Chapter 8

Self Portrait

Arrays

Learning Objectives

- *Describe the purpose of an array*
- *List and describe the use of single and multidimensional arrays*
- *Describe how array objects are allocated in memory*
- *Describe the difference between arrays of value types vs. arrays of reference types*
- *Demonstrate your ability to create arrays using array literals*
- *Demonstrate your ability to create single and multidimensional arrays*
- *Describe how to access individual array elements via indexing*
- *Demonstrate your ability to manipulate arrays with iteration statements*
- *Demonstrate your ability to use the Main() method's string array parameter*
- *Use the System.Environment class to get command-line parameters*
- *State the similarities between arrays and strings*
- *Demonstrate your ability to perform common operations on strings*
- *State the differences between composite strings, verbatim strings, and interpolated strings*
- *Use the foreach statement to iterate over the elements in an array*
- *Use Language Integrated Query (LINQ) and lambda expressions to query arrays*

Introduction

The purpose of this chapter is to give you a solid foundational understanding of arrays and their usage. Since arrays enjoy special status in the C# language, you will find them easy to understand and use. This chapter builds upon the material presented in chapters 6 and 7. Here you will learn how to utilize arrays in simple programs and manipulate them with program control-flow statements to yield increasingly powerful programs. As you will soon learn, arrays are powerful data structures that can be used to solve many programming problems. Detailed knowledge of arrays will give you great powers, along with the ability to judge whether an array is right for your particular application.

In this chapter you will learn the meaning of the term *array*, how to create and manipulate single and multidimensional arrays, and how to use arrays in your programs. Starting with single-dimensional arrays of simple predefined value types, you will learn how to declare array references and how to use the `new` operator to dynamically create array objects. To help you better understand the concepts of arrays and their use, I will show you how they are represented in memory. A solid understanding of the memory concepts associated with array allocation helps you to better utilize arrays in your programs. I will then show you how to manipulate single-dimensional arrays using the program control-flow statements you learned in the previous chapter. Understanding the concepts and use of single-dimensional arrays enables you to easily understand the concepts behind multidimensional arrays.

Along the way, you will learn the difference between arrays of value types and arrays of reference types. I will show you how to dynamically allocate array element objects and how to call methods on objects via array element references. I will also explain to you the difference between rectangular and ragged arrays.

Although you will learn a lot about arrays in this chapter, I have omitted some material I feel is best covered later in the book. For instance, I have postponed discussion of how to pass arrays as method arguments until you learn about classes and methods in the next chapter.

What Is An Array?

An array is a contiguous memory allocation of same-sized or homogeneous data type elements. *Contiguous* means the array elements are located one after the other in memory. *Same-sized* means that each array element occupies the same amount of memory space. The size of each array element is determined by the type of objects an array is declared to contain. So, for example, if an array is declared to contain integer types, each element would be the size of an integer and occupy 4 bytes. If, however, an array is declared to contain double types, the size of each element would be 8 bytes. The term *homogeneous* is often used in place of the term *same-sized* to refer to objects having the same data type and therefore the same size. Figure 8-1 illustrates these concepts.

Figure 8-1 shows an array of 5 elements of no specified type. The elements are numbered consecutively, beginning with 1 denoting the first element and 5 denoting the last, or 5th, element in the array. Each array element is referenced or accessed by its array index number. An index number is always one less than the element number it accesses. For example, when you want to access the 1st element of an array, use index number 0. To access the 2nd element of an array, use index number 1, etc.

The number of elements an array contains is referred to as its *length*. The array shown in figure 8-1 contains 5 elements, so it has a length of 5. The index numbers associated with this array will range from 0 to 4 (that is 0 to *[length - 1]*).

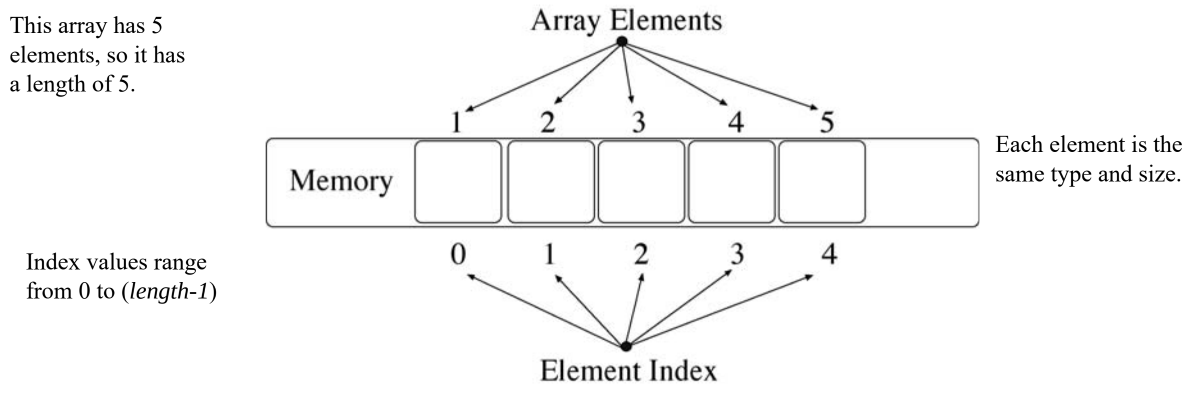

This array has 5 elements, so it has a length of 5.
Array Elements
1 2 3 4 5
Memory
Each element is the same type and size.
Index values range from 0 to (*length-1*)
0 1 2 3 4
Element Index

Figure 8-1: Array Elements are Contiguous and Homogeneous

Specifying Array Types

Array elements can be value types, reference types, or arrays of these types. When you declare an array, you must specify the type its elements will contain. Figure 8-2 illustrates this concept through the use of the array declaration and allocation syntax.

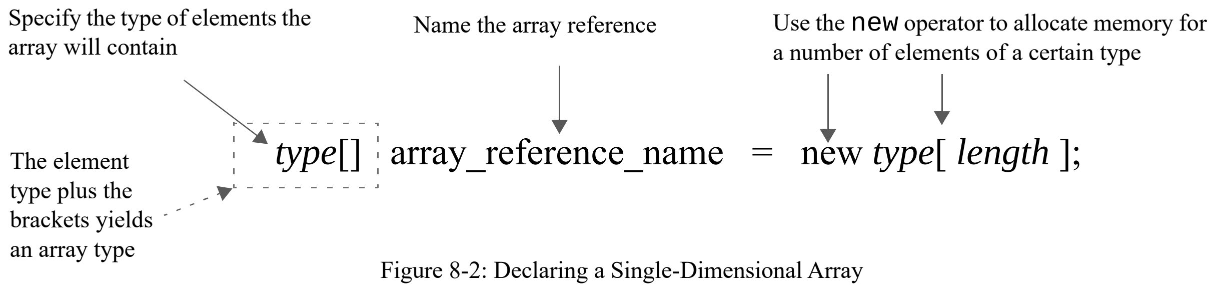

Specify the type of elements the array will contain
Name the array reference
Use the new operator to allocate memory for a number of elements of a certain type
The element type plus the brackets yields an array type
type[] array_reference_name = new *type*[*length*];

Figure 8-2: Declaring a Single-Dimensional Array

Figure 8-2 shows the array declaration and allocation syntax for a single-dimensional array having a particular *type* and *length*. The declaration begins with the array element type. The elements of an array can be value types or reference types. Reference types can include any reference type specified in the .NET Framework, reference types you create, or third-party types created by someone else.

The element type is followed by a set of empty brackets. Single-dimensional arrays use one set of brackets. Multidimensional arrays add either commas or brackets, depending on whether you're declaring a rectangular or jagged array. You will add a comma or a set of brackets for each additional *dimension* or *rank* you want a multidimensional array to have. The element type plus the brackets yield an *array type*. This array type is followed by an identifier that declares the name of the array. To actually allocate memory for an array, use the `new` operator followed by the type of elements the array can contain followed by the length of the array in brackets. The `new` operator returns a reference to the newly created array object and the assignment operator assigns it to the array reference name.

Figure 8-2 combines the act of declaring an array and the act of creating an array object on one line of code. If required, you could declare an array in one statement and create the array in another. For example, the following line of code declares and allocates memory for a single-dimensional array of integers having a length of 5:

```
int[] int_array = new int[5];
```

The following line of code would simply declare an array of floats:

```
float[] float_array;
```

And this line would then allocate enough memory to hold 10 float values:

```
float_array = new float[10];
```

The following line of code would declare a two-dimensional rectangular array of boolean-type elements and allocate some memory:

```
bool[,] boolean_array_2d = new bool[10,10];
```

The following line of code would create a single-dimensional array of strings...

```
String[] string_array = new String[8];
```

...but because it is an array of reference types, each element points to null until a string object is created and assigned to an element.

You will soon learn the details about single and multidimensional arrays. If the preceding concepts seem confusing now just hang in there. By the time you complete this chapter, you will be using arrays like a pro!

Quick Review

Arrays are contiguously allocated memory elements of homogeneous data types. Contiguous means the elements are arranged in memory one after the other. Homogeneous means each element of the array is of the same data type. An array containing *n* elements is said to have a length equal to *n*. Access array elements via their index value, which ranges from 0 to (*length - 1*). The index value of a particular array element is always one less than the element number you wish to access (*i.e.,* the 1st element has index 0, the 2nd element has index 1, ... , the nth element has index n-1)

Functionality Provided By C# Array Types

As you learned in chapter 6, the C# language has two primary data-type categories: value types and reference types. Arrays are a special case of reference types. When you create an array in C#, it is an object just like a reference type object. However, C# arrays possess special features over and above ordinary reference types because they inherit from the System.Array class. This section explains what it means to be an array type.

Array-Type Inheritance Hierarchy

When you declare an array in C#, you specify an array type as was shown previously in figure 8-2. The array you create automatically inherits the functionality provided by the System.Array class, which itself extends from the System.Object class. Figure 8-3 shows the UML inheritance diagram for an array type.

Referring to figure 8-3 — The inheritance from the Array and Object classes is taken care of automatically by the C# language when you declare an array. The Array class is a special class in the .NET Framework in that you cannot derive from it directly to create a new array type subclass. Any attempt to explicitly extend from System.Array in your code will cause a compiler error.

The Array class provides several public properties and methods that make it easy to manipulate arrays. Some of these properties and methods can be accessed via an array reference, while others are meant only

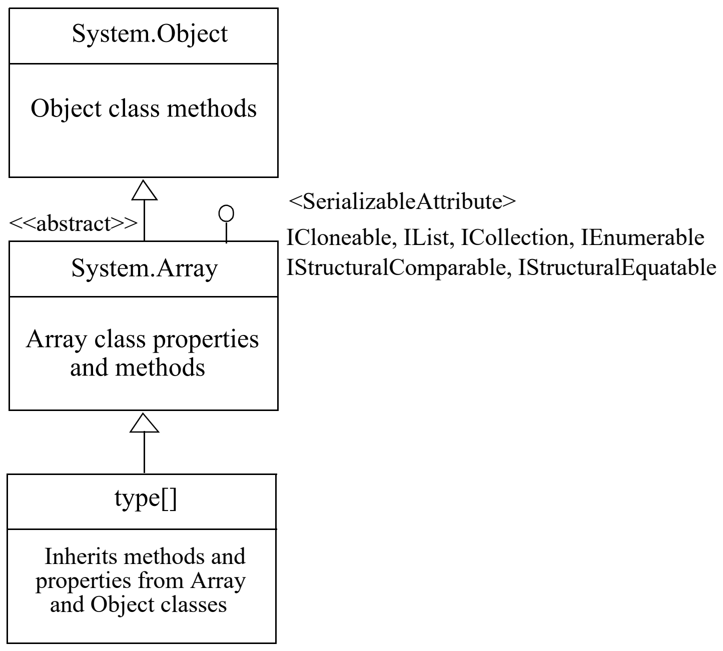

System.Object
Object class methods
<<abstract>>
<SerializableAttribute>
ICloneable, IList, ICollection, IEnumerable
IStructuralComparable, IStructuralEquatable
System.Array
Array class properties and methods
type[]
Inherits methods and properties from Array and Object classes

Figure 8-3: Array-Type Inheritance Hierarchy

to be accessed via the Array class itself. (i.e., static members) You will see examples of the Array class's methods and properties in action as you progress through this chapter. In the meantime, however, it would be a good idea to access the Microsoft Docs website and pay a visit to the System.Array class documentation to learn more of what it has to offer.

Special Properties Of C# Arrays

Table 8-1 summarizes the special properties of C# arrays.

