INTERNET.

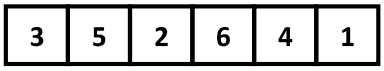
**IMPORTANT:**
BEFORE GOING DEEP INTO DATA STRUCTURE, YOU SHOULD HAVE A GOOD KNOWLEDGE OF PROGRAMMING EITHER IN C OR IN JAVA.

## Arrays

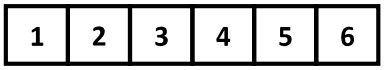
An array is an aggregate data structure that is designed to store a group of objects of the same types. Arrays can hold primitives as well as references. The array is the most efficient data structure for storing and accessing a sequence of objects.

### Organization

An Array can store elements sequentially but neither array is sequential access nor like a list which store the elements in an organizational order (e.g. Ascending or Descending). By default Arrays do not have an organizational system; however we can organize the data in array using sorting algorithms e.g. bubble sort, quick sort, merge sort etc. The problem is arrays are random access data structures and any elements can be accessed at a given time. To keep the organizational order of the array; it must be accessed in a sequential way after the sorting has been done on the array.
Example: Consider an array of 6 integer elements. It is has no organizational order.




If the sorting is applied and elements are sorted then the array becomes:




However, to keep the organizational system intact the access must be in sequential order.

### Creation

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

```
dataType[]  arrayName;   // preferred way.
```

Or,

```
dataType  arrayNamer[];  //  works but not preferred.
```

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

The following code snippets are examples of this syntax:

```
double[] myList;          // preferred way.
```

Or,

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

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

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

The above statement does two things:

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

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

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

Alternatively you can create arrays as follows:

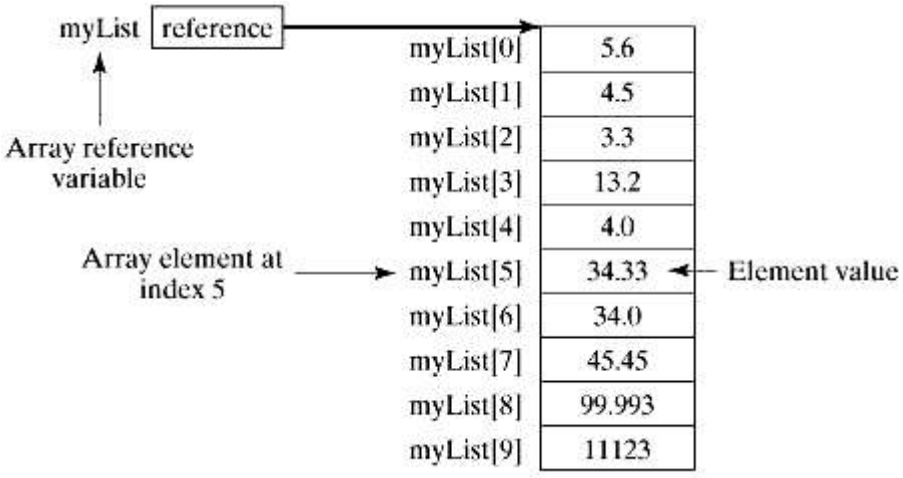
```
dataType[] arrayName = {value0, value1, ..., valuek};
```

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

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

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

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



## Memory Representation

One dimensional arrays are the simple form of arrays. In previous examples we have shown how to initialize and declare one dimensional array, now we will discuss how it is represented in the memory. As we know now 1-d array are linear array in which elements are stored in the successive memory locations. The element at the very first position in the array is called its base address. Now consider the following example:

```
int arr[5];
```

| Index | Address | Values |
|---|---|---|
| | | USED |
| arr[0] | 100 | 34 |
| arr[1] | 104 | 78 |
| arr[2] | 108 | 98 |
| arr[3] | 112 | 45 |
| arr[4] | 116 | 56 |
| | | EMPTY |

Here we have defined an array of five elements of integer type whose first index element is at base address 100. i.e, the element arr[0] is stored at base address 100. Now for calculating the starting address of the next element i.e. of a[1], we can use the following formula:

***Address of nth element = Base Address + (Size of Data type (C) * index of element n)***

Here ***C*** is constant integer and vary according to the data type of the array, for e.g. for integer the value of ***C*** will be 4 bytes, since an integer occupies 4 bytes of memory.

Now, we can calculate the starting address of second element of the array as:

arr[1] = 100 + (4 * 1) = 104

Thus starting address of second element of array is 104
Similarly other addresses can be calculated in the same manner as:

arr[2] = 100 + (4 * 2) = 108
arr[3] = 100 + (4 * 3) = 112
arr[4] = 100 + (4 * 4) = 116

## The Array Class

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

Following is the examples of some functions with description.

| S# | Methods with Description |
|---|---|
| 1 | **public static int binarySearch(Object[ ] a, Object key)**<br>Searches the specified array of Object ( Byte, Int , double, etc.) for the specified value using the binary search algorithm. The array must be sorted prior to making this call. This returns index of the search key, if it is contained in the list; otherwise, (-(insertion point + 1). |
| 2 | **public static boolean equals(long[ ] a, long[ ] a2)**<br>Returns true if the two specified arrays of longs are equal to one another. Two arrays are considered equal if both arrays contain the same number of elements, and all corresponding pairs of elements in the two arrays are equal. This returns true if the two arrays are equal. Same method could be used by all other primitive data types (Byte, short, Int, etc.) |
| 3 | **public static void fill(int[ ] a, int val)**<br>Assigns the specified int value to each element of the specified array of ints. Same method could be used by all other primitive data types (Byte, short, Int etc.) |
| 4 | **public static void sort(Object[ ] a)**<br>Sorts the specified array of objects into ascending order, according to the natural ordering of its elements. Same method could be used by all other primitive data types ( Byte, short, Int, etc.) |
| ... | ... |

# Lecture

**NOTE:** FOR FURTHER DETAILS AND MORE COMPREHENSIVE STUDY, PLEASE SEE RECOMMENDED BOOKS OR INTERNET.

**IMPORTANT:**
BEFORE GOING DEEP INTO DATA STRUCTURE, YOU SHOULD HAVE A GOOD KNOWLEDGE OF PROGRAMMING EITHER IN C OR IN JAVA.

