Arrays

Java provides a data structure, the **array**, which stores a fixed-size sequential collection of elements of the same type. An **array** is used to store a collection of data, but it is often more useful to think of an **array** as a collection of variables of the same type.

Instead of declaring individual *variables*, such as number0, number1, …, and number99, you declare one array *variables* such as numbers and use numbers[0], numbers[1], and …, numbers[99] to represent individual variables.

This section introduces how to declare **array** variables, create arrays, and process arrays using indexed variables.

Declaring Array Variables

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[] arrayRefVar;
```

> **Example:**
> `double[] myList;`

Creating Arrays

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

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

The above statement does two things –

- It creates an *array* using ***new dataType[arraySize]***.
- 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[] arrayRefVar = new dataType[arraySize];
```

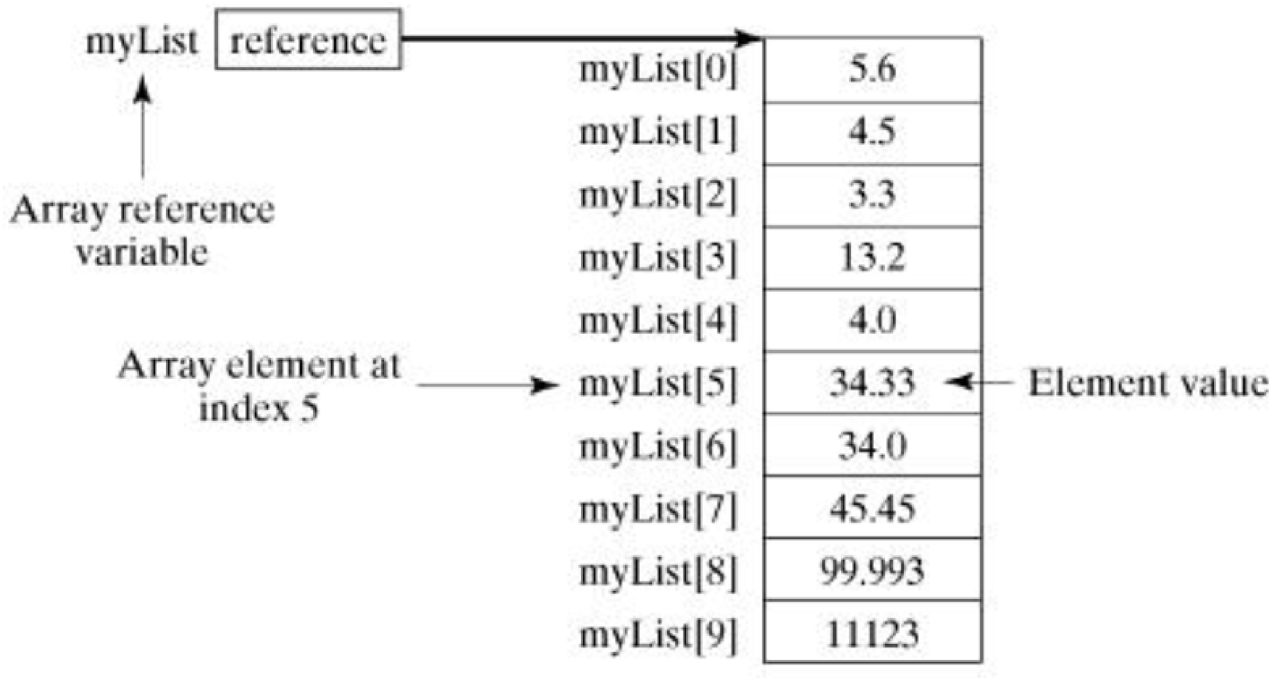
Alternatively you can create arrays as follows –

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

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

> **Example:**
> Following statement declares an ***array*** variable, *myList*, and 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.



Processing Arrays

When processing array elements, we often use either **for** loop or **foreach** loop because all of the elements in an array are of the same type and the size of the array is known.

> **Example:**
> Here is a complete example showing how to create, initialize, and process arrays –

```
public class TestArray {
   public static void main(String[] args) {
       double[] myList = { 1.9, 2.9, 3.4, 3.5 };
       // Print all the array elements
       for (int i = 0; i < myList.length; i++) {
           System.out.println(myList[i] + " ");
       }
       // Summing all elements
       double total = 0;
       for (int i = 0; i < myList.length; i++) {
           total += myList[i];
       }
       System.out.println("Total is " + total);
       // Finding the largest element
       double max = myList[0];
       for (int i = 1; i < myList.length; i++) {
            if (myList[i] > max)
                max = myList[i];
        }
        System.out.println("Max is " + max);
    }
}
```

> **Output:**
> 1.9
> 2.9
> 3.4
> 3.5
> Total is 11.7
> Max is 3.5

The foreach Loops

JDK 1.5 introduced a new for loop known as foreach loop or enhanced for loop, which enables you to traverse the complete array sequentially without using an index variable.

