C# Arrays

Topics Covered

- Array Qualities - Overview
- Array Structure in Memory
- Initializing Arrays
- Runtime Sizing
- Filling an Array
- Array Properties
- Largest Value in an Array
- Indexing Outside of an Array
- Copying Arrays
- Testing for Array Equality
- Arrays and foreach Loops
- The Array Class

Array Overview

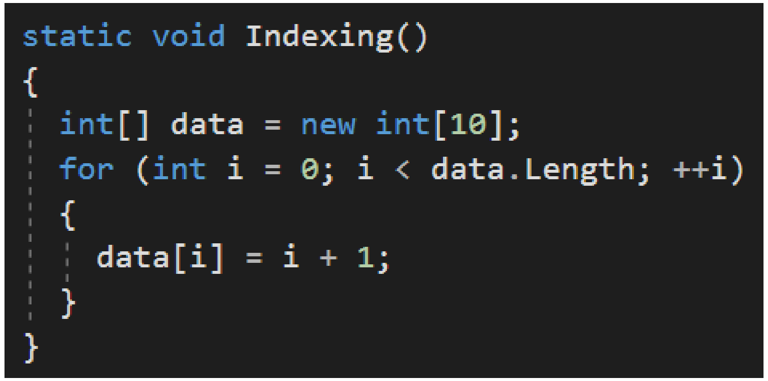
- An array can be Single-Dimensional, Multidimensional or Jagged.
- The number of dimensions and the length of each dimension are established when the array instance is created. These values can't be changed during the lifetime of the instance.
- The default values of numeric array elements are set to zero, and reference elements are set to null.
- A jagged array is an array of arrays, and therefore its elements are reference types and are initialized to null.
- Arrays are zero indexed: an array with n elements is indexed from 0 to n-1.
- Array elements can be of any type, including an array type.
- Array types are reference types derived from the abstract base type Array. Since this type implements IEnumerable and IEnumerable<T>, you can use foreach iteration on all arrays in C#.

Structure of Arrays

As in C / C++, C# elements of an array are layed out contiguously in memory. This means that if we instantiate an array of 10 ints, a block of 40 bytes of memory will be allocated to hold the array's values. Similarly, an array of 10 doubles requires 80 bytes of memory.

In C# we don't normally access memory directly (though it can be done!), but arrays provide an indexing operator to access the individual elements. It allows us to both read and write to a given array position.

Performance for the indexing operations is O(1).



```
static void Indexing()
{
    int[] data = new int[10];
    for (int i = 0; i < data.Length; ++i)
    {
        data[i] = i + 1;
    }
}
```

Initializing Arrays

Arrays are reference types, and we instantiate them using the new operator and square brackets to signify the array:

```
int[] values = new int[10];            // an array of int
object[] ovalues = new object[255];   // an array of object
```

For the arrays of value types, the elements are initialized to 0. For arrays of reference types, the elements are initialized to null.

We can initialize arrays by providing a list of values after the new operator. This statement creates an array with 10 elements initialized with the values 1 – 10:

```
int[] values = new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
```

platform By Per Scholas

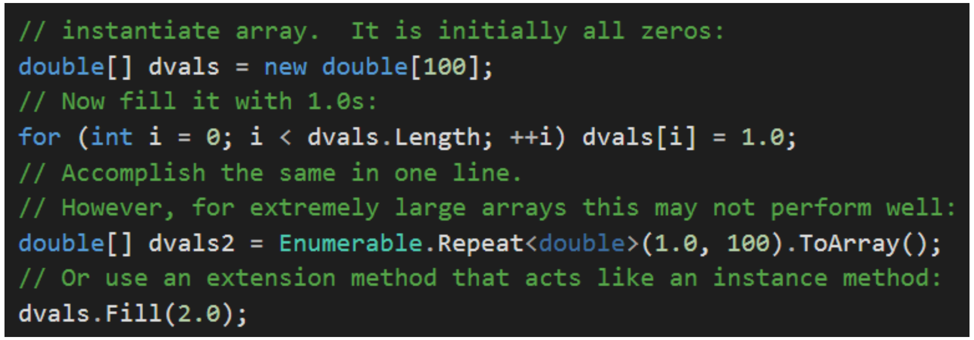
Run-time Sizing

The length of an array does not need to be known at compile-time. We might, for example, fill an array with values read from a database. Assume in the example that the function ExecScalar returns the scalar value from a SQL statement:

```
int nRecords = ExecScalar(“select count(*) from forecasts”);
Forecast[] data = new Forecast[nRecords];
// data is now has space to hold each of the forecast records.
```

