We assume you know about arrays in some language, like Python, Matlab, C, and so on. Arrays in Java are similar, but there are differences from language to language.

**One-dimensional arrays**

For any type T, T[] (pronounced “T-array”) is the type of an array of elements of type T. Here are two examples:

1. **int**[] An array of elements of type **int**.
2. String[] An array of elements of class-type String

Below is a declaration of an int-array b. Declare an array of any type in a similar fashion.

**int**[] b;

This declaration doesn’t create an array; it simply indicates the need for variable b. In Java, an array is actually an object, so a variable of type **int**[] contains a pointer to the array object. Thus, the above declaration results in a variable b that contains **null** (unless it is a local variable, which is not initialized).

The following assignment actually creates an **int** array of 3 elements and stores (a pointer to) it in b, producing the array and variable b shown to the right:

int[]@2

0

0

0

int[]@2

0

1

2

b

b= **new int**[3];

The array elements are assigned default values for their type, in this case, 0. For a String array created using **new** String[3], each element would contain **null**.

b.length is the number of elements in array b. In this case, b.length is 3. Note that length is a variable, not a function; b.length() is syntactically incorrect.

As in most programming languages, once created, the length of the array cannot be changed, But, of course, one could assign another array to b, for example, using b= **new** **int**[60];

**Referencing array** **elements**

int[]@2

3

5

0

int[]@2

0

1

2

b

The index of the first element of any array is 0. With b containing the value int[]@2, as shown above, the elements are b[0], b[1], and b[2]. To the right, we show how array b is changed by execution of these statements:

b[1]= 5;  
 b[0]= b[1] – 2;

The language spec indicates that b’s array elements are in contiguous memory locations and that it takes the same constant time to reference any array element. Example: retrieving the value b[0] takes essentially the same amount of time as retrieving the value b[2].

**Array elements occupy adjacent memory locations and take constant time to access**

Consider an **int** array b. *Its elements are guaranteed to occupy adjacent memory locations*. We explain. An **int** value takes 4 bytes. Suppose b[0] is at location 100. Then b[1] is at location 104, b[2] is at location 108, and in general b[i] is at location 100 + 4\*i. This means that b[i] can be accessed with this formula 100 + 4\*i.

Such a formula can be used to get to b[i] no matter what the type of i. It takes the same amount of time to access b[0] as it does to access b[1000]. We say it takes *constant time*.

**Array initializers**

We can write a sequence of statements as sown below to create an array and initialize its elements:

**int**[] c= **new int**[5];

c[0]= 5; c[1]= 4; c[2]= 0; c[3]= 6; c[4]= 1;

That’s awkward. Instead, use an *array initializer* and write the declaration like this:

**int**[] c= **new int**[] {5, 4, 0, 6, 1};

The array initializer is a list of expressions separated by commas and delimited by braces {}. Note that no expression appears between the brackets []. The size of the array is the number of elements in the array initializer.

Here’s another example: create a static array whose values are abbreviations of the days of the week:

**static** String[] weekDays= **new** String[] {“Mon”, “Tue”, “Wed”, “Thu”, “Fri”, “Sat”, Sun”};

**Multidimensional arrays**

One can create a rectangular 5-by-6 array d like this:

String[][] d= **new** String[5][6]

This rectangular array is viewed as having 5 rows and 6 columns. The number of rows is given by d.length, but the number of columns is given in a strange way:

d[0].length number of elements in row 0

d[1].length number of elements in row 1

…

The reason for this rather strange (at first) way of accessing the size of a row will become clear in the next section.

One can have 3-dimensional, 4-dimensional, etc. arrays in a similar fashion.

**Java implementation of multidimensional arrays**

Below is a declaration of a 2-dimensional array with an array initializer to give its elements. The 2-by-3 array is depicted to the right. This shows you how multi-dimensional array initializers can be used.

3 5 9

4 7 6

**int**[][] e= **new int**[][] {{3, 5, 9}, {4, 7, 6}};

The implementation of this array in many languages, including old ones like Fortran, Algol 60, and C, would put the values in row-major order in contiguous memory locations —that is, first row 0, then row 1, etc., as in the diagram to the right.

3 5 9 4 7 6

But Java does not. Instead, this Java views this two-dimensional array **int**[][] as a 1-dimensional array whose elements are 1-dimensional arrays. Array e looks like this:

int[][]@28

int[][]@28

0

1

e

int[][]

int[]@64

0

1

2

int[]

3

5

9

int[]@80

0

1

2

int[]

4

7

6

Thus, object e, whose type is **int**[][], contains a “row” of two pointers to objects of type **int**[], each of which contains the elements of that row.

This explains the weird notation e[i].length for the number of elements in row i. e[i] is a 1-dimensional array, and e[i].length is the number of elements in it.

You should continue to think of rectangular arrays as just that: a rectangular array. But know that its implementation is different. Further, know that this implementation allows us to have 2-dimensional arrays whose rows have different lengths, as we show the document *Ragged/jagged arrays*.