> **Example:**
> The following code displays all the elements in the array myList –

```
 public class TestArray {
   public static void main(String[] args) {
       double[] myList = { 1.9, 2.9, 3.4, 3.5 };
       // Print all the array elements
       for (double element : myList) {
           System.out.println(element);
       }
   }
}
```

> **Output:**
> 1.9
> 2.9

> 3.4
> 3.5

Passing Arrays to Methods

Just as you can pass primitive type values to methods, you can also pass arrays to methods. For example, the following method displays the elements in an int array −

> **Example:**

```
public static void printArray(int[] array) {
   for (int i = 0; i < array.length; i++) {
      System.out.print(array[i] + " ");
   }
}
```

You can invoke it by passing an array. For example, the following statement invokes the printArray method to display 3, 1, 2, 6, 4, and 2 −

```
printArray(new int[]{3, 1, 2, 6, 4, 2});
```

Returning an Array from a Method

A method may also return an array. For example, the following method returns an array that is the reversal of another array −

```
public static int[] reverse(int[] list) {
   int[] result = new int[list.length];

   for (int i = 0, j = result.length - 1; i < list.length; i++, j--) {
      result[j] = list[i];
   }
   return result;
}
```

Multi Dimensional Arrays

We have seen that an array can hold references to objects and an array itself is a reference type so an array of arrays seems quite plausible. These are referred to as multidimensional arrays. We can arrays of two, three or more dimensions. We will look at multi dimensional arrays, with special focus on two dimensional arrays which are quite frequently used. They are generally used to represent matrices. But unlike a mathematical matrix, there are a few differences which we will see now.

We have represented an array type using a pair of brackets. Two dimensional arrays are in a similar way represented by two such pairs of brackets and an N dimensional array is represented by using N such pairs of brackets. The following statement creates a two dimensional array of integers, which contains 3 arrays containing 4 integers each.

```
int[][] a=new int[3][4];
```

We can also think of this array as a 3*4 matrix in mathematics, with each row of the matrix being a Java array. You might also think of it in the other way, also as a 4*3 matrix, but we generally use the first notation.

Elements of this array are accessed by specifying the index numbers, here two of them. The first representing the array number and the second representing the index element in that particular array.

```
a[0][2] = 34;
```

A two dimensional array can also be initialized using an array initialiser in the following way. This paints a better picture of a 2D array as an array of arrays.

```
int[][] d = { { 1,5,74,2}, {4,68,45,65},{5,0,34,54}}:
```

The above array a[][] contains three arrays { 1,5,74,2}, {4,68,45,65} and {5,0,34,54}. This particular 2D array contains equal number of elements in each of its sub arrays. We can, if we wish to, create 2D arrays which have different number of elements in its sub arrays.

```
int[][] c = { { 4,56,7}, {1}, {45,78} };
```

The above two dimensional array contains three arrays. The first of these has three elements, the second has a single element and the last of these has two elements. We can also create such non symmetric arrays using the new keyword in the following way:

```
int[][] b = new int[3][];
b[0] = new int[3];
b[1] = new int[1];
b[2] = new int[2];
```

The above three sets of statements also creates an array space b identical to that of a created earlier using an array initializer. Observe that in the first statement, we haven't specified any integer in the second pair of square brackets following [3]. b[][] is reference type and so are b[0], b[1] and b[2]. However, b[0][0], b[0][1] …. are of primitive type int. These things are to be dealt with carefully when we pass an array, a sub array or an element of the array as parameters.

The length of a two dimensional or a multi-dimensional array gives the number of arrays it contains. For example: b.length gives 3 while b[0].length gives 3, b[1].length gives 1 and so on. All these facts are easy to assimilate if we consider two dimensional arrays to be an array of arrays rather than as a mathematical matrix.

We manipulate multi-dimensional arrays using nested loops. For example, the following code snippet is used to print the elements of an array as a matrix. The code works even when the rows of the array are of different lengths.

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

Searching a two dimensional array is performed using a linear search. The labelled break statement is used to break out of the loop once the element is found.

```
outer:
 for ( int i=0; i<a.length; i++) {
     for(int j=0; j< a[i].length; j++) {
         if ( a[i][j] == target) {
             System.out.println("found it at ["+i+"]["+j+"]");
             break outer;
         }
     }
 }
```

Sorting isn't generally performed on multidimensional arrays since we generally use them to represent a physical situation rather than as data structures to hold information. For example, we may use a three dimensional array to hold Student objects in a more realistic way. We have different cities and each city has different schools which has students. Let each city, school and Student have a code. A Student object in that case can be accessed more conveniently as s[city code][school code][student code]. This 3D representation gives a more realistic picture than a 1D array of Students.

The Arrays 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.

No.	Method & Description
1	**public static int binarySearch(Object[] a, Object key)** 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, it returns (– (insertion point + 1)).
2	**public static boolean equals(long[] a, long[] a2)** 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)** Assigns the specified int value to each element of the specified array of ints. The same method could be used by all other primitive data types (Byte, short, Int, etc.)
4	**public static void sort(Object[] a)** Sorts the specified array of objects into an ascending order, according to the natural ordering of its elements. The same method could be used by all other primitive data types (Byte, short, Int, etc.)

Assignment

Write a program to generate numbers in the following format. Hint: Use two dimensional array to hold these numbers. Also, use `System.out.printf("%3d", your_int_variable)` to print.

```
  0   1   2   3   4   5   6   7   8   9
 10  11  12  13  14  15  16  17  18  19
 20  21  22  23  24  25  26  27  28  29
 30  31  32  33  34  35  36  37  38  39
 40  41  42  43  44  45  46  47  48  49
 50  51  52  53  54  55  56  57  58  59
 60  61  62  63  64  65  66  67  68  69
 70  71  72  73  74  75  76  77  78  79
 80  81  82  83  84  85  86  87  88  89
 90  91  92  93  94  95  96  97  98  99
```

Further reading

Java Arrays