Property	Description
Their length cannot be changed once created.	Array objects have an associated length when they are created. The length of an array cannot be changed after the array is created. However, arrays can be automatically resized with the help of the Array.Resize() method.

Their number of dimensions or rank can be determined by accessing the Rank property.	For example:
	`int[] int_array = new int[5];`
	This code declares a single-dimensional array of five integers. The following line of code prints to the console the number of dimensions int_array contains:
	`Console.WriteLine(int_array.Rank);`

Table 8-1: C# Array Properties

Property	Description
The length of a particular array dimension or rank can be determined via the GetLength() method.	Array objects have a method named GetLength() that returns the value of the length of a particular array dimension or rank. To call the GetLength() method, use the dot operator and the name of the array. For example: `int[] int_array = new int[5];` This code declares and initializes an array of integer elements with length 5. The next line of code prints the length of the int_array to the console: `Console.WriteLine(int_array.GetLength(0));` The GetLength() method is called with an integer argument indicating the desired dimension. In the case of a single-dimensional array, there is only one dimension.
Array bounds are checked by the virtual execution system at runtime.	Any attempt to access elements of an array beyond its declared length will result in a runtime exception. This prevents mysterious data corruption bugs that can manifest themselves when misusing arrays in other languages like C or C++.
Array types directly subclass the System.Array class.	Because arrays subclass System.Array they have the functionality of an Array.
Elements are initialized to default values.	Predefined simple value type array elements are initialized to the default value of the particular value type each element is declared to contain. For example, integer array elements are initialized to zero. Each element of an array of references is initialized to null.

Table 8-1: C# Array Properties

Quick Review

C# array types have special functionality because of their inheritance hierarchy. C# array types directly and automatically inherit the functionality of the System.Array class and implement the ICloneable, IList, ICollection, IEnumerable, IStructuralComparable, and IStructuralEquatable interfaces. Arrays are also serializable.

Creating And Using Single-Dimensional Arrays

This section shows you how to declare, create, and use single-dimensional arrays of both value types and reference types. Once you know how a single-dimensional array works you can easily apply the concepts to multidimensional arrays.

Arrays Of Value Types

The elements of a value type array can be any of the C# predefined value types or value types that you declare (*i.e.*, structures). The predefined value types include *bool*, *byte*, *sbyte*, *char*, *short*, *ushort*, *int*, *uint*, *long*, *ulong*, *float*, *double*, and *decimal*. Example 8.1 shows an array of integers being declared, created, and utilized in a short program.

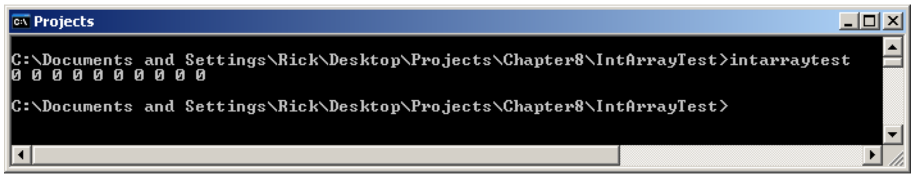
8.1 IntArrayTest.cs

```
using System;

public class IntArrayTest {
  static void Main(){
```

```
5       int[] int_array = new int[10];
6       for(int i = 0; i < int_array.GetLength(0); i++){
7         Console.Write(int_array[i] + " ");
8       }
9       Console.WriteLine();
10    }
11  }
```

Referring to example 8.1 — This program demonstrates several important concepts. First, an array of integers of length 10 is declared and created on line 5. The name of the array is int_array. To demonstrate that each element of the array is automatically initialized to zero, the `for` statement on line 6 iterates over each element of the array beginning with the first element [0] and proceeding to the last element [9], and prints each element value to the console.

Figure 8-4 shows the results of running this program.



```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter8\IntArrayTest>intarraytest
0 0 0 0 0 0 0 0 0 0

C:\Documents and Settings\Rick\Desktop\Projects\Chapter8\IntArrayTest>
```

Figure 8-4: Results of Running Example 8.1

As you can see from looking at figure 8-4, this results in all zeros being printed to the console.

Notice how each element of int_array is accessed via an index value that appears between square brackets appended to the name of the array (*i.e.*, int_array[i]). In this example, the value of i is controlled by the `for` loop.

How Value-Type Array Objects Are Arranged In Memory

Figure 8-5 shows how the integer array int_array declared and created in example 8.1 is represented in memory. The name of the array, int_array, is a reference to an object in memory of type *System.Int32[]*.

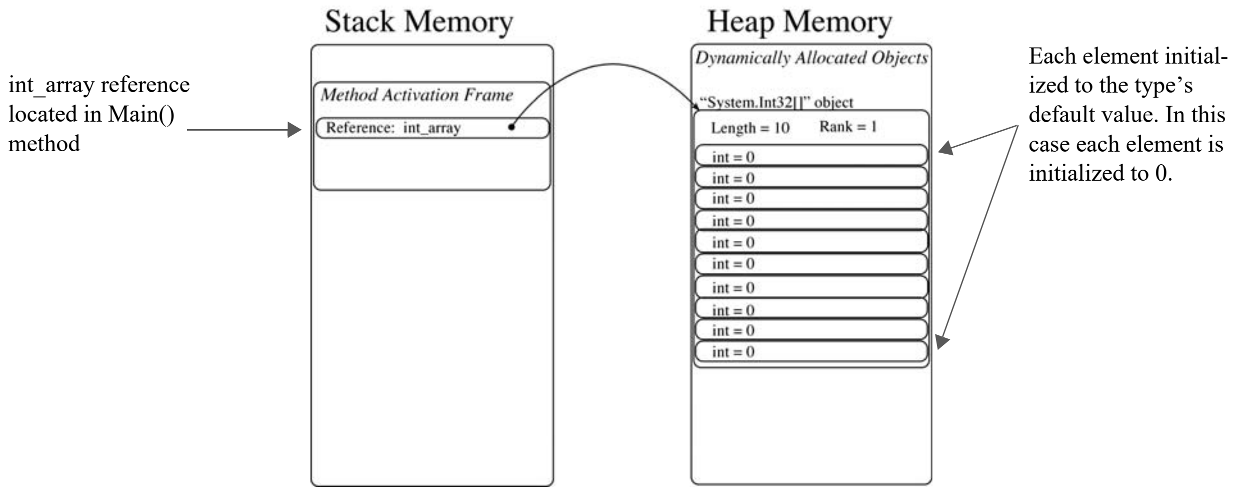


Figure 8-5: Memory Representation of Value Type Array int_array Showing Default Initialization

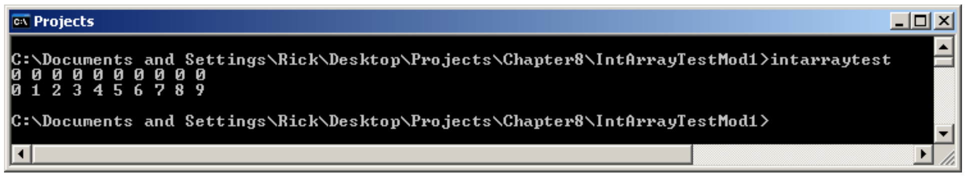
The array object is dynamically allocated on the application's memory heap with the `new` operator. Its memory location is assigned to the int_array reference. At the time of array object creation, each element is initialized to the default value for integers which is 0. The array object's Length property returns the value of the total number of elements in the array, which in this case is 10. The array object's Rank property returns the total number of dimensions in the array, which in this case is 1.

Let's make a few changes to the code given in example 8.1 by assigning some values to the int_array elements. Example 8.2 adds another `for` loop to the program that initializes each element of int_array to the value of the `for` loop's index variable i.

8.2 IntArrayTest.cs (Mod 1)

```
1   using System;
2
3   public class IntArrayTest {
4     static void Main(){
5       int[] int_array = new int[10];
6       for(int i = 0; i < int_array.GetLength(0); i++){
7         Console.Write(int_array[i] + " ");
8       }
9       Console.WriteLine();
10      for(int i = 0; i < int_array.GetLength(0); i++){
11        int_array[i] = i;
12        Console.Write(int_array[i] + " ");
13      }
14      Console.WriteLine();
15    }
16  }
```

Referring to example 8.2 — Notice on line 11 how the value of the second `for` loop's index variable i is assigned directly to each array element. When the array elements print to the console, each element's value has changed except for the first, which is still zero. Figure 8-6 shows the results of running this program.



```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter8\IntArrayTestMod1>intarraytest
0 0 0 0 0 0 0 0 0 0
0 1 2 3 4 5 6 7 8 9

C:\Documents and Settings\Rick\Desktop\Projects\Chapter8\IntArrayTestMod1>
```

Figure 8-6: Results of Running Example 8.2

Figure 8-7 shows the memory representation of int_array after its elements have been assigned their new values.

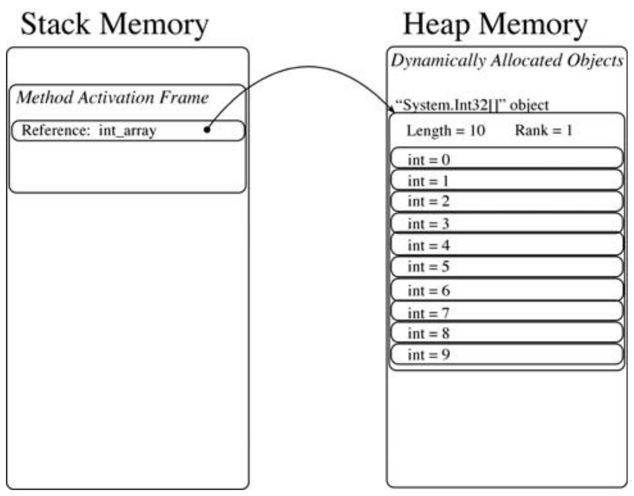


Figure 8-7: Element Values of int_array After Initialization Performed by Second `for` Loop

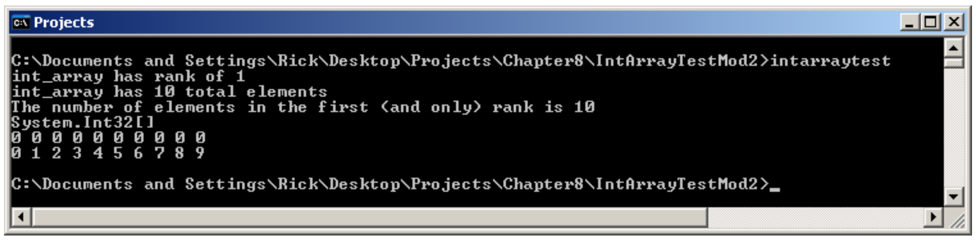
Finding An Array's Type, Rank, And Total Number of Elements

Study the code shown in example 8.3, paying particular attention to lines 6 through 10.

8.3 IntArrayTest.cs (Mod 2)

```
1   using System;
2
3   public class IntArrayTest {
4     static void Main(){
5       int[] int_array = new int[10];
6       Console.WriteLine("int_array has rank of " + int_array.Rank);
7       Console.WriteLine("int_array has " + int_array.Length + " total elements");
8       Console.WriteLine("The number of elements in the first (and only) rank is " +
9                         int_array.GetLength(0));
10      Console.WriteLine(int_array.GetType());
11
12      for(int i = 0; i < int_array.GetLength(0); i++){
13        Console.Write(int_array[i] + " ");
14      }
15      Console.WriteLine();
16      for(int i=0; i<int_array.GetLength(0); i++){
17        int_array[i] = i;
18        Console.Write(int_array[i] + " ");
19      }
20      Console.WriteLine();
21    }
22  }
```

Referring to example 8.3 — Lines 6 through 10 show how to use Array class methods to get information about an array. On line 6, the Rank property is accessed via the int_array reference to print out the number of int_array's dimensions. On line 7, the Length property returns the total number of array elements. On lines 8 and 9, the GetLength() method is called with an argument of 0 to determine the number of elements in the first rank. In the case of single-dimensional arrays, the Length property and GetLength(0) return the same value. On line 10, the GetType() method determines the type of the int_array reference. It returns the value "System.Int32[]" where the single pair of square brackets signifies an array type. Figure 8-8 gives the results of running this program.



```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter8\IntArrayTestMod2>intarraytest
int_array has rank of 1
int_array has 10 total elements
The number of elements in the first (and only) rank is 10
System.Int32[]
0 0 0 0 0 0 0 0 0 0
0 1 2 3 4 5 6 7 8 9

C:\Documents and Settings\Rick\Desktop\Projects\Chapter8\IntArrayTestMod2>_
```

Figure 8-8: Results of Running Example 8.3

Creating Single-Dimensional Arrays Using Array Literal Values

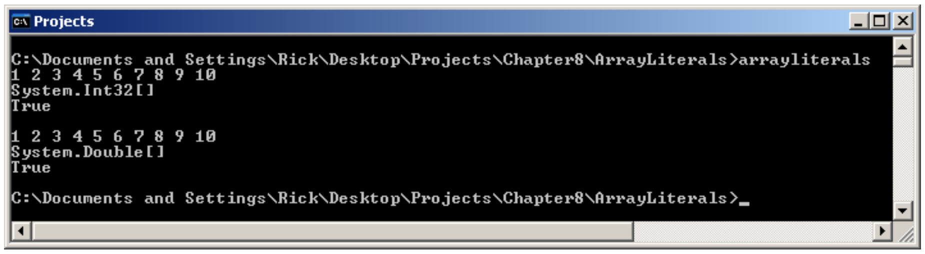
Up to this point you have seen how memory for an array can be allocated using the `new` operator. Another way to allocate memory for an array and initialize its elements at the same time is to specify the contents of the array using *array literal* values. The length of the array is determined by the number of literal values appearing in the declaration. Example 8.4 shows two arrays being declared and created using literal values.

