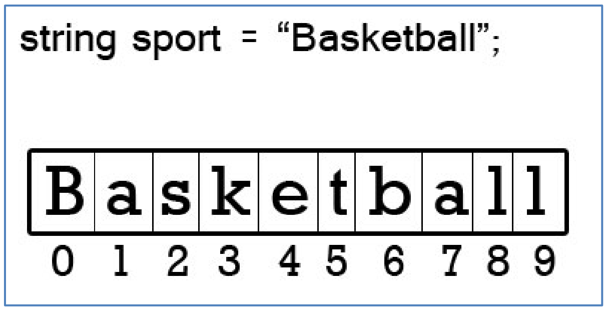
LESSON 01 – 1-DIMENSIONAL (1D) ARRAYS

In this lesson we explore 1 dimensional (1D) arrays that can be used to store data in memory. We will also look at how to traverse arrays to retrieve that data.

Sections:

I. WHAT IS AN ARRAY?

An **array** is a data structure that contains a group of elements. So far, we used variables to store single elements of a specific data type (i.e., *int*, *float, char,* etc.) in memory. An array, however, allows us to hold a group of elements of a specific type in memory. In fact, we have already been using an array data structure when we used the *string* data type. A *string* is a group of *char* elements (an array of *chars*):



```
string sport = “Basketball”;
```

The above *string* contains 10 *chars* starting from 0 to 9. Recall that if we wanted to access a specific letter in the above string (or array of *chars*) we could do the following:

```
Console.WriteLine(sport[3]);
```

The above line of code would output the character ‘k’ to the screen. Recall that the number inside the square brackets **[ ]** is called the **index**. In this case, the index ‘3’ will access the letter ‘k’.

Recall: The **index** always **starts** at **‘0’.**

Like an array of *chars*, we can make an array of any data type, for example:

```
int[] myArray = { 1, 5, 9, 11, 4 };
```

The above line of code will create an array of *ints*. A couple things to note here:

- We use the square brackets **[ ]** beside our variable name our **data type keyword** (in this case **int**) to indicate that we are creating an array (in this case an array of integers).
- We use the curly braces **{ }** to define what our group of elements will be. Each element within the curly braces { } is separated by a **comma.** Therefore, to output the 4th element '11' in this array we can do the following:

```
Console.WriteLine(myArray[3]);
```

Similarly, we can create an array of *doubles, chars* and even *strings:*

```
double[] myDoubleArray = { 34.4, 56.6, 99.999 };

char[] myCharArray = { 'a', 'b', '\n', 'z' };   // notice that we can store
                                                // the newline character '\n'
string[] myStringArray = { "Hello", "World", "C# Rocks!!!" };
```

And just like before we can output elements of each array, for example:

```
Console.WriteLine(myDoubleArray[2]);
```

Output will be: 99.999

```
Console.WriteLine(myCharArray[2]);
```

Output will be a new line.

```
Console.WriteLine(myStringArray[0]);
```

Output will be: Hello

II. PROPER USAGE OF ARRAYS IN C#:

In the last section, we saw how to create simple arrays in C# with a predetermined set of elements (a **hardcoded** set of values). Sometimes, however, it is necessary to create an array without out any initial values, for example:

```
double[] myDoubleArray = new double[3];
myDoubleArray[0] = 0.5;
Console.WriteLine(myDoubletArray[0]);
```

Sample **Output:**

```
0.5
```

The above line of code will create a *double* array named 'myDoubleArray' and will allocate 3 spots for it in memory. We then assign the value '0.5' to the first spot in the array. Finally, we output the value at index '0' (our first spot).

Notice how we created the array:

```
double[] myDoubleArray = new double[3];
```

Notice how we make this array equal to: **`new double[3];`**

Let's break this down:

- We are using the keyword **'new'**
- We have the **data type keyword** appear again with brackets and a number inside. In our case, we have the 'double' keyword appear again with the number **'3'** in brackets **[ ].** This will reserve 3 spots in memory for our double array.
- The number in the brackets represents the **size of the array** and must always be **positive integer.**

The above code demonstrates the standard way of creating arrays when the element list is unknown. When you create an array, you must indicate the **size of the array.** We did so above with the number in our brackets [ ]. We also did in the previous section by supplying a fixed number of known elements in our braces { }. C# inferred the size of our array because it counted the number of elements in our braces at runtime.

To expand a little further, let's play with our array a little bit:

```
double[] myDoubleArray = new double[3];
myDoubleArray [0] = 0.5;
Console.WriteLine(myDoubleArray [2]);
```

Sample **Output:**

```
0
```

Since there was no assigned value at index ‘2’ in our above code, a default of ‘0’ is stored at that position.

```
double[] myDoubleArray = new double[3];
myDoubleArray[0] = 0.5;
Console.WriteLine(myDoubleArray[3]);
```

The above code will throw an ‘Index out of range’ **exception** at runtime since index ‘3’ is out of bounds (i.e., the index does not exist in our array. More specifically, since our start index is always ‘0’, our highest possible index in this case is ‘2’).

```
double[] myDoubleArray  = new double[3];
myDoubleArray [0] = 0.5;
Console.WriteLine(myDoubleArray [-1]);
```

The above code will also throw an ‘Index out of range’ **exception** at runtime since index ‘-1’ is out of bounds.

```
double[] myDoubleArray  = new double[3];
myDoubleArray [0] = 0.5;
Console.WriteLine(myDoubleArray);
```

Sample **Output:**

```
System.Double[]
```

The above output might appear strange. If you notice, we did not indicate an index number when outputting our array. We simply outputted the array variable ‘myDoubleArray’. When Console.WriteLine() encounters a situation like this, it will simply output the **data type** of our array which in this case is System.Double[].

III. TRAVERSING AN ARRAY:

When using arrays, we must have some way of going through each element from start to end. This is called **traversing** an array. For example, let's say we wanted to output each element in an array to the screen. We could do something like the following:

```
int[] myArray = { 4, 8, 21, 66 };
Console.WriteLine(myArray[0]+ " " + myArray[1] + " " + myArray[2] + " " + myArray[3]);
```

Sample **Output:**

```
4 8 21 66
```

But imagine we had hundreds, thousands or even millions of elements in our array. This approach would not be practical. A common way to traverse an array is to use a **for-loop** and use the **counter variable** within the for-loop to access each element in that array. This is best demonstrated with an example:

```
int[] myArray = { 4, 8, 21, 66 };

for (int x = 0; x < myArray.Length; x++)
{
    Console.Write(myArray[x] + " ");
}
```

Sample **Output:**

```
4 8 21 66
```

This loop will iterate 4 times from **0** to **myArray.Length.** Just like when we traversed the characters of a string, we can use the **.Length** property to retrieve the number of items in an array. **Note:** We use the term 'property' here and not 'built-in function'. If there are no brackets () at the end of our built-in feature, then we usually refer to such built-in features as a 'propertiey'. For example, .Length is a **property** whereas .Trim() is a **function.**

Inside our for-loop, notice that we have 'x' as our index for our array myArray[] during output. The result is that on every loop iteration it will first output myArray[0] then myArray[1] then myArray[2] and so on. Now we can traverse any array of any size using a simple for-loop.

Note: In our **Console.Write()** statement we add myArray[x] to a space in **double quotes " "** not single quotes ' '. Even though a space is a single character, if we added the space using single quotes, our Console.Write() statement would think that we are adding char values and our output will not be what you might expect (try it out if you wish). We won't get into the details of why this is, but for simplicity, just use double quotes " " in this situation.