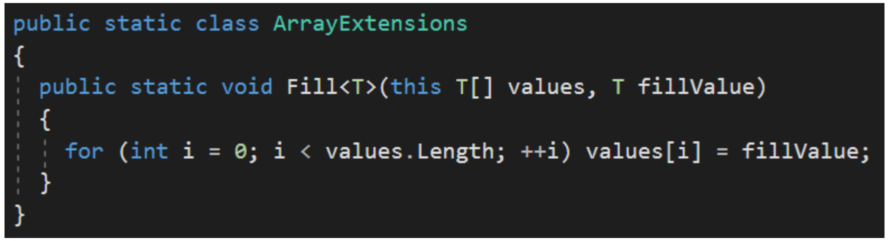
Filling an Array

There is no built-in “Fill” method for an array. However, it is easily done:



```
// instantiate array.  It is initially all zeros:
double[] dvals = new double[100];
// Now fill it with 1.0s:
for (int i = 0; i < dvals.Length; ++i) dvals[i] = 1.0;
// Accomplish the same in one line.
// However, for extremely large arrays this may not perform well:
double[] dvals2 = Enumerable.Repeat<double>(1.0, 100).ToArray();
// Or use an extension method that acts like an instance method:
dvals.Fill(2.0);
```

The third technique uses a feature of the C# language called extension methods where one can define static methods that act like instance methods. Once defined, they are completely re-usable:



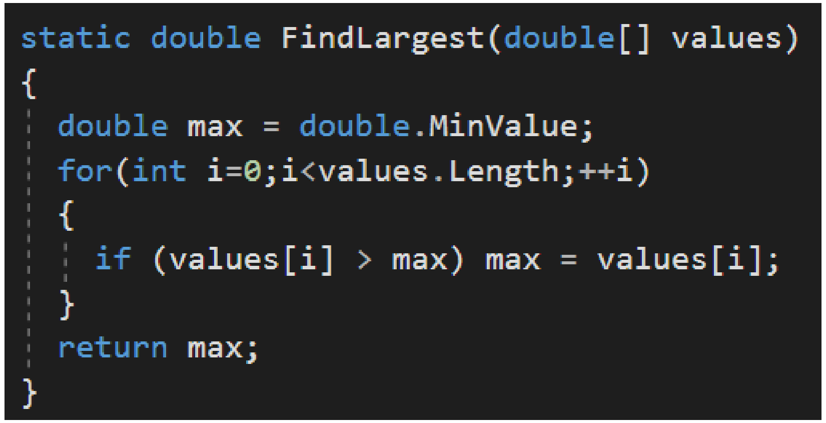
```
public static class ArrayExtensions
{
  public static void Fill<T>(this T[] values, T fillValue)
  {
    for (int i = 0; i < values.Length; ++i) values[i] = fillValue;
  }
}
```

Array Properties

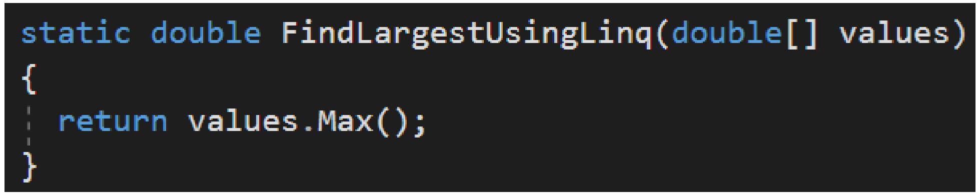
Property Name	Return Type	Description
Length	int	Gets the total number of elements in all the dimensions of the Array.
LongLength	long	Same as Length, but can be used for extremely large arrays where the # elements would overflow int.
Rank	int	Gets the rank (number of dimensions) of the Array. For example, a one-dimensional array returns 1, a two-dimensional array returns 2, and so on.

Largest Value in an Array

Here are two ways to find the largest value in an array:



```
static double FindLargest(double[] values)
{
    double max = double.MinValue;
    for(int i=0;i<values.Length;++i)
    {
        if (values[i] > max) max = values[i];
    }
    return max;
}
```