8.4 ArrayLiterals.cs

```
using System;

public class ArrayLiterals {
  static void Main(){
    int[] int_array = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
6      double[] double_array = {1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0};
7
8      for(int i = 0; i < int_array.GetLength(0); i++){
9        Console.Write(int_array[i] + " ");
10     }
11     Console.WriteLine();
12     Console.WriteLine(int_array.GetType());
13     Console.WriteLine(int_array.GetType().IsArray);
14
15     Console.WriteLine();
16
17     for(int i = 0; i < double_array.GetLength(0); i++){
18       Console.Write(double_array[i] + " ");
19     }
20     Console.WriteLine();
21     Console.WriteLine(double_array.GetType());
22     Console.WriteLine(double_array.GetType().IsArray);
23   }
24 }
```

Referring to example 8.4 — The program declares and initializes two arrays using array literal values. On line 5 an array of integers named int_array is declared. The elements of the array are initialized to the values that appear between the braces. Each element's literal value is separated by a comma. The length of the array is determined by the number of literal values appearing between the braces. The length of int_array is 10.

On line 6, an array of doubles named double_array is declared and initialized with double literal values. The contents of both arrays are printed to the console. Array class methods are then used to determine the characteristics of each array and the results are printed to the console. Notice on lines 13 and 22 the use of the IsArray property. It will return true if the reference via which it is called is an array type. Figure 8-9 shows the results of running this program.



```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter8\ArrayLiterals>arrayliterals
1 2 3 4 5 6 7 8 9 10
System.Int32[]
True

1 2 3 4 5 6 7 8 9 10
System.Double[]
True

C:\Documents and Settings\Rick\Desktop\Projects\Chapter8\ArrayLiterals>_
```

Figure 8-9: Results of Running Example 8.4

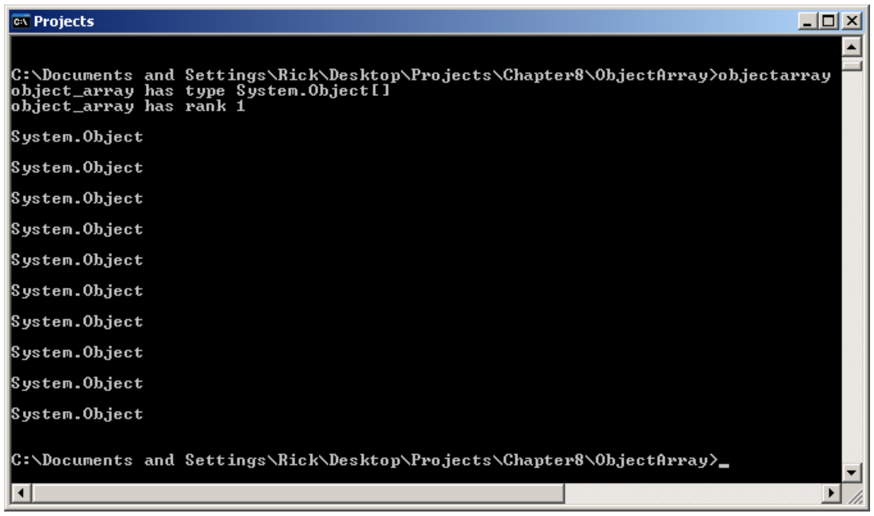
Differences Between Arrays Of Value Types And Arrays Of Reference Types

The key difference between arrays of value types and arrays of reference types is that value-type values can be directly assigned to value-type array elements. The same is not true for reference type elements. In an array of reference types, each element is a reference to an object in memory. When you create an array of references in memory you are *not* automatically creating each element's object. Instead, each reference element is automatically initialized to *null*. You must explicitly create each object you want each reference element to point to. Alternatively, the object must already exist somewhere in memory and be reachable. To illustrate these concepts, I will use an array of Objects. Example 8.5 gives the code for a short program that creates and uses an array of Objects.

8.5 ObjectArray.cs

```
using System;

public class ObjectArray {
  static void Main(){
    Object[] object_array = new Object[10];
    Console.WriteLine("object_array has type " + object_array.GetType());
7       Console.WriteLine("object_array has rank " + object_array.Rank);
8       Console.WriteLine();
9
10      object_array[0] = new Object();
11      Console.WriteLine(object_array[0].GetType());
12      Console.WriteLine();
13
14      object_array[1] = new Object();
15      Console.WriteLine(object_array[1].GetType());
16      Console.WriteLine();
17
18      for(int i = 2; i < object_array.GetLength(0); i++){
19        object_array[i] = new Object();
20        Console.WriteLine(object_array[i].GetType());
21        Console.WriteLine();
22      }
23    }
24  }
```

Figure 8-10 shows the results of running this program.



```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter8\ObjectArray>objectarray
object_array has type System.Object[]
object_array has rank 1

System.Object

System.Object

System.Object

System.Object

System.Object

System.Object

System.Object

System.Object

System.Object

System.Object

C:\Documents and Settings\Rick\Desktop\Projects\Chapter8\ObjectArray>_
```

Figure 8-10: Results of Running Example 8.5

Referring to example 8.5 — On line 5, an array of Objects of length 10 is declared and created. After line 5 executes, the object_array reference points to an array of Objects in memory with each element initialized to null, as is shown in figure 8-11.

On lines 6 and 7, the program writes to the console some information about the object_array, namely, its type and rank. On line 10, a new object of type Object is created and its memory location is assigned to the Object reference located in object_array[0]. The memory picture now looks like that shown in figure 8-12. Line 11 calls the GetType() method on the object pointed to by object_array[0].

The execution of line 14 results in the creation of another object of type Object in memory. The memory picture now looks like that shown in figure 8.13. The `for` statement on line 18 creates the remaining Object objects and assigns their memory locations to the remaining object_array reference elements. Figure 8.14 shows the memory picture after the `for` statement completes execution.

Now that you know the difference between value and reference type arrays, let's see some single-dimensional arrays being put to good use.

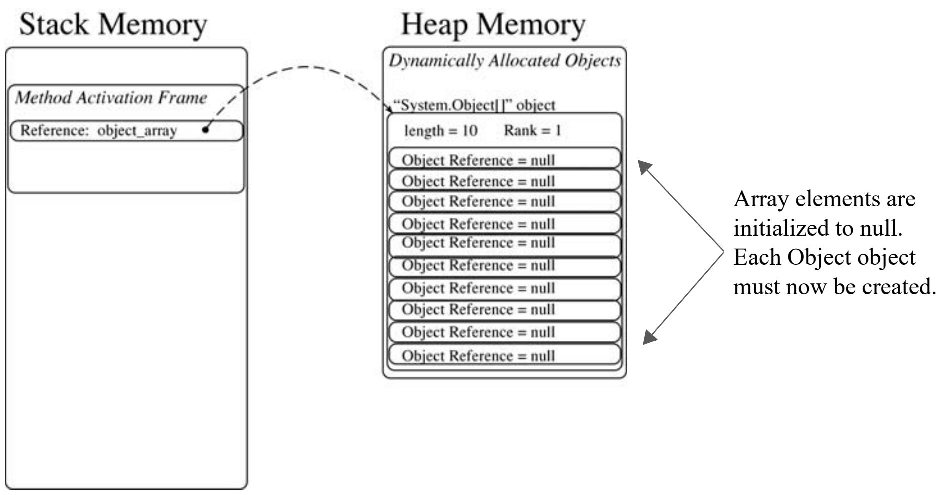


Figure 8-11: State of Affairs After Line 5 of Example 8.5 Executes

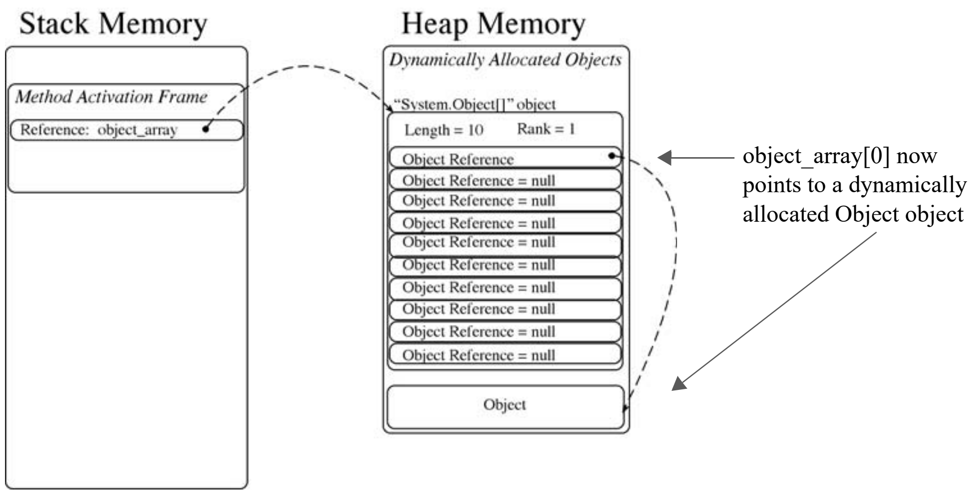


Figure 8-12: State of Affairs After Line 10 of Example 8.5 Executes.

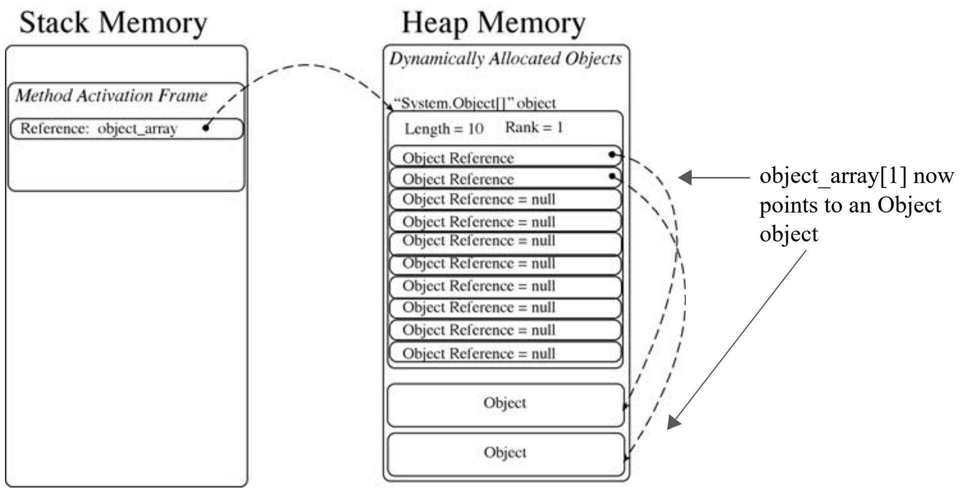


Figure 8-13: State of Affairs After Line 14 of Example 8.5 Executes

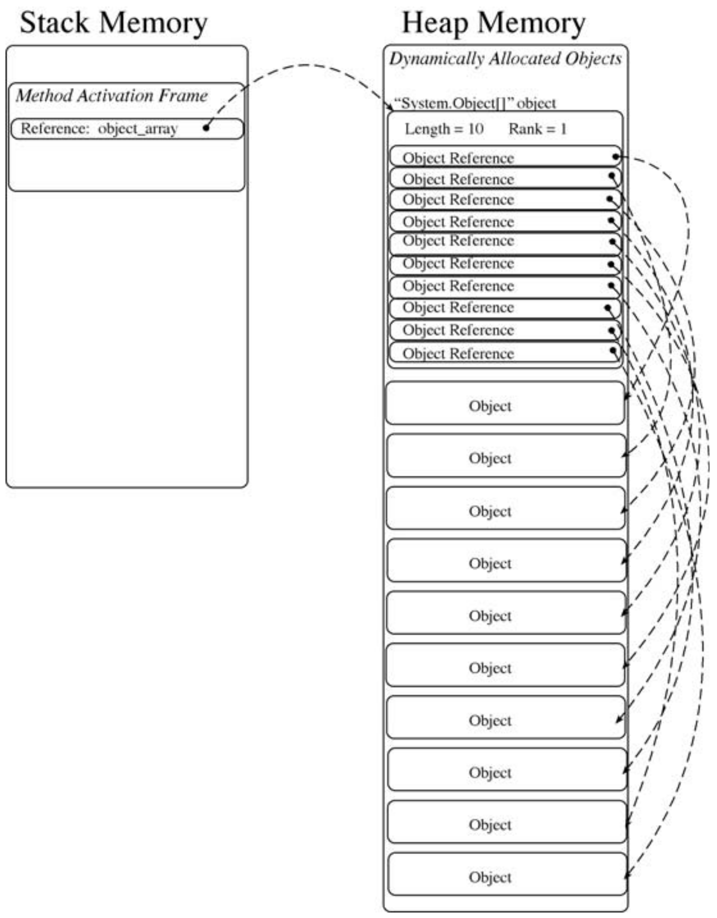


Figure 8-14: Final State of Affairs: All object_array Elements Point to an Object object

Single-dimensional Arrays In Action

This section offers several example programs showing how single-dimensional arrays can be used in programs. These programs represent an extremely small sampling of the usefulness arrays afford.

