Now that we know what an array is, the question arises, "How do we introduce one into a Java program?" In part, this must involve declaring a variable to have a suitable data type. To this end, we are helped by the fact that Java only allows us to work with arrays all of whose elements store values of the same data type. So every Java array is an array of ints, or an array of doubles, or an array of Strings, and so on. Here, for example, is how to declare the variable myArray to have the data type "array of ints". That is, it declares myArray in such a way that it references an array each of whose elements will store an int:

    int[] myArray;

It is the pair of brackets, [], with nothing between them, that reveals that an array is being declared. The fact that the brackets in this data type declaration appear without a gap immediately after an int data type reference reveals that myArray is being declared as an array of ints. In discursive text, it is common to say that myArray is being declared to have data type int[].

It is also permitted in Java to place the pair of brackets immediately after the variable being declared rather than immediately following the data type reference, like this:

int a[];

We do not use this alternative declaration method in this course, but you should be familiar with both methods.

The first method proceeds in two stages: first, announce the length of the array; then assign values to the array elements. The following code illustrates how to assign to myArray an array of length 10:

    myArray = new int[ 10 ];

In this statement, the keyword new is required, as are the brackets [ ... ] enclosing the integer that specifies the length. (We have more to say about the keyword new a little later in this course.) Following execution of this statement, we say that the array myArray has been *initialized* — in this case it has been initialized to an array of ints of length 10. It is important to note that, once an array has been initialized like this to have a particular length, there is no way ever to change that length.

We are now at a stage in which the array has been initialized but its elements have not. There is not much we can do with the array — apart, that is, from querying its length — until the elements are initialized. Element initialization requires that we be able to refer to the array's elements one by one. To do this, we use indices. For example, myArray[ 0 ] refers to the first element of the array, myArray[ 1 ] to the second, and so on. In the following code, values are assigned to the first three elements, and to the fifth:

    myArray[ 0 ] = 5;   
    myArray[ 1 ] = 0;   
    myArray[ 2 ] = 5400;   
    myArray[ 4 ] = -96725;

The values assigned to array elements can, of course, be calculated rather than specified outright. Here, for example, the value of myArray's last element is assigned the sum of its first two elements:

    myArray[ 9 ] = myArray[ 0 ] + myArray[ 1 ];

Before such an assignment will work, the first and second elements of myArray must already have been initialized.

As was the case for previous data types, the declaration and assignment phases of array initialization may be combined into one. The following statement, for example, declares the variable d as an array of doubles and assigns to it an array of 100 doubles:

    double[] d = new double[ 100 ];

We reiterate that this statement only initializes the array d. The *elements* of d have not yet been initialized.

[Recall](http://pages.eimacs.com/eimacs/mainpage?epid=E2316502760&cid=162149#StrLength) that, if s is a string, then the expression s.length() evaluates to the length of the string. In that context, length names a method. When dealing with an array a, on the other hand, the expression we evaluate to determine a's length has a different form; in particular, it does not have a pair of parentheses at the end. In the context of arrays, length is referred to as a *property*.

To reference an element of an array, we use the element's index. For any given array, the index of each of its elements is an integer in the range from 0 through one less than the length of the array. The desired index does not have to be specified directly; it can be the value of any expression, provided of course that the value falls into the required range. Use the following code fragment to find out what happens when you enter the expression 5 - 3 into the white edit box.

    String[] names;   
  
    names = new String[ 10 ];   
  
    names[ 0 ] = "albert";   
    names[ 1 ] = "bertie";   
    names[ 2 ] = "cuthbert";   
  
    int i = ;   
         
    System.out.println( names[ i ] );

cuthbert

Note that attempting to reference an element of an array using a negative index or an index that is greater than or equal to the length of the array causes program execution to be interrupted, an ArrayIndexOutOfBoundsException exception to be thrown, and a corresponding error message to be posted.

On the other hand, referencing an uninitialized element of an array does not provoke an error. As you saw in Exercise 44(d) and (e), when an element of an array of Strings has not been assigned a value, Java reports the value of such an element as null. This is not a String. It is the value of the literal null and it has a special data type that it shares with nothing else. (Furthermore, to prevent anything ever being declared to have this special data type, the data type has no name.) Java often reports null as a value when no value exists.

