INTRODUCTION TO ARRAYS

Up to this point, our applications have used variables that only hold a single value at a time. This can prove extremely inconvenient and tedious if, let's say, we wanted to create a program that stored the song titles of all the songs we have stored on our iPods. If I have 500 songs on my iPod that I want stored in a program, this would mean that I would need 500 variables. In other words, our program would end up looking something like this:

```
Dim song1 As String
Dim song2 As String
...

song1 = "Karma Police.mp3"
song2 = "Society.mp3"
...
```

Rather than having to declare and initialize 500 variables, a special construct called an ARRAY can be used to store all the song titles together in one group.

WHAT IS AN ARRAY?

An ARRAY is a data structure that can store many of the same kind of data together at once. For example, an array can store 2500 String characters representing all your individual student numbers, or an array can store 2500 integer values representing each student's age. Each array can only store one type of data; it cannot, for example, store both integers and Strings. Arrays are an important and useful programming concept because it allows programmers to store a large amount of data without having to name and declare more than one variable.

An array has a fixed length and can only contain as many data items as its length allows. For example, the following is an example of an array that stores the names of five students in our class:

0	1	2	3	4
Marcus	Erlito	Dewayne	Malcolm	Jaimy

An array **element** is one of the data items in an array. For example, in the above array, Jaimy is an element. Each element has an **index** value, with 0 being the index of the first item, 1 being the index of the second item, etc. So, continuing with our example, Jaimy is the fifth element in the array and has an index value of 4.

DECLARING AN ARRAY

An array is declared with a **Dim** statement that includes the array name followed by the index of the last element in parentheses and then the data type of the elements. Let's say, for example, I wanted an array that will store the 500 song titles that are on my iPod. The declaration statement for the array would look something like this:

```
Dim mySongs(499) As String
```

The above statement declares an array called **mySongs** that has 500 elements with indexes 0 to 499. Note that the size of the array is indicated with index of the last element. The elements are automatically initialized to the default value of the element data type. So, in the above example, each element is initialized to **Nothing**.

You can also declare an array using two steps. The first step requires that you declare the array:

```
Dim mySongs() As String
```

The second step requires that you initialize the array by specifying the size of the array as follows:

```
mySongs = New String(499) {}
```

You can also initialize an array by explicitly specifying the array bounds as follows:

```
mySongs = New String(0 To 499) {}
```

ASSIGNING VALUES TO ELEMENTS IN AN ARRAY

Just as there is more than one way to declare and initialize an array, there is more than one way you can assign values for each element in an array. Elements in an array can be assigned values within the declaration statement itself, as follows:

```
Dim mysongs() As String = {"Karma Police.mp3", "Society.mp3",
      "Montana.mp3", "Today.mp3", "Jed the Humanoid.mp3"}
```

In the above example, the index is not included in the parentheses because the number of values in the braces indicate the size of the array. This means that the above array only has 5 indexes (0 – 4), where index 0 contains the value “Karma Police.mp3” and index 3 contains the value “Today.mp3”.

Another way of assigning a value to an element in an array is as follows:

```
mySongs(3) = "Today.mp3"
```

The above statement assigns the value “Today.mp3” to the element at index 3.

ACCESSING ELEMENTS IN AN ARRAY

One way in which an array element is accessed is by including its index in brackets after the array name. For example, the following statement displays the third element in the array in a label called **lblOutput**:

```
lblOutput.Text = mySongs(4)
```

A more useful way of going through the elements of an array is by using a **for** loop. A **for** loop is a useful way of accessing the elements of an array because the loop control variable can be used as the array index. For example, the following statement displays each song title of the **mySongs** array in a list box I’ve called **lstSongs**:

```
For i As Integer = 0 To mySongs.Length - 1
        lstSongs.Items.Add(mySongs(i))
Next
```

The loop repeats from 0 to one less than the length of the array and the loop counter (that is, the variable **i**) is used as the array index. Note that the loop iterates from 0 to one less than the length of the array because the property **Length** is a count of the number of elements, not the greatest index value.

Another useful method you can use in a **for** loop that you use to access all the elements of an array is a method called **GetUpperBound()**. This method returns the index of the last element in the array. So, our **for** loop can be re-written as follows:

```
For i As Integer = 0 To mySongs.GetUpperBound(0)
        lstSongs.Items.Add(mySongs(i))
Next
```

In the above example, the **GetUpperBound()** method returns a value of 4 since the last index of the **mySongs** array is 4. This, of course, produces the same result we got when we used the statement **mySongs.Length - 1**. You'll notice that the **GetUpperBound()** method is passed a value of 0 when the array is a one-dimensional array (1 if it's a two-dimensional array). We will discuss multi-dimensional arrays at a later time.

DETERMINING THE LENGTH OF AN ARRAY

The array structure includes a **Length** property, which can be used to determine the length of the array.

```
numElements = mySongs.Length;
```

The above line of code would return the length of my array called **mySongs**, which in this case would be 500.

CHANGING THE SIZE OF AN ARRAY

Once an array has been declared, its size can be changed using one of two approaches. The first approach is uses the Visual Basic keyword **ReDim**, which destroys the array and then recreates it, thus losing any data held in the array. **ReDim** is useful for re-using an existing array, but of little use in terms of dynamically resizing that array since all data is lost.

So if, for example, I wanted to resize my array called **mySongs** so that it stores 1000 songs instead of 500, I would write the following line of code:

```
ReDim mySongs(999)
```

If you want to resize an array without destroying the contents, you can use the **Resize()** method included in the **Array** class. This method takes two parameters: the array that you want resized and the new size of the array. So, for example, if I want to resize my array called **mySongs** so that it stores 1000 songs instead of 500 without losing any data currently contained in the array, I would write the following line of code:

```
Array.Resize(mySongs, 1000)
```

Note that when you use the **Resize()** method, you need to pass the length of the array, unlike when you use **ReDim** where you need to indicate the index of the last element.

THE IndexOutOfRangeException ERROR

A run-time error will occur in your program whenever an invalid index is used. For example, the following assignment statement would produce an **IndexOutOfRangeException** error:

```
Dim names(9) As String
names(10) = "Mr. Bulhao"
```

The above statement would produce an exception error because we are trying to store a value in an index that does not exist in the array. Since the declaration statement suggests that the array contains indexes from 0 to 9, we cannot possibly store a value in index 10.