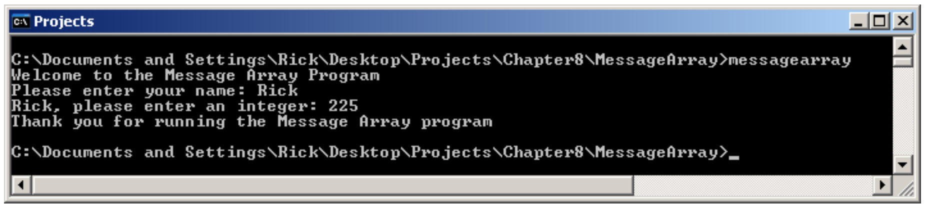
Message Array

One handy use for an array is to store a collection of string messages for later use in a program. Example 8.6 shows how such an array might be utilized.

8.6 MessageArray.cs

```
using System;

public class MessageArray {
  static void Main(){
    String name = null;
    int int_val = 0;

    String[] messages = { "Welcome to the Message Array Program",
              "Please enter your name: ",
              ", please enter an integer: ",
              "You did not enter an integer!",
              "Thank you for running the Message Array program" };

    const int WELCOME_MESSAGE     = 0;
15      const int ENTER_NAME_MESSAGE  = 1;
16      const int ENTER_INT_MESSAGE   = 2;
17      const int INT_ERROR           = 3;
18      const int THANK_YOU_MESSAGE   = 4;
19
20      Console.WriteLine(messages[WELCOME_MESSAGE]);
21      Console.Write(messages[ENTER_NAME_MESSAGE]);
22      name = Console.ReadLine();
23
24      Console.Write(name + messages[ENTER_INT_MESSAGE]);
25
26      try{
27        int_val = Int32.Parse(Console.ReadLine());
28      }catch(FormatException) { Console.WriteLine(messages[INT_ERROR]); }
29
30      Console.WriteLine(messages[THANK_YOU_MESSAGE]);
31    }
32  }
```

Referring to example 8.6 — This program creates a single-dimensional array of strings named messages. It initializes each string element using string literals. On lines 14 through 18, an assortment of constants are declared and initialized. These constants are used to index the messages array as is shown on lines 20 and 21. The program simply asks the user to enter a name followed by a request to enter an integer value. If the user fails to enter an integer, the Int32.Parse() method will throw a FormatException. Figure 8-15 shows the results of running this program.



```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter8\MessageArray>messagearray
Welcome to the Message Array Program
Please enter your name: Rick
Rick, please enter an integer: 225
Thank you for running the Message Array program

C:\Documents and Settings\Rick\Desktop\Projects\Chapter8\MessageArray>_
```

Figure 8-15: Results of Running Example 8.6

Calculating Averages

The program given in example 8.7 calculates class grade averages.

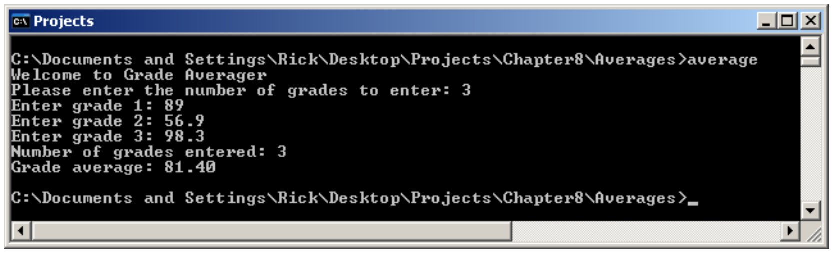
8.7 Average.cs

```
using System;

public class Average {
  static void Main(){
    double[] grades      = null;
    double   total       = 0;
    double   average     = 0;
    int      grade_count = 0;

    Console.WriteLine("Welcome to Grade Averager");
    Console.Write("Please enter the number of grades to enter: ");
    try{
      grade_count = Int32.Parse(Console.ReadLine());
    } catch(FormatException) { Console.WriteLine("You did not enter a number!"); }

    if(grade_count > 0){
      grades = new double[grade_count];
      for(int i = 0; i < grade_count; i++){
        Console.Write("Enter grade " + (i+1) + ": ");
        try{
          grades[i] = Double.Parse(Console.ReadLine());
        } catch(FormatException) { Console.WriteLine("You did not enter a number!"); }
      } //end for
24
25        for(int i = 0; i < grade_count; i++){
26          total += grades[i];
27        } //end for
28
29        average = total/grade_count;
30        Console.WriteLine("Number of grades entered: " + grade_count);
31        Console.WriteLine("Grade average: {0:F2}           ", average);
32
33      }//end if
34    } //end main
35  }// end Average class definition
```

Referring to example 8.7 — An array reference of doubles named grades is declared on line 5 and initialized to null. On lines 6 through 8, several other program variables are declared and initialized.

The program then prompts the user to enter the number of grades. If this number is greater than 0 then it is used on line 17 to create the grades array. The program then enters a `for` loop on line 18, reads each grade from the console, converts it to a double, and assigns it to the ith element of the grades array.

After all the grades are entered into the array, the grades are summed in the `for` loop on line 25. The average is calculated on line 29. Notice how numeric formatting is used on line 31 to properly format the double value contained in the average variable. Figure 8-16 shows the results of running this program



```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter8\Averages>average
Welcome to Grade Averager
Please enter the number of grades to enter: 3
Enter grade 1: 89
Enter grade 2: 56.9
Enter grade 3: 98.3
Number of grades entered: 3
Grade average: 81.40

C:\Documents and Settings\Rick\Desktop\Projects\Chapter8\Averages>_
```

Figure 8-16: Results of Running Example 8.7

Histogram: Letter Frequency Counter

Letter frequency counting is an important part of deciphering messages encrypted using monoalphabetic substitution. Example 8.8 gives the code for a program that counts the occurrences of each letter appearing in a text string and prints the letter frequency display to the console. The program ignores all characters except the 26 letters of the alphabet.

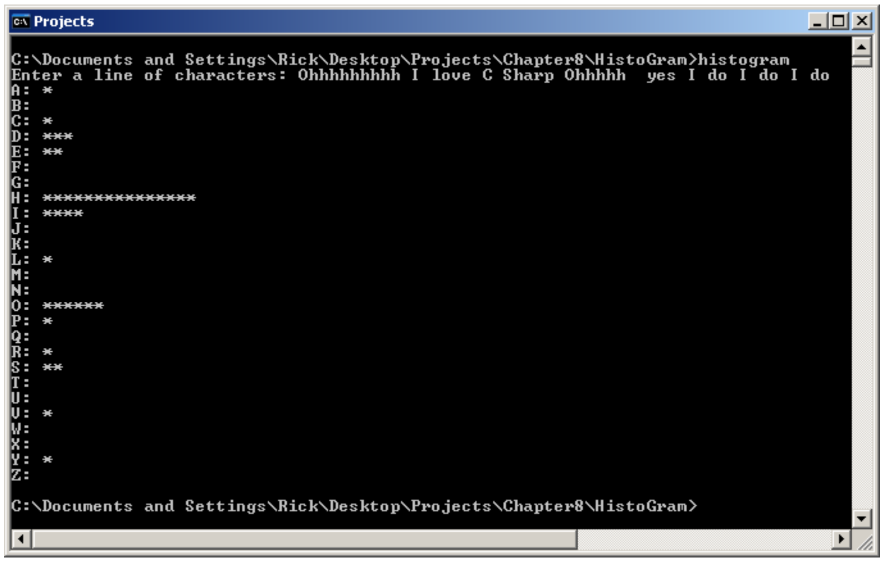
8.8 Histogram.cs

```
using System;

public class Histogram {
  static void Main(String[] args){
    int[] letter_frequencies = new int[26];
    const int A = 0, B = 1, C = 2, D = 3, E = 4, F = 5, G = 6,
              H = 7, I = 8, J = 9, K = 10, L = 11, M = 12, N = 13,
              O = 14, P = 15, Q = 16, R = 17, S = 18, T = 19, U = 20,
              V = 21, W = 22, X = 23, Y = 24, Z = 25;
    String input_string = null;

    Console.Write("Enter a line of characters: ");
    input_string = Console.ReadLine().ToUpper();


    if(input_string != null){
      for(int i = 0; i < input_string.Length; i++){
        switch(input_string[i]){
          case 'A': letter_frequencies[A]++;
                    break;
          case 'B': letter_frequencies[B]++;
                    break;
          case 'C': letter_frequencies[C]++;
                    break;
          case 'D': letter_frequencies[D]++;
                    break;
          case 'E': letter_frequencies[E]++;
                    break;
          case 'F': letter_frequencies[F]++;
                    break;
          case 'G': letter_frequencies[G]++;
                    break;
          case 'H': letter_frequencies[H]++;
                    break;
          case 'I': letter_frequencies[I]++;
                    break;
          case 'J': letter_frequencies[J]++;
                    break;
          case 'K': letter_frequencies[K]++;
                    break;
          case 'L': letter_frequencies[L]++;
                    break;
          case 'M': letter_frequencies[M]++;
                    break;
          case 'N': letter_frequencies[N]++;
                    break;
          case 'O': letter_frequencies[O]++;
                    break;
          case 'P': letter_frequencies[P]++;
                    break;
          case 'Q': letter_frequencies[Q]++;
                    break;
          case 'R': letter_frequencies[R]++;
                    break;
          case 'S': letter_frequencies[S]++;
                    break;
          case 'T': letter_frequencies[T]++;
                    break;
          case 'U': letter_frequencies[U]++;
                    break;
          case 'V': letter_frequencies[V]++;
                    break;
          case 'W': letter_frequencies[W]++;
                    break;
          case 'X': letter_frequencies[X]++;
                    break;
          case 'Y': letter_frequencies[Y]++;
                    break;
          case 'Z': letter_frequencies[Z]++;
                    break;
          default : break;
        } //end switch
      } //end for

      for(int i = 0; i < letter_frequencies.Length; i++){
        Console.Write((char)(i + 65) + ": ");
        for(int j = 0; j < letter_frequencies[i]; j++){
          Console.Write('*');
        } //end for
        Console.WriteLine();
      } //end for

    } //end if
  } // end main
} // end Histogram class definition
```

Referring to example 8.8 — On line 5, an integer array named letter_frequencies is declared and initialized to contain 26 elements, one for each letter of the English alphabet. On lines 6 through 9, several constants are declared and initialized. The constants, named A through Z, are used to index the letter_frequencies array later in the program. On line 10, a string reference named input_string is declared and initialized to null.

The program then prompts the user to enter a line of characters. The program reads this line of text and converts it to upper-case using the String.ToUpper() method. Most of the work is done within the body of the `if` statement that starts on line 16. If the input_string is not null, then the `for` loop will repeatedly execute the `switch` statement, testing each letter of input_string and incrementing the appropriate letter_frequencies element.

