Lecture 08 – Java Array

Arrays are ordered collections of identical objects that can be accessed via an index. In C and C++, an array is implemented as a contiguous block of memory accessed via pointers to the elements. Java arrays have their own behavior that is encapsulated into their definition when the compiler creates them. Since arrays are objects, array variables behave like class-type variables.

It is important to distinguish between the array variable and the array instance to which it refers. Declaring an array only declares the array variable. To instantiate the array, the new operator must be used with the [] operator to enclose the array size. The array size can be any integer expression. The following code declares an array variable and initializes it to null:

```
char arr[] = null;
```

This can also be written as:

```
char[] arr = null;
```

To declare an array variable and initialize it to refer to a new instance, we use the following syntax:

```
int length = 60;

char[] arr = new char [length];
```

Array Constants

The declaration of an array must include its type, but not its size. Arrays may be initialized with conventional initialization syntax. An array initializer is either an expression (such as a new expression) that evaluates to the correct array type, or a list of initial element values enclosed in { }. This latter notation is borrowed from C/C++. In Java, this is called an *array constant*. Any expression can be used for the initial array element values. (C requires initializers be compiler-time calculable; both C++ and Java allow initializers to be run-time calculable). The following array constant provides initial values for its elements:

```
int[] array = {1, 3, 4, 15, 0};
```

Each element of an array of class-type objects behaves like a class-type variable:

- Elements may be allocated by new, making them initially null:

- Elements may be initialized in an array constant:

```
Box b = new Box (2.2, 3.3);

Box[] boxes = { b, new Box (1.1, 2.2) };
```

Note: Arrays are neither a true class (there is no class called Array), nor a built-in type (Arrays have members like a class). Arrays in Java are in between a class and a data type although they are implemented as a class internally by Java.

Using Arrays

Array elements can be accessed using C notation (an index in square brackets, [n]), where the index must be an integer. Like arrays in C, arrays start with index number 0, not 1. The index can be thought of as an offset from the beginning of the array. The index may not be less than zero or greater than the declared size. If the array is declared size n, the index must be in the range 0 to n-1.

Unlike C, arrays are not implemented as pointers, so programming techniques like negative array indexing are not allowed. Any attempt to index an array with an illegal value will cause an exception to be thrown. All arrays have a data field called **length** that indicates its size. This value is read-only, and contains the number of elements in the array:

```
int arr[] = new int[100];

for (int i = 0; i < arr.length; i++) {

    arr[i] = i * i;

}
```

Since length is read-only, its value cannot be changed manually:

```
arr.length = 20; // error
```

Sorting of a One-Dimensional Array

If the elements of an array are unsorted, we can apply any standard sorting technique to sort them in ascending or descending order. An example of such a sorting technique is Bubble Sort method. If this sorting technique takes an array of 'n' elements then it requires at most n-1 iterations to sort them ascending order. In a series of n-1 iterations, the successive elements, list[index] and list[index + 1] of list are compared. If list[index] is greater than list[index + 1], then the elements of list[index] and list[index + 1] are

swapped. This process is repeated through until no swaps are necessary. On average, for an array with length 'n', the method takes (n*(n-1))/2 key comparisons. The time complexity of this technique is $O(n^2)$. The code for this technique is shown below:

```
//Listing 1

public class TestBubbleSort {
    public static void main(String[] args) {
        //Random set of numbers for example.
        int unsortedArray[] = {10, 97, 6, 23, 0, -45, 697, -100, 1, 0};
        int i;

        //Pass the array to be sorted and its length.
        bubbleSort(unsortedArray, unsortedArray.length);

        //Just to show that it worked.
        System.out.println("After sorting, the list elements are: ");

        for(i=0; i<unsortedArray.length; i++) {
            System.out.print(unsortedArray[i] + " ");
        }
    }
    private static void bubbleSort(int[] unsortedArray, int length) {
        int temp, counter, index;

        //Loop once for each element in the array.
        for(counter=0; counter<length-1; counter++)
        {
            //Once for each element, minus the counter.
            for(index=0; index<length-1-counter; index++)
            {
                //Test if need a swap or not.
                if(unsortedArray[index] > unsortedArray[index+1])
                {
                    //These three lines just swap the two elements:
                    temp = unsortedArray[index];
                    unsortedArray[index] = unsortedArray[index+1];
                    unsortedArray[index+1] = temp;
                }
            }
        }
    }
}
```

Output

After sorting, the list elements are:
-100 -45 0 0 1 6 10 23 97 697

Multidimensional Arrays

So far, we've looked at simple arrays that hold their data in a list. However, most programming languages also support multidimensional arrays, which are more like tables than lists. For example, take a look at Figure 1. The first array in the figure is a one-dimensional array, which is like the arrays we've used so far in this chapter. The next type of array in the figure is two-dimensional, which works like the typical spreadsheet type of table we're used to seeing.

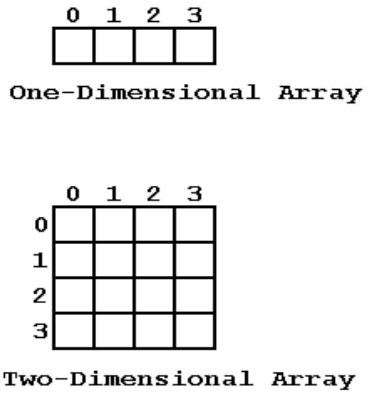

Figure 1: Arrays can have more than one dimension

Although Java doesn't support multidimensional arrays in the conventional sense, it does enable us to create arrays of arrays, which amount to the same thing. For example, to create a two-dimensional array of integers like the second array in Figure 7.1, we might use a line of code like this:

```
int table[][] = new int[4][4];
```

This line of Java code creates a table that can store 16 values-four across and four down. The first subscript selects the column and the second selects the row. To initialize such an array with values, we might use the lines shown in Listing A (see below), which would give us the array shown in Figure 2.

