Arrays

An **array is a set of contiguous data points of the same type (**an array of ints, an array of strings, an array of SportsCars, and so on).

string[] booksOnDotNet = new string[100];

When declaring a C# array using this syntax, the number used in the array declaration represents the total number of items, not the upper bound. Also note that the lower bound of an array always begins at 0. Thus, when you write int[] myInts = new int[3], you end up with an array holding three elements, indexed at positions 0, 1, and 2.

Do be aware that if you declare an array but do not explicitly fill each index, each item will be set to the default value of the data type (e.g., an array of bools will be set to false or an array of ints will be set to 0).

In addition to filling an array element by element, you are able to fill the items of an array using C# **array initialization syntax.** To do so, specify each array item within the scope of curly brackets ({}). This syntax can be helpful when you are creating an array of a known size and want to quickly specify the initial values.

```
string[] stringArray = new string[]    { "one", "two", "three" };
// or
bool[] boolArray = { false, false, true };
// or
int[] intArray = new int[4] { 20, 22, 23, 0 };
```

If there is a mismatch between the declared size and the number of initializers (whether you have too many or too few initializers), you are issued a *compiletime error.*

Recall that the var keyword allows you to define a variable, whose underlying type is determined by the compiler. In a similar vein, *the var keyword can be used to define implicitly typed local arrays*. Using this technique, you can allocate a new array variable without specifying the type contained within the array itself (note *you must use the new keyword when using this approach*).

```
var a = new[] { 1, 10, 100, 1000 };
```

Defining an array of Objects

If you were to define an array of System.Object data types, the subitems could be anything at all.

```
object[] myObjects = new object[4];
myObjects[0] = 10;
myObjects[1] = false;
myObjects[2] = new DateTime(1969, 3, 24);
myObjects[3] = "Form & Void";
```

Working with Multidimensional Arrays

In addition to the single-dimension arrays you have seen thus far, C# also supports two varieties of multidimensional arrays. The first of these is termed a **rectangular array**, *which is simply an array of multiple dimensions, where each row is of the same length.*

```
int[,] myMatrix;
myMatrix = new int[3,4];
```

The second type of multidimensional array is termed a **jagged array.** As the name implies, *jagged arrays contain some number of inner arrays, each of which may have a different upper limit.*

```
// Here we have an array of 5 different arrays.
int[][] myJagArray = new int[5][];
```

The System.Array Base Class

Every array you create gathers much of its functionality from the System.Array class. Using these common members, you are able to operate on an array using a consistent object model.

- Clear() - This static method sets a range of elements in the array to empty values (0 for numbers, null for object references, false for Booleans).
- Copy() - This method is used to copy elements from the source array into the destination array.
- Length - This property returns the number of items within the array.
- Rank - This property returns the number of dimensions of the current array.
- Reverse() - This static method reverses the contents of a one-dimensional array.
- Sort() - This static method sorts a one-dimensional array of intrinsic types. If the elements in the array implement the IComparer interface, you can also sort your custom types.

```
int[] ar2 = new int[3] { 2, 2, 5};
int[] ar3 = { 5, 6 , 7};
// the 2-nd element of ar2 copies into the ar3's 2-nd element
Array.Copy(ar3, ar2, 2);

Console.WriteLine(ar2.Rank);  // 1

int[] ar = new int[] {5, 5, 5, 6 };
Array.Clear(ar, 1, 1);    // 5 0 5 6

Array.Reverse(ar3);  // 7 6 5

int[] ar4 = { 5, 3, 7};
Array.Sort(ar4);   // 3 5 7
```

Notice that many members of System.Array are defined as static members and are, therefore, called at the class level (for example, the Array.Sort() and Array.Reverse() methods). Methods such as these are passed in the array you want to process. Other members of System.Array (such as the Length property) are bound at the object level; thus, you are able to invoke the member directly on the array.