Take special note on line 17 of how the length of the input_string is determined using the String class's Length property. Also note that a string's characters can be accessed using array notation. Figure 8-17 gives the results of running this program with a sample line of text.



```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter8\HistoGram>histogram
Enter a line of characters: Ohhhhhhhhh I love C Sharp Ohhhhh  yes I do I do I do
A: *
B:
C: *
D: ***
E: **
F:
G:
H: ***************
I: ****
J:
K:
L: *
M:
N:
O: ******
P: *
Q:
R: *
S: **
T:
U:
V: *
W:
X:
Y: *
Z:

C:\Documents and Settings\Rick\Desktop\Projects\Chapter8\HistoGram>
```

Figure 8-17: Results of Running Example 8.8

Quick Review

Single-dimensional arrays have one dimension — length. You can get an array's length by calling the GetLength() method with an integer argument indicating the particular dimension in which you are interested. Arrays can have elements of either value or reference types. An array type is created by specifying the type name of array elements followed by one set of brackets [ ]. Use System.Array class methods and properties to get information about an array.

Each element of an array is accessed via an index value indicated by an integer within a set of brackets (*e.g.*, array_name[0]). Value-type element values can be directly assigned to array elements. When an array of value types is created, each element is initialized to the type's default value. Each element of an array of references is initialized to null. Each object that a reference element points to must either already exist or be created during program execution.

Creating And Using Multidimensional Arrays

C# supports two kinds of multidimensional arrays: *rectangular* and *ragged*. In this section you will learn how to create and use both kinds of multidimensional arrays. I will also show you how to create multidimensional arrays using the `new` operator as well as how to initialize multidimensional arrays using literal values.

Rectangular Arrays

A *rectangular array* is a multidimensional array whose shape is fixed based on the length of each dimension or rank. All of a rectangular array's dimensions must be specified when the array object is created. Figure 8-18 gives the rectangular array declaration syntax for a two-dimensional rectangular array.

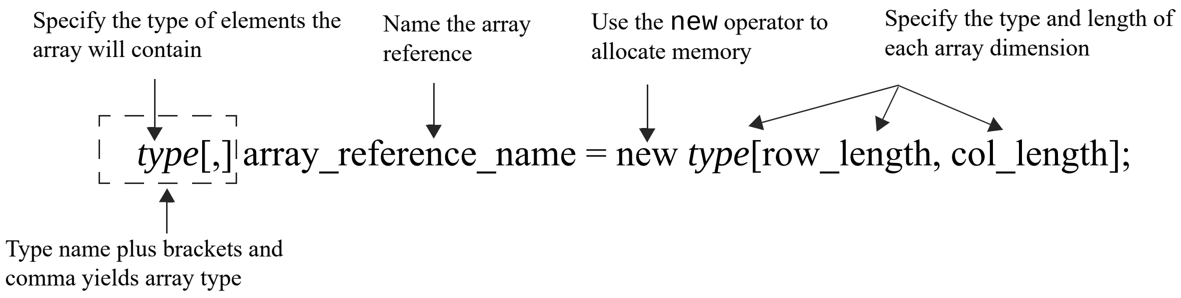

Specify the type of elements the array will contain
Name the array reference
Use the new operator to allocate memory
Specify the type and length of each array dimension
type[,] array_reference_name = new *type*[row_length, col_length];
Type name plus brackets and comma yields array type

Figure 8-18: Rectangular Array Declaration Syntax

Referring to figure 8-18 — The type name combined with the brackets and comma yield the array type. For example, the following line of code declares and creates a two-dimensional rectangular array of integers having 10 rows and 10 columns:

```
int[,] int_2d_array = new int[10,10];
```

A two-dimensional array can be visualized as a grid or matrix comprised of rows and columns, as is shown in figure 8-19. Each element of the array is accessed using two index values, one each for the row and column you wish to access. For example, the following line of code would write to the console the element located in the first row, second column of int_2d_array:

```
Console.WriteLine(int_2d_array[0,1]);
```

Figure 8-19 also includes a few more examples of two-dimensional array element access.

Example 8.9 offers a short program that creates a two-dimensional array of integers and prints their values to the console in the shape of a grid.

8.9 TwoDimensionalArray.cs

```
using System;

public class TwoDimensionalArray {
  static void Main(String[] args){

    try{
      int rows = Int32.Parse(args[0]);
      int cols = Int32.Parse(args[1]);

      int[,] int_2d_array = new int[rows, cols];
      Console.WriteLine("         Array rank: " + int_2d_array.Rank);
      Console.WriteLine("         Array type: " + int_2d_array.GetType());
      Console.WriteLine("Total array elements: " + int_2d_array.Length);
      Console.WriteLine();

      for(int i = 0, element = 1; i<int_2d_array.GetLength(0); i++){
```

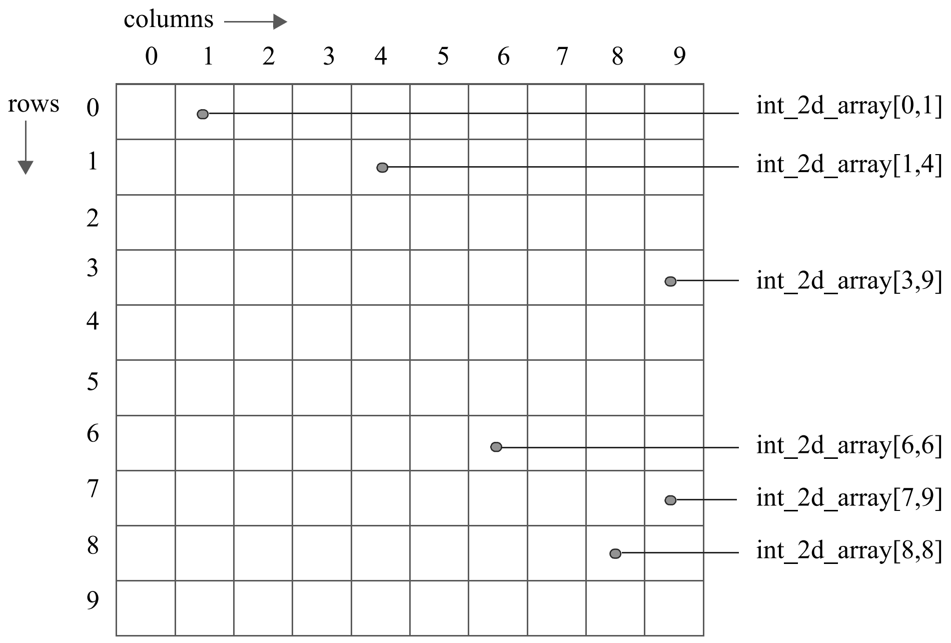


Figure 8-19: Accessing Two-Dimensional Array Elements

```
17        for(int  j = 0; j<int_2d_array.GetLength(1); j++){
18          int_2d_array[i,j] = element++;
19          Console.Write("{0:D3} ",int_2d_array[i,j]);
20        }
21        Console.WriteLine();
22      }
23
24    }catch(IndexOutOfRangeException){
25      Console.WriteLine("This program requires two command-line arguments.");
26    }catch(FormatException){
27      Console.WriteLine("Arguments must be integers!");
28    }
29  }
30 }
```

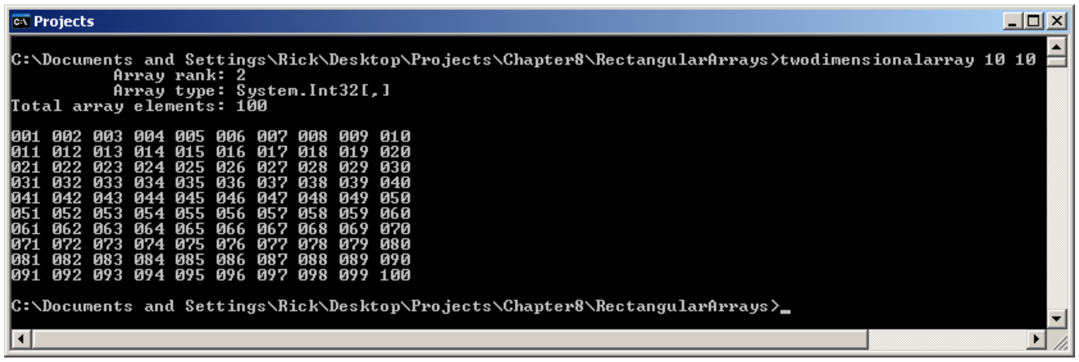
Referring to example 8.9 — When the program executes, the user enters two integer values on the command line for the desired row and column lengths. These values are read and converted on lines 7 and 8, respectively. The two-dimensional array of integers is created on line 10, followed by several lines of code that writes some information about the array including its rank, type, and total number of elements to the console. The nested `for` statement beginning on line 16 *iterates* over each element of the array. Notice that the outer `for` statement on line 16 declares an extra variable named element. It's used in the body of the inner `for` loop to keep count of how many elements the array contains so that its value can be assigned to each array element. The statement on line 19 prints each array element's value to the console with the help of numeric formatting. Figure 8-20 gives the results of running this program.

Initializing Rectangular Arrays With Array Literals

Rectangular arrays can be initialized using literal values in an array initializer expression. Study the code offered in example 8.10.

8.10 RectangularLiterals.cs

```
1  using System;
2
3  public class RectangularLiterals {
4    static void Main(){
5      char[,] char_2d_array = { {'a', 'b', 'c'},
```

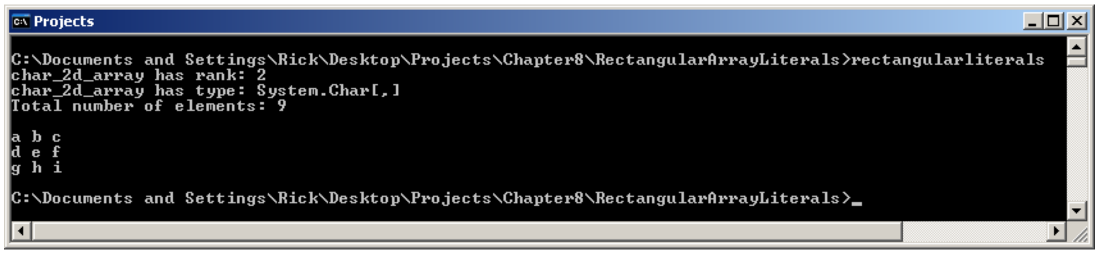


```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter8\RectangularArrays>twodimensionalarray 10 10
          Array rank: 2
          Array type: System.Int32[,]
Total array elements: 100

001 002 003 004 005 006 007 008 009 010
011 012 013 014 015 016 017 018 019 020
021 022 023 024 025 026 027 028 029 030
031 032 033 034 035 036 037 038 039 040
041 042 043 044 045 046 047 048 049 050
051 052 053 054 055 056 057 058 059 060
061 062 063 064 065 066 067 068 069 070
071 072 073 074 075 076 077 078 079 080
081 082 083 084 085 086 087 088 089 090
091 092 093 094 095 096 097 098 099 100

C:\Documents and Settings\Rick\Desktop\Projects\Chapter8\RectangularArrays>_
```

Figure 8-20: Results of Running Example 8.9

```
6                                   {'d', 'e', 'f'},
7                                   {'g', 'h', 'i'} };
8
9       Console.WriteLine("char_2d_array has rank: " + char_2d_array.Rank);
10      Console.WriteLine("char_2d_array has type: " + char_2d_array.GetType());
11      Console.WriteLine("Total number of elements: " + char_2d_array.Length);
12      Console.WriteLine();
13
14      for(int i = 0; i<char_2d_array.GetLength(0); i++){
15        for(int j = 0; j<char_2d_array.GetLength(1); j++){
16          Console.Write(char_2d_array[i,j] + " ");
17        }
18        Console.WriteLine();
19      }
20    }
21  }
```

Referring to example 8.10 — A two-dimensional array of chars named char_2d_array is declared and initialized on line 5 to have 3 rows and 3 columns. Notice how each row of characters appears in a comma-separated list between a set of braces. Each row of initialization data is itself separated from the next row by a comma, except for the last row of data on line 7. Lines 9 through 11 write some information about the character array to the console, namely, its rank, type, and total number of elements. The nested `for` statement beginning on line 14 iterates over the array and prints each character to the console in the form of a grid. Figure 8-21 shows the results of running this program.



```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter8\RectangularArrayLiterals>rectangularliterals
char_2d_array has rank: 2
char_2d_array has type: System.Char[,]
Total number of elements: 9

a b c
d e f
g h i

C:\Documents and Settings\Rick\Desktop\Projects\Chapter8\RectangularArrayLiterals>_
```

Figure 8-21: Results of Running Example 8.10

Ragged Arrays

A *ragged array* is an array of arrays. Ragged arrays can be any number of dimensions, but the last, or rightmost, dimension is omitted from the array creation expression. Each rightmost array object must then be dynamically created during program execution, resulting in the possibility that the array dimensions may differ in length, hence the name ragged array. The Length property returns the number of array elements declared in the leftmost dimension.

