

LESSON 03 – 2D ARRAYS



In this lesson we will explore 2 dimensional (2D) arrays used to store tables of data. We will also examine how to traverse a 2D-Array.

Sections:

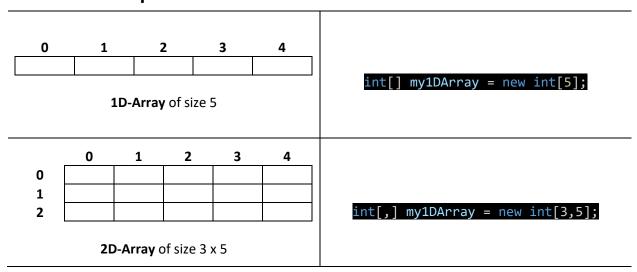
l.	1D-ARRAY VS 2D-ARRAY	PAGE 1
II.	2D-ARRAYS WITH DATA IN C#	PAGE 3
III.	TRAVERSING A 2D-ARRAY	PAGE 4

I. 1D-ARRAY VS 2D-ARRAY:

A **2D-Array** is a **multidimensional array** that has multiple rows of data. They are very useful for representing tables of data (or grids). For example: a maze in a game, pixels in an image, a chess board, etc. Here is a conceptual view of what both a 1D-Array and 2D-Array look like, along with how you can declare them in C#:

Conceptual View:

C# Declaration:



010

10011 0101

As you can see the syntax for a 2D-Array is a little different. A 2D-Array in C# has a comma in the declaration. On the right side of the qual sign, you can see another comma separating two integers. The first integer represents the number of rows, and the second integer the number of columns. For this example, we would say that this is a 3 by 5 array (i.e., 3 rows and 5 columns), or it may be referred to as a 3 x 5 array.

Like 1D-Arrays, 2D-Arrays can be created as doubles and strings as well, for example:

double[,] myDoubleArray = new double[3,5]; string[,] myStringArray = new string[3,5];

00₁₇₀₀10₁₀₁₀₁₀₁₀₁₀000 10₀₇₇ 01010101010 01010

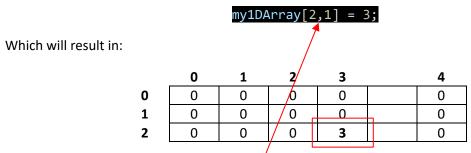
70₇₀₇₇ 01₇₀₁₀₁₀₁₀₁₀ 010 07₅₀₇₀ 01₇₀₇₀₁₀ 111001010101010 07₅₀₇₀ 01₇₀₇₀₁₀ 2D ARRAYS WITH DATA IN C#:

We saw in the last section how to create a **blank** 2D Array:

This will create a 3x5 array and the default values will be '0' since we are creating a 2D integer array:

	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0	0
2	0	0	0	0	0

We can write to a particular spot in the array similar to how we do with a 1D-Array, for example:



And of course, we can access an element in a similar manner, for example:

Note: As you can see, we are using commas to separate our row number and column number.

Output will be:

3

If we wish to create a 2D-Array with predetermined data (hardcoded data), we can use braces {} like we did with 1D-Arrays. With 2D-Arrays we will have multiple sets of braces { } where each set of braces represents a row, for example:

int[,] my2DArray = {-{ 3, 4, 22, 33, 2 }, { 99, 88, 1, 17, 3 }, { 3, 4, 22, 4, 22 }};

Notice that we have a set of braces surrounding all our data, and then each row of data has it's own set of braces { }. Also notice the use of commas to separate each value in each row as well as the use of commas to separate each row. The above statement will create an array that looks like the following:

	0	1	2	3	4
0	3	4	22	33	2
1	99	88	1	17	3
2	3	4	22	4	22

10070101010101010 0\\\ 107077 070101010 \\ 107077 0701010 TRAVERSING A 2D-ARRAY: 1070701070010TRAV

When we traversed a 1D-Array we used a for-loop. Similarly, with a 2D-Array we can use a nested for-loop in which the outer loop traverses the rows, and the inner loop traverses the columns. Let's look at an example...

```
int[,] my2DArray = { { 3, 4, 22, 33, 2 }, { 99, 88, 1, 17, 3 }, { 3, 4, 22, 4, 22 } };
//print grid to screen
for (int x = 0; x < 3; x++)
    for (int y = 0; y < 5; y++)
        Console.Write(my2DArray[x,y] + " ");
    Console.WriteLine();
```

Output will be:

```
3 4 22 33 2
99 88 1 17 3
3 4 22 4 22
```

Notice that we have hardcoded our values in 'my2DArray' (Recall: hardcode means that we typed in all the values manually). Notice the number of curly braces needed when hardcoding the values for a 2D-Array (i.e., one set always surrounding the entire array, and one set for each of the rows). Also notice how we use our variable 'x' from our outer for-loop for the rows, and our variable 'y' from the inner for-loop for the columns.

Recall: As indicated in Lesson 01, we use double quotes "" when adding a space in our Console.Write() statement or we will get strange outputs.