## Multidimensional Arrays

Array having more than one subscript variable is called multidimensional array. Multidimensional array is also called as matrix or tables.

### Two Dimensional Array

1. Two Dimensional Array requires **Two Subscript Variables**
2. Two Dimensional Array stores the values in the form of matrix.
3. One Subscript Variable denotes the "**Row**" of a matrix.
4. Another Subscript Variable denotes the "**Column**" of a matrix.

| | Column 0 | Column 1 | Column 2 | Column 3 |
|---|---|---|---|---|
| Row 0 | a[ 0 ][ 0 ] | a[ 0 ][ 1 ] | a[ 0 ][ 2 ] | a[ 0 ][ 3 ] |
| Row 1 | a[ 1 ][ 0 ] | a[ 1 ][ 1 ] | a[ 1 ][ 2 ] | a[ 1 ][ 3 ] |
| Row 2 | a[ 2 ][ 0 ] | a[ 2 ][ 1 ] | a[ 2 ][ 2 ] | a[ 2 ][ 3 ] |

### Declaration and use of Two Dimensional Array

```
int arr[3][4];
```

*Use :*

```
for(i=0; i<row; i++)
{
   for(j=0; j<col; j++)
   {
     System.out.println(arr[i][j]);
   }
}
```

What is meant by the program in the above array is:

1. Matrix have 3 rows ( i takes value from 0 to 2 )
2. Matrix have 4 columns ( j takes value from 0 to 3 )
3. Above Matrix 3×4 matrix will have 12 blocks having 3 rows and 4 columns.
4. Name of 2-D array is '**arr**' and each block is identified by the row & column number.
5. Row number and Column number starts from 0.

| Cell Location | Meaning |
|---|---|
| `arr[0][0]` | 0th Row and 0th Column |
| `arr[0][1]` | 0th Row and 1st Column |
| `arr[0][2]` | 0th Row and 2nd Column |
| `arr[0][3]` | 0th Row and 3rd Column |
| `arr[1][0]` | 1st Row and 0th Column |
| `arr[1][1]` | 1st Row and 1st Column |
| `arr[1][2]` | 1st Row and 2nd Column |
| `arr[1][3]` | 1st Row and 3rd Column |
| `arr[2][0]` | 2nd Row and 0th Column |
| `arr[2][1]` | 2nd Row and 1st Column |
| `arr[2][2]` | 2nd Row and 2nd Column |
| `arr[2][3]` | 2nd Row and 3rd Column |

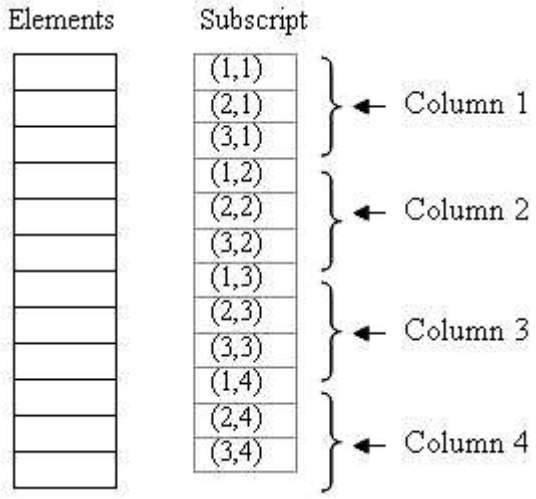
## Representing 2D array

A 2D array's elements are stored in continuous memory locations. It can be represented in memory using any of the following two ways:

***1. Column-Major Order***
***2. Row-Major Order***

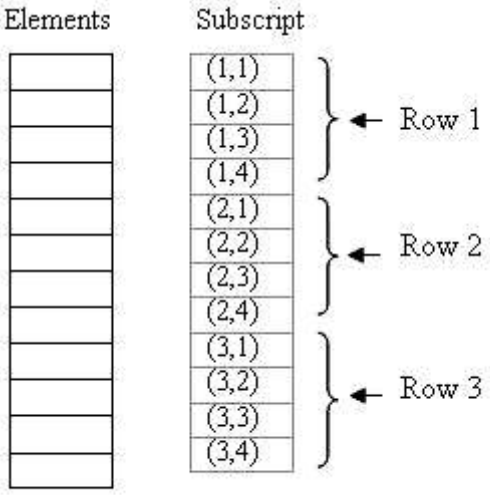
### *1. Column-Major Order:*

In this method the elements are stored column wise, i.e. m elements of first column are stored in first m locations, m elements of second column are stored in next m locations and so on. A 3 x 4 array will stored as below:




### *2. Row-Major Order:*

In this method the elements are stored row wise, i.e. n elements of first row are stored in first n locations, n elements of second row are stored in next n locations and so on. A 3 x 4 array will stored as below:


Elements
Subscript
(1,1)
(1,2)
(1,3)
(1,4)
(2,1)
(2,2)
(2,3)
(2,4)
(3,1)
(3,2)
(3,3)
(3,4)
Row 1
Row 2
Row 3