Figure 8.22 shows the ragged array declaration syntax for a two-dimensional ragged array.

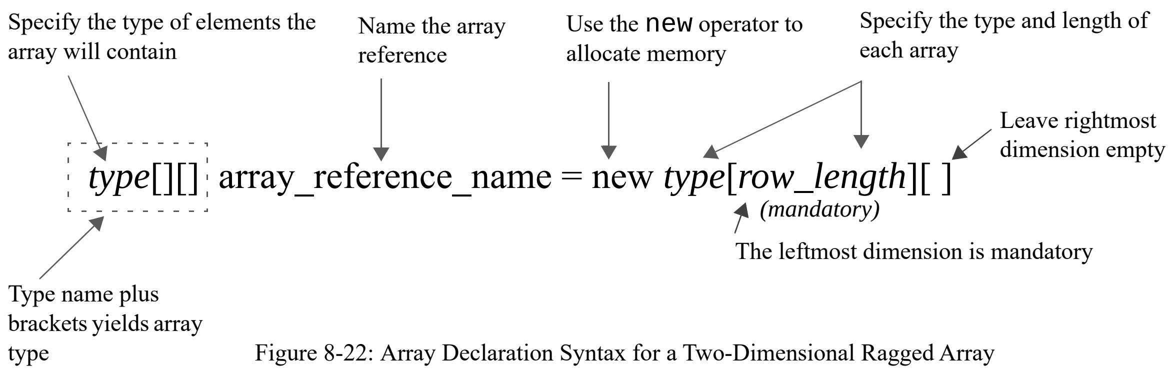

Specify the type of elements the array will contain
Name the array reference
Use the new operator to allocate memory
Specify the type and length of each array
Leave rightmost dimension empty
type[][] array_reference_name = new type[row_length][ ]
(mandatory)
The leftmost dimension is mandatory
Type name plus brackets yields array type

Figure 8-22: Array Declaration Syntax for a Two-Dimensional Ragged Array

Example 8.11 gives a short program showing the use of a ragged array.

8.11 Ragged2dArray.cs

```
1   using System;
2
3   public class Ragged2dArray {
4     static void Main(){
5       int[][] ragged_2d_array = new int[10][];
6
7       Console.WriteLine("ragged_2d_array has rank: " + ragged_2d_array.Rank);
8       Console.WriteLine("ragged_2d_array has type: " + ragged_2d_array.GetType());
9       Console.WriteLine("Total number of elements: " + ragged_2d_array.Length);
10      Console.WriteLine();
11
12      for(int i = 0; i<ragged_2d_array.GetLength(0); i++){
13        ragged_2d_array[i] = new int[i+1];
14      }
15
16      for(int i = 0; i<ragged_2d_array.GetLength(0); i++){
17        for(int j = 0; j<ragged_2d_array[i].GetLength(0); j++){
18          Console.Write(ragged_2d_array[i][j] + " ");
19        }
20        Console.WriteLine();
21      }
22    }
23  }
```

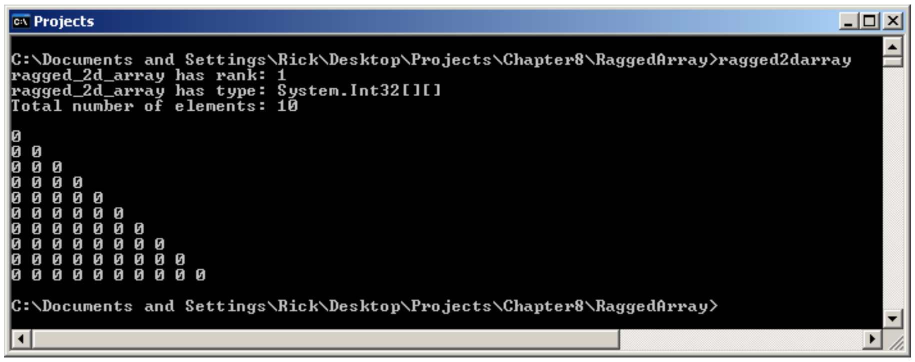
Referring to example 8.11 — On line 5 a two-dimensional ragged array of integers is declared and created. Lines 7 through 9 write some information about the array including its rank, type, and total number of elements to the console. The `for` statement beginning on line 12 creates 10 new arrays of varying lengths and assigns their references to each element of ragged_2d_array. The next `for` statement on line 16 iterates over the ragged two-dimensional array structure and writes the value of each element to the console. Figure 8-23 shows the results of running this program.

Multidimensional Arrays In Action

The example presented in this section shows how single and multidimensional arrays can be used together effectively.

Weighted Grade Tool

Example 8.12 gives the code for a class named WeightedGradeTool. The program calculates a student's final grade based on weighted grades.



```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter8\RaggedArray>ragged2darray
ragged_2d_array has rank: 1
ragged_2d_array has type: System.Int32[][]
Total number of elements: 10

0
0 0
0 0 0
0 0 0 0
0 0 0 0 0
0 0 0 0 0 0
0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0

C:\Documents and Settings\Rick\Desktop\Projects\Chapter8\RaggedArray>
```

Figure 8-23: Results of Running Example 8.11

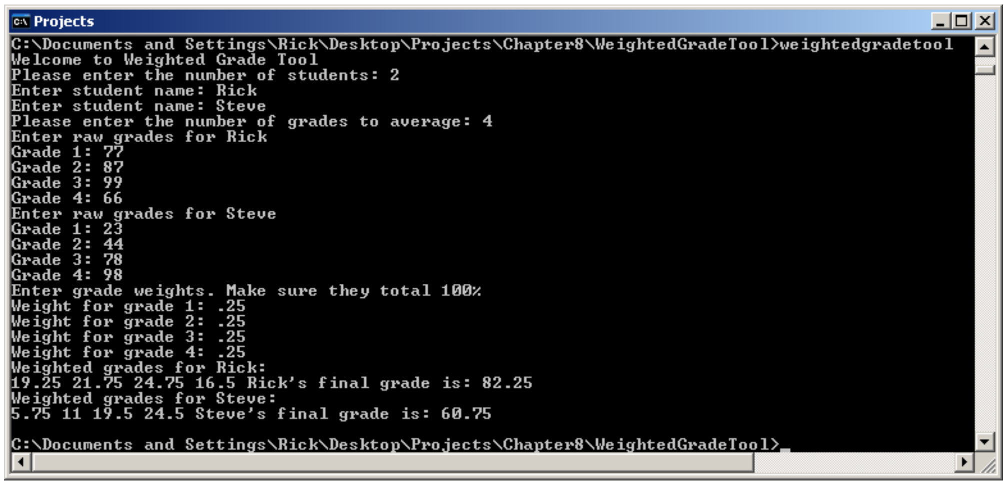
8.12 WeightedGradeTool.cs

```
using System;

public class WeightedGradeTool {
  static void Main() {

    double[,] grades = null;
    double[] weights = null;
    String[] students = null;
    int student_count = 0;
    int grade_count = 0;
    double final_grade = 0;

    Console.WriteLine("Welcome to Weighted Grade Tool");

    /*************** get student count ********************/
    Console.Write("Please enter the number of students: ");
    try {
      student_count = Int32.Parse(Console.ReadLine());
    }catch (FormatException) {
        Console.WriteLine("That was not an integer!");
        Console.WriteLine("Student count will be set to 3.");
        student_count = 3;
    }


    if (student_count > 0) {
      students = new String[student_count];
      /**************** get student names *********************/
      for (int i = 0; i < students.Length; i++) {
        Console.Write("Enter student name: ");
        students[i] = Console.ReadLine();
      }

      /*************** get number of grades per student **********/
      Console.Write("Please enter the number of grades to average: ");
      try {
        grade_count = Int32.Parse(Console.ReadLine());
      }catch (FormatException) {
          Console.WriteLine("That was not an integer!");
          Console.WriteLine("Grade count will be set to 3.");
          grade_count = 3;
      }

      /***************** get raw grades ****************************/
      grades = new double[student_count, grade_count];
      for (int i = 0; i < grades.GetLength(0); i++) {
        Console.WriteLine("Enter raw grades for " + students[i]);
    for (int j = 0; j < grades.GetLength(1); j++) {
     Console.Write("Grade " + (j + 1) + ": ");
     try {
      grades[i, j] = Double.Parse(Console.ReadLine());
     }catch (FormatException) {
       Console.WriteLine("That was not a double!");
       Console.WriteLine("Grade will be set to 100");
       grades[i, j] = 100;
     }
    }//end inner for
   }

   /***************** get grade weights ********************/
   weights = new double[grade_count];
   Console.WriteLine("Enter grade weights. Make sure they total 100%");
   for (int i = 0; i < weights.Length; i++) {
    Console.Write("Weight for grade " + (i + 1) + ": ");
    try {
     weights[i] = Double.Parse(Console.ReadLine());
    }catch (FormatException) {
      Console.WriteLine("That was not a double!");
      Console.WriteLine("The weight will be set to .25");
      weights[i] = .25;
    }
   }

   /****************** calculate weighted grades *******************/
   for (int i = 0; i < grades.GetLength(0); i++) {
    for (int j = 0; j < grades.GetLength(1); j++) {
     grades[i, j] *= weights[j];
    }//end inner for
   }

   /***************** calculate and print final grade ********************/
   for (int i = 0; i < grades.GetLength(0); i++) {
    Console.WriteLine("Weighted grades for " + students[i] + ": ");
    final_grade = 0;
    for (int j = 0; j < grades.GetLength(1); j++) {
     final_grade += grades[i, j];
     Console.Write(grades[i, j] + " ");
    }//end inner for
    Console.WriteLine(students[i] + "'s final grade is: " + final_grade);
   }
  }// end if
 }// end Main
}// end class
```

Figure 8-24 shows the results of running this program.

Quick Review

C# supports two kinds of multidimensional arrays: *rectangular* and *ragged*. A *rectangular array* is a multidimensional array whose shape is fixed based on the length of each *dimension* or *rank*. All of a rectangular array's dimensions must be specified when the array object is created. A *ragged array* is an array of arrays. Ragged arrays can be any number of dimensions, but the last, or rightmost, dimension is omitted from the array creation expression. Each rightmost array object must then be created during program execution, introducing the possibility that the array's dimensions may differ in length.



```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter8\WeightedGradeTool>weightedgradetool
Welcome to Weighted Grade Tool
Please enter the number of students: 2
Enter student name: Rick
Enter student name: Steve
Please enter the number of grades to average: 4
Enter raw grades for Rick
Grade 1: 77
Grade 2: 87
Grade 3: 99
Grade 4: 66
Enter raw grades for Steve
Grade 1: 23
Grade 2: 44
Grade 3: 78
Grade 4: 98
Enter grade weights. Make sure they total 100%
Weight for grade 1: .25
Weight for grade 2: .25
Weight for grade 3: .25
Weight for grade 4: .25
Weighted grades for Rick:
19.25 21.75 24.75 16.5 Rick's final grade is: 82.25
Weighted grades for Steve:
5.75 11 19.5 24.5 Steve's final grade is: 60.75

C:\Documents and Settings\Rick\Desktop\Projects\Chapter8\WeightedGradeTool>_
```

Figure 8-24: Results of Running Example 8.12

The Main() Method's String Array

Now that you have a better understanding of arrays, the Main() method's string array should make more sense. This section explains the purpose and use of the Main() method's string array.

Purpose And Use Of The Main() Method's String Array

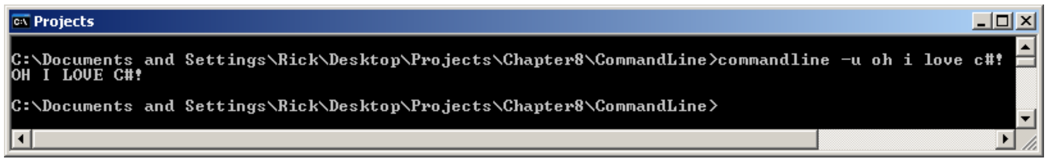
The purpose of the Main() method's string array is to enable C# applications to accept and act upon command-line arguments. The csc compiler is an example of a program that takes command-line arguments, the most important of which is the name of the file to compile. This chapter and the previous chapter also gave several examples of accepting program input via the command line. Now that you are armed with a better understanding of how arrays work, you have the knowledge to write programs that accept and process command-line arguments.

