Ragged/Jagged Arrays

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

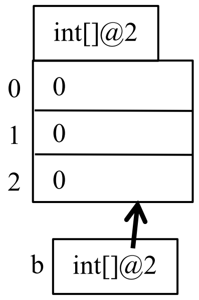
Below is a declaration of an int-array b. Declare an array of any type in a similar fashion.

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:

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**.

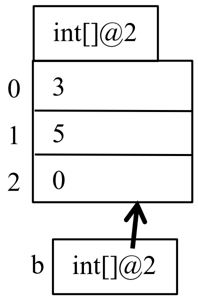
b.length is the number of elements in array b. In this case, b.length is 3. Note that length is a variable, not a function; b.length() is syntactically incorrect.

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

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 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"};

Ragged/Jagged Arrays

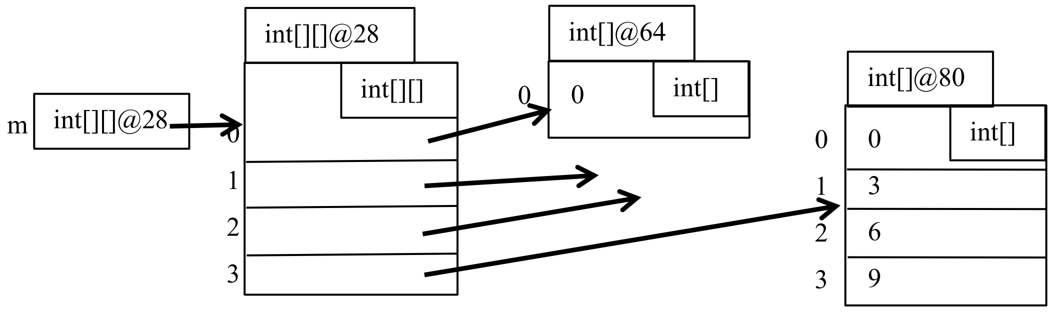
A multi-dimensional array whose rows are different lengths is called a *ragged array* or a *jagged array*, depending on whom you talk to.

Suppose we want a multiplication table m, with m[i][j] = i*j. Since i*j = j*i, we don't need a rectangular table for this, we only need half of it, as shown to the right.

```
0
0  1
0  2  4
0  3  6  9
```

The statement below uses an array initializer in which the first row has 1 value, the second 2, the third 3, and the fourth 4. Execution of this statement stores in m the conceptual triangular array shown above, but it is *implemented* as shown below the declaration. The important point is that array m of type int[][] is implemented as a 1-dimensional array whose elements are of objects of type int[]! In the interests of simplicity and clarity, we have drawn only objects m[0] and m[3] and not m[1] and m[2].

```
public static int[][] m= new int[][]{{0}, {0, 1}, {0, 2, 4}, {0, 3, 6, 9}};
```



Thus, you see how an array initializer can be used to create a ragged array. But suppose we want to create a much larger multiplication table, say, with 100 rows. Using an array initializer is out of the question. Instead, we write procedure multTable shown below. We discuss its important features.

First, note the new-expression new int[n][], given by arrow (1) below. The second pair of brackets has no expression inside it. This statement creates an object in which each array element is null and stores it in m. So each m[i] contains null.

As stated by the statement-comment (arrow (2)), each iteration i of the outer loop creates and populates row m[i]. It does this by creating a 1-dimensional array with i+1 elements and storing it in m[i] (arrow (3)) and then populating its values (the for-loop given by arrow (4)).

The two important points are: (1) the new-expression new int[n][], which creates an array with n null elements, and (2) the statement m[i]= new int[i+1]; which stores an int-array in element m[i].

```
/** Return a triangular array m (say) with n rows in which
  * each m[i][j], for 0 <= i <= j < n, contains i*j */
  public static int[][] multTable(int n) {
     int[][] m= new int[n][];  ←                      (1)

     for (int i= 0; i < n; i= i+1) {
        // Create and populate row m[i]  ←            (2)
        m[i]= new int[i+1];  ←                        (3)
        for (int j= 0; j <= i; j= j+1) {  ←           (4)
           m[i][j]= i*j;
        }
     }
     return m;
  }
```