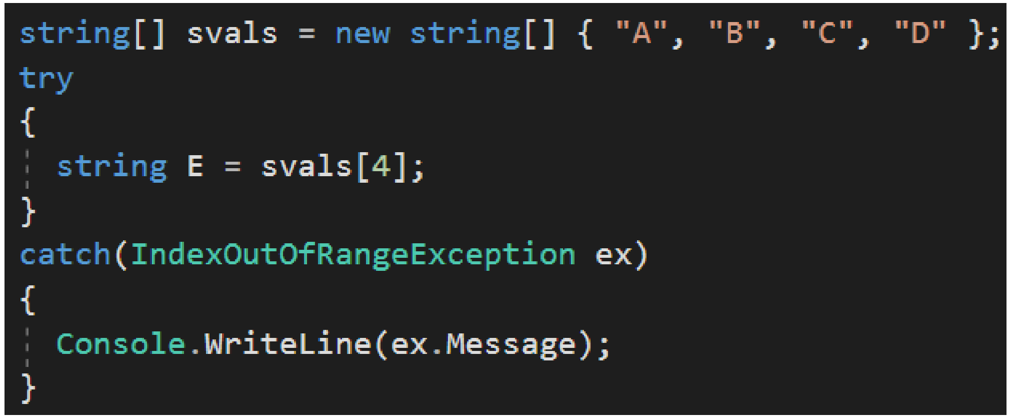
```
static double FindLargestUsingLinq(double[] values)
{
    return values.Max();
}
```

On the left is a straight-forward for loop that tests each value in the array.

The approach on the right uses another extension method defined in the System.Linq namespace. Again, for extremely large arrays, you should test the relative performance of each of these approaches.

Indexing Outside of an Array

In C#, if you attempt to access an element beyond the bounds of an array, an IndexOutOfRangeException occurs:



```
string[] svals = new string[] { "A", "B", "C", "D" };
try
{
    string E = svals[4];
}
catch(IndexOutOfRangeException ex)
{
    Console.WriteLine(ex.Message);
}
```

The output of this code is: “Index was outside the bounds of the array.”

Compare this behavior to what happens with C or C++.

Copying Arrays

Here is a naive attempt to copy an array:

```
int[] ivals = new int[] { 5, 10, 15, 20, 25, 30, 35 };
int[] ivalsCopy = ivals;  // That was easy .... NOT!
```

There is no copy created here, no new memory allocated to store the copy. The variable *ivalsCopy* merely points to the very same array as *ivals*.

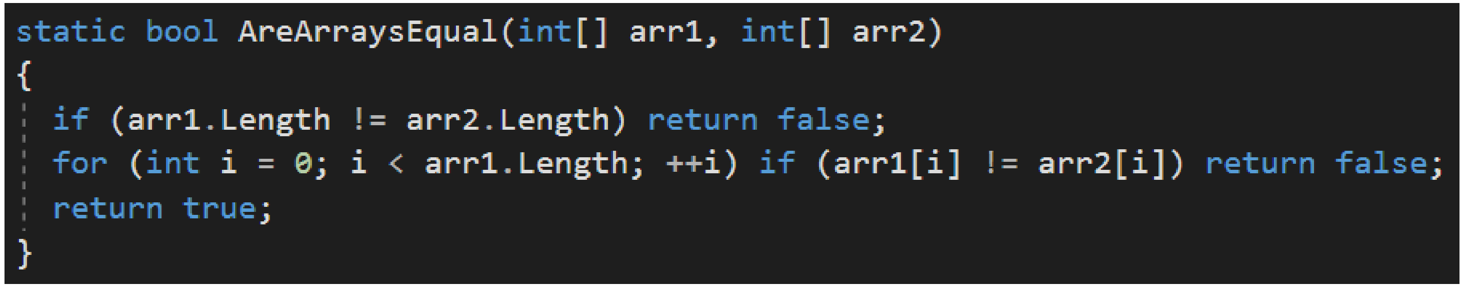
The Array class provides a Clone method for creating actual copies:

```
int[] ivals = new int[] { 5, 10, 15, 20, 25, 30, 35 };
int[] ivalsCopy = (int[])ivals.Clone();
```

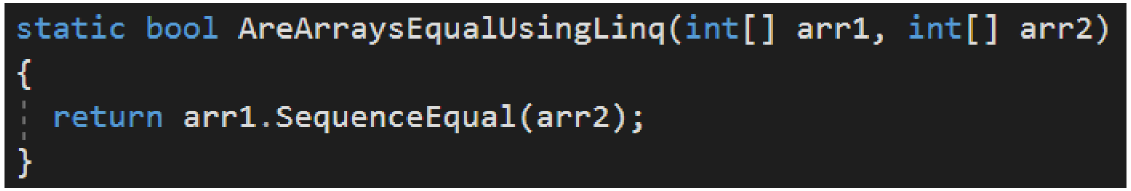
Note that we must use an explicit cast. This is because Clone() returns a value of type Object. The copy is a *shallow copy*. The distinction is irrelevant for value types, but for reference types it means that the new array contains references to the very same objects as in the original array.