Example 8.13 gives a short program that accepts a line of text as a command-line argument and displays it in lower or upper-case depending on the first command-line argument.

8.13 CommandLine.cs

```
using System;
using System.Text;

public class CommandLine {
  static void Main(String[] args){
    StringBuilder sb = null;
    bool upper_case = false;
    int start_index = 0;

    /********** check for upper-case option **************/
    if(args.Length > 0){
      switch(args[0][0]){ // get the first character of the first argument
        case '-': if(args[0].Length > 1){
                    switch(args[0][1]){ // get the second character of the first argument
                      case 'U':
                      case 'u': upper_case = true;
                                break;
                      default: upper_case = false;
                               break;
                    }
                  }
                  start_index = 1;
                  break;
        default: upper_case = false;
                 break;
      }// end outer switch

      sb = new StringBuilder();   //create StringBuffer object

      /******* process text string *********************/
      for(int i = start_index; i < args.Length; i++){
        sb.Append(args[i] + " ");
      }//end for

      if(upper_case){
        Console.WriteLine(sb.ToString().ToUpper());
      }else {
        Console.WriteLine(sb.ToString().ToLower());
      }//end if/else

    } else { Console.WriteLine("Usage: CommandLine [-U | -u] Text string"); }
  }//end main
}//end class
```

Figure 8.25 shows the results of running this program.



```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter8\CommandLine>commandline -u oh i love c#!
OH I LOVE C#!

C:\Documents and Settings\Rick\Desktop\Projects\Chapter8\CommandLine>
```

Figure 8-25: Results of Running Example 8.13

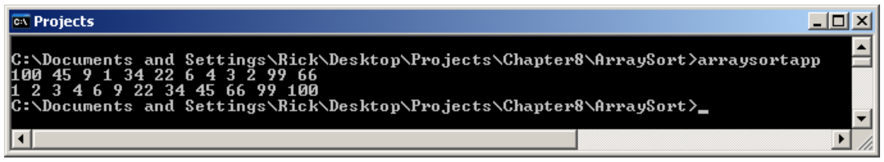
Manipulating Arrays With The System.Array Class

The .NET platform makes it easy to perform common array manipulations such as searching and sorting with the System.Array class. Example 8.14 offers a short program that shows the Array class in action sorting an array of integers.

8.14 ArraySortApp.cs

```
using System;

public class ArraySortApp {
  static void Main() {
    int[] int_array = { 100, 45, 9, 1, 34, 22, 6, 4, 3, 2, 99, 66 };

    for (int i = 0; i < int_array.Length; i++) {
      Console.Write(int_array[i] + " ");
    }
    Console.WriteLine();

    Array.Sort(int_array);

    for (int i = 0; i < int_array.Length; i++) {
      Console.Write(int_array[i] + " ");
    }
  } // end Main() method
} // end ArraySortApp class definition
```

Figure 8-26 shows the results of running this program.



```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter8\ArraySort>arraysortapp
100 45 9 1 34 22 6 4 3 2 99 66
1 2 3 4 6 9 22 34 45 66 99 100
C:\Documents and Settings\Rick\Desktop\Projects\Chapter8\ArraySort>_
```

Figure 8-26: Results of Running Example 8.14

Operations On Strings

Strings deserve their very own section in this chapter because, as you will soon see, they are closely related to arrays. In this section I'll show you how to perform common operations on strings like splitting a string into sections based on a delimiter character, converting a string to upper, lower, and title-case, and how to access substrings. You'll find these common operations helpful as you tackle more challenging projects. I'll also discuss the differences between composite strings, verbatim strings, and interpolated strings. Interpolated strings are new to C# 6. Finally, I'll show you how to use the Environment class to get command-line parameters, which, as you have already learned, is an array of strings.

Common Operations on Strings

The operations discussed in this section represent the most common operations you'll perform on string object. You've encountered a few of these operations already in previous examples. I repeat those operations here, like string concatenation using the '+' operator to highlight additional considerations you need to be aware of.

Concatenation with the '+' Operator

You already know that you can join, or concatenate, one string to another using the overloaded '+' operator, as you've seen in multiple examples presented earlier in the text. Here are a few things to keep in mind when concatenating strings:

Strings are Immutable – Concatenation Results in a New String Object

String objects are immutable, meaning you can't change them once they are created. Take the following code snippet:

```
string s1 = “Hello”;
s1 = s1 + “ World!”;
```

The string literal “ World!” is not appended to s1, rather, a new string object is created from the concatenation of s1 and “ World!” and s1 is set to point to that new string object.

Use Concatenation to Improve Long String Literals and Constant Readability

You'll see code formatted like this in many examples in this book:

```
private const string UPDATE_EMPLOYEE = "UPDATE tbl_employee " +
 "SET FirstName = " + FIRST_NAME +
 ", MiddleName = " + MIDDLE_NAME +
 ", LastName = " + LAST_NAME +
 ", Birthday = " + BIRTHDAY +
 ", Gender = " + GENDER +
 ", Picture = " + PICTURE + " " +
  "WHERE EmployeeID = " + EMPLOYEE_ID;
```

Here, the string concatenation operator is used to break the rather long UPDATE_EMPLOYEE string constant into smaller chunks to make it easier to read and understand. This concatenation operation is performed at compile time.

Use StringBuilder.Append() to Build Up Large Strings at Runtime

If you find yourself concatenating lots of strings at runtime, use the StringBuilder.Append() method instead.

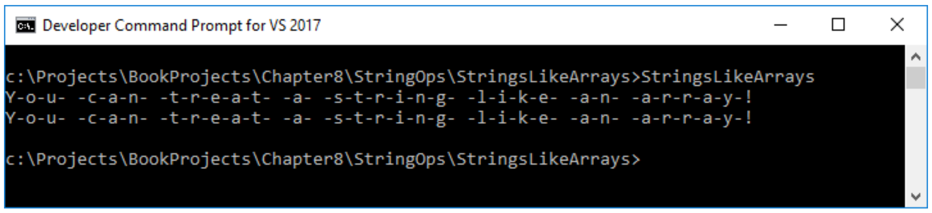
Treating a String like an Array

The String class defines an indexer property that enables you to access each character of the string using array indexer notation. Remember, a string is immutable and the string's indexer is read-only as is shown in example 8.15.

8.15 StringsLikeArrays.cs

```
using System;

public class StringsLikeArrays {
  public static void Main(){
    string s1 = "You can treat a string like an array!";

    //using the foreach statement
    foreach(char c in s1){
      if(c != '!'){
        Console.Write(c + "-");
      }else{
        Console.WriteLine(c);
      }
    }

    //using array indexing
    for(int i=0; i<s1.Length; i++){
      if(s1[i] != '!'){
        Console.Write(s1[i] + "-");
      }else{
        Console.WriteLine(s1[i]);
      }
    }
  } // end Main()
} // end StringsLikeArrays class definition
```

Figure 8-27 shows the results of running this program.



```
c:\Projects\BookProjects\Chapter8\StringOps\StringsLikeArrays>StringsLikeArrays
Y-o-u- -c-a-n- -t-r-e-a-t- -a- -s-t-r-i-n-g- -l-i-k-e- -a-n- -a-r-r-a-y-!
Y-o-u- -c-a-n- -t-r-e-a-t- -a- -s-t-r-i-n-g- -l-i-k-e- -a-n- -a-r-r-a-y-!

c:\Projects\BookProjects\Chapter8\StringOps\StringsLikeArrays>
```

Figure 8-27: Results of Running Example 8.15

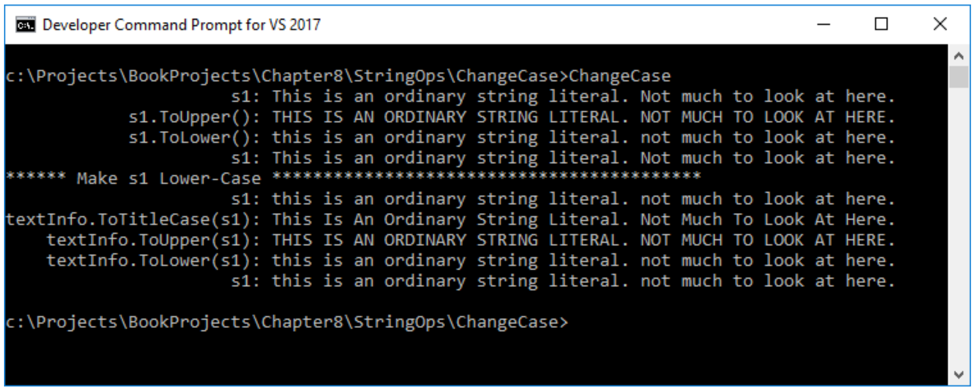
Converting Strings to Upper-Case, Lower-Case, and Title-Case

You'll encounter many situations where you'll need to change the case of a string. The String class provides the ToUpper() and ToLower() methods to change a string to upper-case and lower-case respectively. You can change a string to title-case, but you'll need to enlist the help from the System.Threading and System.Globalization namespaces as is shown in example 8.16.

8.16 ChangeCase.cs

```
using System;
using System.Threading;
using System.Globalization;

public class ChangeCase {

  public static void Main(){
    string s1 = "This is an ordinary string literal. Not much to look at here.";

    // ToUpper() and ToLower() return new strings.
    // Original string is unaffected
    Console.WriteLine("                      s1: " + s1);
    Console.WriteLine("            s1.ToUpper(): " + s1.ToUpper());
    Console.WriteLine("            s1.ToLower(): " + s1.ToLower());
    Console.WriteLine("                      s1: " + s1);
    Console.WriteLine("****** Make s1 Lower-Case *************************************");
    // Make s1 a lower-case string
    s1 = s1.ToLower();
    Console.WriteLine("                      s1: " + s1);

    CultureInfo cultureInfo   = Thread.CurrentThread.CurrentCulture;
    TextInfo textInfo = cultureInfo.TextInfo;

    Console.WriteLine("textInfo.ToTitleCase(s1): " + textInfo.ToTitleCase(s1));
    Console.WriteLine("    textInfo.ToUpper(s1): " + textInfo.ToUpper(s1));
    Console.WriteLine("    textInfo.ToLower(s1): " + textInfo.ToLower(s1));
    Console.WriteLine("                      s1: " + s1);

  } // end Main()
} // end ChangeCase class definition
```

Figure 8-28 shows the results of running this program.



```
c:\Projects\BookProjects\Chapter8\StringOps\ChangeCase>ChangeCase
                      s1: This is an ordinary string literal. Not much to look at here.
            s1.ToUpper(): THIS IS AN ORDINARY STRING LITERAL. NOT MUCH TO LOOK AT HERE.
            s1.ToLower(): this is an ordinary string literal. not much to look at here.
                      s1: This is an ordinary string literal. Not much to look at here.
****** Make s1 Lower-Case *****************************************
                      s1: this is an ordinary string literal. not much to look at here.
textInfo.ToTitleCase(s1): This Is An Ordinary String Literal. Not Much To Look At Here.
    textInfo.ToUpper(s1): THIS IS AN ORDINARY STRING LITERAL. NOT MUCH TO LOOK AT HERE.
    textInfo.ToLower(s1): this is an ordinary string literal. not much to look at here.
                      s1: this is an ordinary string literal. not much to look at here.

c:\Projects\BookProjects\Chapter8\StringOps\ChangeCase>
```

Figure 8-28: Results of Running Example 8.16

Splitting Strings into Tokens

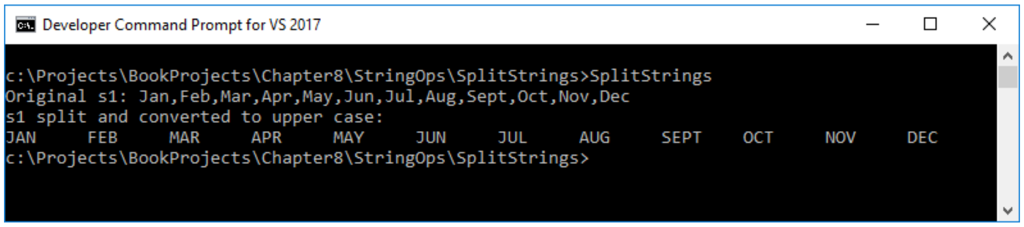
Real-world programming often requires ingesting long streams of data and breaking it into smaller chunks for processing. A prime example of this is reading a comma delimited (csv) file and parsing it into rows and columns. The String.Split() method let's you split long strings into an array of smaller strings, or tokens, base on a delimiter character as is shown in example 8.17.