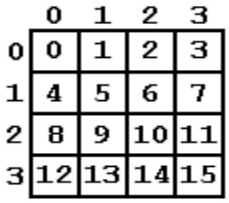

Figure 2: Here's the two-dimensional array as initialized in Listing A

Listing A: Initializing a Two-Dimensional Array.

```
table[0][0] = 0;   table[1][0] = 4;   table[2][0] = 8;    table[3][0] = 12;
table[0][1] = 1;   table[1][1] = 5;   table[2][1] = 9;    table[3][1] = 13;
table[0][2] = 2;   table[1][2] = 6;   table[2][2] = 10;   table[3][2] = 14;
table[0][3] = 3;   table[1][3] = 7;   table[2][3] = 11;   table[3][3] = 15;
```

Creating a Two-Dimensional Array

Suppose that we need a table-like array that can hold 12 integers in 3 rows and 4 columns. First, we'd declare the array like this:

```
int data[][];
```

After declaring the array, we need to create it in memory, like this:

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

The last step is to initialize the array, probably using nested for loops:

```
for (int x=0; x<3; ++x) {
     for (int y=0; y<4; ++y) {
            data[x][y] = 0;
     }
}
```

These lines initialize the data[][] array to all zeroes.

```
//Listing 2

public class Array2D {
  public static void main(String[] args) {
    int data[][];
    data = new int[3][4];
    for (int x=0; x<3; ++x) {
         for (int y=0; y<4; ++y) {
            data[x][y] = 0;
         }
    }
    System.out.println("Row Size = " + data.length);
    System.out.println("Column Size = " + data[0].length);
  }
}
```

Output

```
Row Size = 3
Column Size = 4
```

Accessing Elements of a Two-Dimensional Array

Suppose that a two dimensional array of int values is initialized, and all the elements of the array are accessed using nested for loops. See the example below:

```
//Listing 3

public class Test2Darray {
  public static void main(String args[]) {
    int[][] a = {
      { 1, 2, 3, 4 },
      { 5, 6, 7, 8 }
    };
    int rows = a.length;
    int cols = a[0].length;

    System.out.println("Printing Matrix a");
    System.out.println("-----------------");
    System.out.println("array["+rows+"]["+cols+"] = {");
    for (int i=0; i<rows; i++) {
      System.out.print("{");
      for (int j=0; j<cols; j++)
        System.out.print(" " + a[i][j] + ",");
      System.out.println("},");
    }
    System.out.println("};");
    System.out.println();
  }
}
```

Output

```
Printing Matrix a
-----------------
array[2][4] = {
{ 1, 2, 3, 4,},
{ 5, 6, 7, 8,},
};
```

Matrix Multiplication of two 2-D Arrays

The example below shows the matrix multiplication of two 2-D arrays.

```
//Listing 4
public class Matrix {

  /* Matrix-multiply two arrays together.
   * The arrays MUST be rectangular.
  */
  public static int[][] multiply(int[][] m1, int[][] m2) {
    int m1rows = m1.length;
    int m1cols = m1[0].length;
    int m2rows = m2.length;
    int m2cols = m2[0].length;
    if (m1cols != m2rows)
      throw new IllegalArgumentException("matrices don't match: "
                                         + m1cols + " != " + m2rows);

    int[][] result = new int[m1rows][m2cols];
```

```
    // multiply
    for (int i=0; i<m1rows; i++)
      for (int j=0; j<m2cols; j++)
        for (int k=0; k<m1cols; k++)
        result[i][j] += m1[i][k] * m2[k][j];

    return result;
  }

  /** Matrix print.
   */
  public static void mprint(int[][] a) {
    int rows = a.length;
    int cols = a[0].length;
    System.out.println("array["+rows+"]["+cols+"] = {");
    for (int i=0; i<rows; i++) {
      System.out.print("{");
      for (int j=0; j<cols; j++)
        System.out.print(" " + a[i][j] + ",");
      System.out.println("},");
    }
    System.out.println("};");
    System.out.println();
  }

  public static void main(String[] argv) {
    int x[][] = {
      { 3, 2, 3 },
      { 5, 9, 8 },
    };
    int y[][] = {
      { 4, 7 },
      { 9, 3 },
      { 8, 1 },
    };
    int z[][] = Matrix.multiply(x, y);
    System.out.println("Matrix One:-> Printing Matrix x");
    System.out.println("--------------------------------");
    Matrix.mprint(x);
    System.out.println("Matrix Two:-> Printing Matrix y");
    System.out.println("--------------------------------");
    Matrix.mprint(y);
    System.out.println("Mul Result:-> Printing Matrix z");
    System.out.println("--------------------------------");
    Matrix.mprint(z);
  }
}
```

Output

```
Matrix One:-> Printing Matrix x
---------------------------------
array[2][3] = {
{ 3, 2, 3,},
{ 5, 9, 8,},
};
```

Matrix Two:-> Printing Matrix y
---
array[3][2] = {
{ 4, 7,},
{ 9, 3,},
{ 8, 1,},
};

Mul Result:-> Printing Matrix z
---
array[2][2] = {
{ 54, 30,},
{ 165, 70,},
};