Testing for Array Equality

Here are two different methods for comparing equality of two arrays. The first uses a simple for loop. The second uses an extension method from System.Linq:



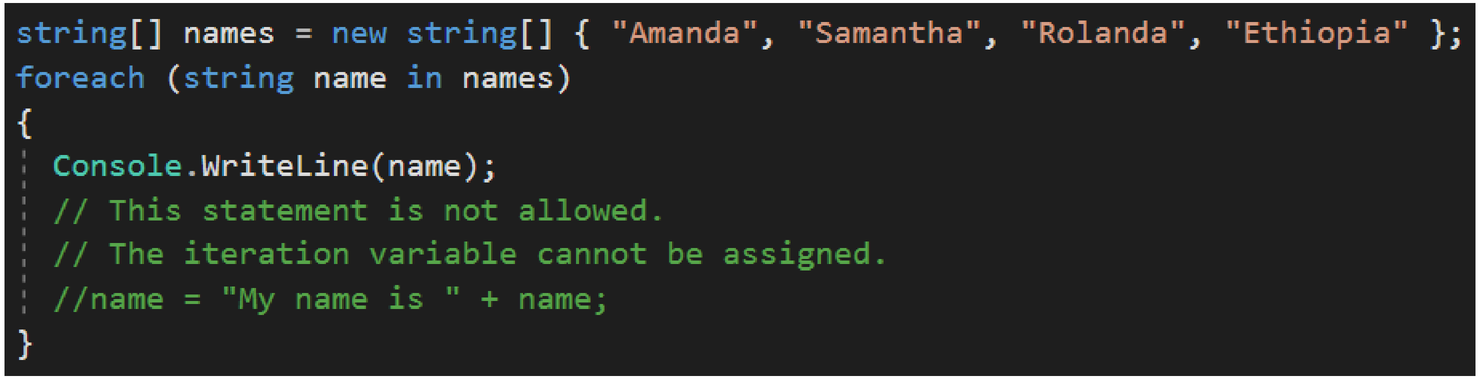
```
static bool AreArraysEqual(int[] arr1, int[] arr2)
{
    if (arr1.Length != arr2.Length) return false;
    for (int i = 0; i < arr1.Length; ++i) if (arr1[i] != arr2[i]) return false;
    return true;
}
```



```
static bool AreArraysEqualUsingLinq(int[] arr1, int[] arr2)
{
    return arr1.SequenceEqual(arr2);
}
```

Arrays and foreach Loops

The Array class implements the IEnumerable interface, allowing arrays to be used in foreach loops:



```
string[] names = new string[] { "Amanda", "Samantha", "Rolanda", "Ethiopia" };
foreach (string name in names)
{
    Console.WriteLine(name);
    // This statement is not allowed.
    // The iteration variable cannot be assigned.
    //name = "My name is " + name;
}
```

As always, the iteration variable is read-only. If you need to loop through an array with the option of updating the element values, us a *for* loop with indexing.

The Array Class

The Array class is the base class for all arrays of any element type.

Array provides many useful **static** methods. A sample:

➢ BinarySearch:	Efficiently searches a pre-sorted array for a given value.
➢ Clear:	Sets a range of elements in the array to their default values.
➢ Fill:	Fills an array with a provided value.
➢ ForEach:	Executes an action on each element in the array.
➢ Resize:	Changes the length of an array.
➢ Reverse:	Reverses the order of elements in an array.
➢ Sort:	Sorts the elements in an array.

Topics Covered

- Array Qualities - Overview
- Array Structure in Memory
- Initializing Arrays
- Runtime Sizing
- Filling an Array
- Array Properties
- Largest Value in an Array
- Indexing Outside of an Array
- Copying Arrays
- Testing for Array Equality
- Arrays and foreach Loops
- The Array Class

Questions

- What is a single-dimensional array?
- What is a multi-dimensional array?
- What is a jagged array?
- How do we access the elements of an array?
- What are three properties of an array?
- What happens if we try to index an element beyond the length of an array?
- The array’s Clone method creates a shallow copy of an array. What is a shallow copy? What is a deep copy?
- What does it mean for two arrays to be equal?
- What feature of arrays allows them to be used in **foreach** loops?

Exercises

1. Declare and initialize an array with 5 integers: 1, 7, 13, 24, and 51.
2. Declare an array of 100 integers. Initialize it with the first 100 odd integers.
3. Declare an array of 1000 doubles and fill it with random values. Find the smallest value in the array.
4. Declare an array of 1000 ints and fill it with random values. Find the median value in the array.
5. Declare an array of 10 ints and fill it with consecutive values 1 – 10. Use Array.Resize to change the length to 20. Is it the same array object? What do you see the new values?
6. Declare an array of 1000 ints and fill it with consecutive values 1-1000. Shuffle it in a way that completely randomizes the values.