Java deals with uninitialized elements of arrays of other data types in a similar fashion. So, in a sense, it is possible to access the values of array elements even if those elements have not been initialized. At worst, Java will respond with a reference to null or some suitable "representative" value such as false, 0, or 0.0. It is, however, considered very bad practice to write code that relies on this behavior. It is far better to assign explicit values to array elements prior to trying to access their values

Java deals with uninitialized elements of arrays of other data types in a similar fashion. So, in a sense, it is possible to access the values of array elements even if those elements have not been initialized. At worst, Java will respond with a reference to null or some suitable "representative" value such as false, 0, or 0.0. It is, however, considered very bad practice to write code that relies on this behavior. It is far better to assign explicit values to array elements prior to trying to access their values.

Our second method of producing initialized arrays with initialized elements reduces the above code to a single line:

    int[] a = { 5, 15, 20 };

The form of this abbreviated array-initialization statement is as follows:

* an array data type declaration (in the above example, int[] a), followed by
* the assignment operator, followed by
* a listing of all the values that are to be stored in successive array elements, separated by commas, and surrounded by a pair of braces — { ... }.

Actually, it is not required that each of the listed initial values be an explicit value. An expression can always be used in place of an explicit value, provided that it is possible to evaluate the expression at the moment when the array is declared. Obviously, when this type of initialization is used, there is no need for us to specify the length of the array because all the values we want to be in the array are available to be counted.

For example, after this code is executed:

    boolean[] b = { true, false, true, true, false };

b is an array of booleans of length 5.

Here is an example in which an expression appears in place of an explicit value:

    int x = 5;   
    int y = 12;   
    int[] z = { 6, 8, (x \* 2), y };

This code assigns to z an array of length 4 with elements containing the integers 6, 8, 10 and 12.

There are strict rules about where and how you may use braces notation when working with arrays. If you want to use braces when initializing a previously-declared array, then you must use the new keyword. On the other hand, if you use braces in a combined declaration and initialization statement, then using the new keyword is optional.

    String[] b;   
    b = { "a", "b", "c" }; // illegal   
    b = new String[] { "a", "b", "c" }; // legal   
  
    int[] z = new int[] { 1, 2, 3, 4, 5 }; // legal   
    int[] z = { 1, 2, 3, 4, 5 }; // legal

When declaring a two-dimensional array, we must include *two* pairs of brackets, one for each index. The following statement, for example, declares a two-dimensional array a of doubles:

    double[][] a;

We can create such an array with 5 rows and 10 columns and assign it to a as follows:

    a = new double[ 5 ][ 10 ];

Alternatively, we can combine the declaration and assignment into a single line of code like this:

    double[][] a = new double[ 5 ][ 10 ];

As in the case of one-dimensional arrays, there is a shorthand method for declaring and initializing a two-dimensional array while at the same time initializing all its elements. This is as simple as specifying the contents of each row, as in this example:

    int[][] t = { { 1, 2, 3, 4 },    
                  { 5, 6, 7, 8 },    
                  { 9, 10, 11, 12 } };

t[ 0 ][ 0 ] = 1

t[ 0 ][ 1 ] = 2

t[ 1 ][ 0 ] = 5

If a is a two-dimensional array, the expression a.length evaluates to the number of rows the array has, and the expression a[ 0 ].length evaluates to the number of columns.

The alternative method of placing the declarative brackets immediately after the variable rather than following the data type reference is also acceptable for two-dimensional arrays, as in these examples:

double a[][];  
double m[][] = new double[ 5 ][ 10 ];  
int t[][] = { { 1, 2, 3, 4 },   
              { 5, 6, 7, 8 },   
              { 9, 10, 11, 12 } };

  int[] x = { 1, 2, 3, 4, 5 };   
    int[] y;   
  
    y = x;

Experiment with the code fragment above, with a view to answering these questions:

1. If a new value is assigned to, say, x[ 3 ], does the value of y[ 3 ] also change?
2. If a new value is assigned to, say, y[ 3 ], does the value of x[ 3 ] also change?
3. Yes. Any change in an element of x causes an identical change in the corresponding element of y. The reason is that x and y reference the same array.
4. Yes. Any change in an element of y causes an identical change in the corresponding element of x. The reason is that x and y reference the same array.