8.17 SplitStrings.cs

```
using System;

```

```
public class SplitStrings {
  public static void Main(){
    string s1 = "Jan,Feb,Mar,Apr,May,Jun,Jul,Aug,Sept,Oct,Nov,Dec";
      onsole.WriteLine("Original s1: " + s1);

    Console.WriteLine("s1 split and converted to upper-case:");
    string[] months = s1.Split(',');
    foreach(string s in months){
      Console.Write(s.ToUpper() + "\t");
    }
  } // end Main()
} // end SplitStrings class definition
```

Figure 8-29 shows the results of running this program.



```
c:\Projects\BookProjects\Chapter8\StringOps\SplitStrings>SplitStrings
Original s1: Jan,Feb,Mar,Apr,May,Jun,Jul,Aug,Sept,Oct,Nov,Dec
s1 split and converted to upper case:
JAN     FEB     MAR     APR     MAY     JUN     JUL     AUG     SEPT    OCT     NOV     DEC
c:\Projects\BookProjects\Chapter8\StringOps\SplitStrings>
```

Figure 8-29: Result of Running Example 8.17

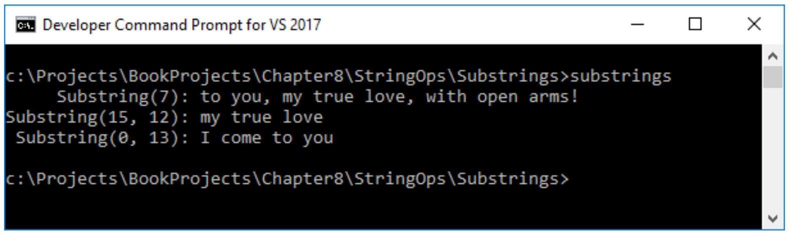
Obtaining Substrings

On many occasions you'll want to access a substring of a larger string. Use the String.Substring() method for this operation as demonstrated in example 8.18.

8.18 SubStrings.cs

```
using System;

public class SubStrings {
  public static void Main(){
    string s1 = "I come to you, my true love, with open arms!";

    Console.WriteLine("     Substring(7): " + s1.Substring(7));
    Console.WriteLine("Substring(15, 12): " + s1.Substring(15,12));
    Console.WriteLine(" Substring(0, 13): " + s1.Substring(0,13));
  } // end Main()
} // end SubStrings class definition
```

Figure 8-30 shows the results of running this program.



```
c:\Projects\BookProjects\Chapter8\StringOps\Substrings>substrings
     Substring(7): to you, my true love, with open arms!
Substring(15, 12): my true love
 Substring(0, 13): I come to you

c:\Projects\BookProjects\Chapter8\StringOps\Substrings>
```

Figure 8-30: Results of Running Example 8.18

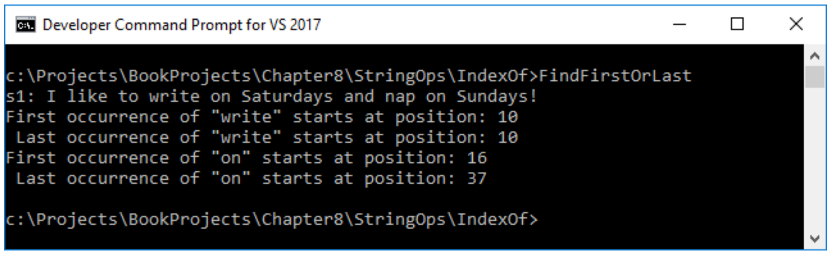
Finding the Index of the First or Last Occurrence of an Element within a String

The String.IndexOf() method returns the index value of the starting character of the first occurrence of either a character or a string. The String.LastIndexOf() method returns the index value of the starting character of the last occurrence of either a character or string, as is shown in example 8.19.

8.19 FindFirstOrLast.cs

```
using System;

public class FindFirstOrLast {
  public static void Main(){
    string s1 = "I like to write on Saturdays and nap on Sundays!";
    int startIndex = s1.IndexOf("write");
    int endIndex = s1.LastIndexOf("write");
    Console.WriteLine("s1: " + s1);
    Console.WriteLine("First occurrence of \"write\" starts at position: " + startIndex);
    Console.WriteLine(" Last occurrence of \"write\" starts at position: " + endIndex);
    startIndex = s1.IndexOf("on");
    endIndex = s1.LastIndexOf("on");
    Console.WriteLine("First occurrence of \"on\" starts at position: " + startIndex);
    Console.WriteLine(" Last occurrence of \"on\" starts at position: " + endIndex);
  } // end Main()
} // end FindFirstOrLast class definition
```

Referring to example 8.19 — Actually, both of these methods are overloaded, so you'll need to consult the String class documentation to get the complete story. Figure 8-31 shows the results of running this program.



```
c:\Projects\BookProjects\Chapter8\StringOps\IndexOf>FindFirstOrLast
s1: I like to write on Saturdays and nap on Sundays!
First occurrence of "write" starts at position: 10
 Last occurrence of "write" starts at position: 10
First occurrence of "on" starts at position: 16
 Last occurrence of "on" starts at position: 37

c:\Projects\BookProjects\Chapter8\StringOps\IndexOf>
```

Figure 8-31: Results of Running Example 8.19

Escape Sequences

Notice in the previous example that to include a double quote character in a string literal I had to "escape" the character using a backslash '\'. This is referred to as an escape sequence. There are a handful of escape sequences you'll need to be familiar with. Some of these are holdovers from the days of teletype terminals.

Escape Sequence	Represents
`\a`	Bell (Alert)
`\b`	Backspace
`\f`	Formfeed
`\n`	New Line
`\r`	Carriage Return
`\t`	Horizontal Tab
`\v`	Vertical Tab
`\'`	Single Quotation Mark

Table 8-2: American National Standards Institute (ANSI) Escape Sequences

Escape Sequence	Represents
`\″`	Double Quotation Mark
`\\`	Backslash
`\?`	Literal Question Mark
`\ ooo`	ASCII Character in Octal Notation
`\x hh`	ASCII Character in Hexadecimal Notation
`\x hhhh`	Unicode Character in Hexadecimal Notation

Table 8-2: American National Standards Institute (ANSI) Escape Sequences

Referring to table 8-2 — If you use ordinary string literals, you'll need to use the escape sequence for the character you wish to include in the string. In example 8.19, I used the `"\""` escape sequence to include double quotes in the string. Later, you'll learn about verbatim strings which allows you to avoid the use of escape sequences.

Numeric Formatting

C# makes it easy to format numeric strings. You have seen several examples of numeric formatting in both this and the previous chapter. You can format numeric results using the String.Format() method or the Console.Write() or Console.WriteLine() methods.

A format string takes the form $C_f nn$ where C_f is a format specifier character and *nn* specifies the number of decimal digits. Table 8-2 lists the standard C# numeric format strings along with some brief example code.

Character	Description	Example Code	Results

C or c	Currency	Console.Write("{0:C}", 4.5);
		Console.Write("{0:C}", -4.5);
			$4.50
			($4.50)

D or d	Decimal	Console.Write("{0:D5}", 45);	00045
E or e	Scientific	Console.Write("{0:E}", 450000);	4.500000E+005

F or f	Fixed-point	Console.Write("{0:F2}", 45);
		Console.Write("{0:F0}", 45);
			45.00
			45

G or g	General	Console.Write("{0:G}", 4.5);	4.5
N or n	Number	Console.Write("{0:N}", 4500000);	4,500,000.00

X or x	Hexadecimal	Console.Write("{0:X}", 450);
		Console.Write("{0:X}", 0xabcd);
			1C2
			ABCD

Table 8-3: Numeric Formatting

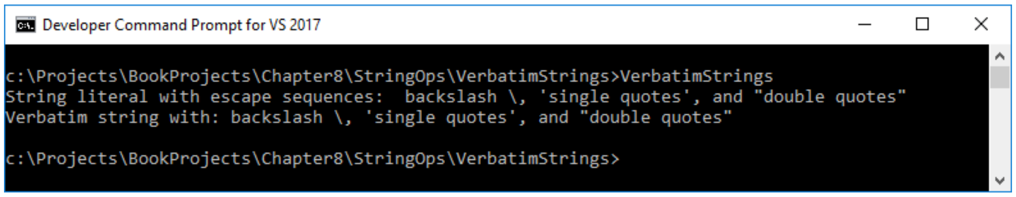
Verbatim Strings

Verbatim strings are prefaced with the '@' character and allow you to avoid the use of escape sequences normally used in string literals. You still must escape double quotes, but the escape character is the double quote character as is shown in example 8.20.

8.20 VerbatimStrings.cs

```
1   using System;
2
3   public class VerbatimStrings {
4     public static void Main(){
5       string s1 = "String literal with escape sequences:  backslash \\, \'single quotes\'," +
6                   " and \"double quotes\"";
7       string s2 = @"Verbatim string with: backslash \, 'single quotes', and ""double quotes""";
8       Console.WriteLine(s1);
9       Console.WriteLine(s2);
10    }// end Main()
11  }// end VerbatimStrings class definition
```

Referring to example 8.20 — Note that on line 7 the verbatim string begins with the '@' character and the backslash and single quote characters are embedded directly in the string with no escape character. The double quotes, on the other hand, are escaped with a double quote character. Figure 8.32 shows the results of running this program.



```
c:\Projects\BookProjects\Chapter8\StringOps\VerbatimStrings>VerbatimStrings
String literal with escape sequences:  backslash \, 'single quotes', and "double quotes"
Verbatim string with: backslash \, 'single quotes', and "double quotes"

c:\Projects\BookProjects\Chapter8\StringOps\VerbatimStrings>
```

Figure 8-32: Results of Running Example 8.20

Verbatim strings are discussed in greater detail in chapter 17, where their use to formulate complex file path strings make them quite handy.

Composite Format Strings and Interpolated Strings

You've already seen how strings can be concatenated with other strings using the concatenation operator `'+'`. You can also use composite format strings and interpolated strings to achieve the same effects in a more compact expression. Example 8.21 shows three different ways to write the same formatted string to the console.

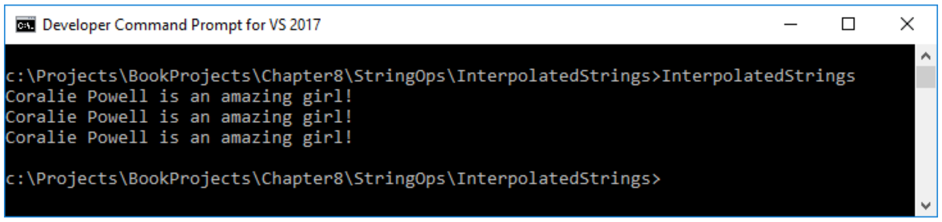
8.21 InterpolatedStrings.cs

```
1   using System;
2
3   public class InterpolatedStrings {
4     public static void Main(){
5       string firstName = "Coralie";
6       string lastName  = "Powell";
7
8       //String Concatenation
9       Console.WriteLine(firstName + " " + lastName + " is an amazing girl!");
10      //Composite String
11      Console.WriteLine("{0} {1} is an amazing girl!", firstName, lastName);
12      //Interpolated String
13      Console.WriteLine($"{firstName} {lastName} is an amazing girl!");
14
15    }// end Main()
16  }// end InterpolatedStrings class definition
```

Referring to example 8.21 — Line 9 demonstrates the use of the string concatenation operator, nothing new there. Line 11 demonstrates the use of a composite format string. Indexed placeholders {0} and {1} are used in conjunction with a list of objects, in this case firstName and lastName. Zero-based indexing is used so {0} corresponds to firstName, while {1} corresponds to lastName. Note how the indexed placeholders are part of the string and enclosed in double quotes.

Line 13 gives an example of an interpolated strings. Note that the interpolated string begins with the **'$'** character. Objects are embedded directly in the string within braces. Interpolated strings were introduced in C# 6.

Figure 8.33 shows the results of running this program.



```
Developer Command Prompt for VS 2017

c:\Projects\BookProjects\Chapter8\StringOps\InterpolatedStrings>InterpolatedStrings
Coralie Powell is an amazing girl!
Coralie Powell is an amazing girl!
Coralie Powell is an amazing girl!

c:\Projects\BookProjects\Chapter8\StringOps\InterpolatedStrings>
```

Figure 8-33: Results of Running Example 8.21

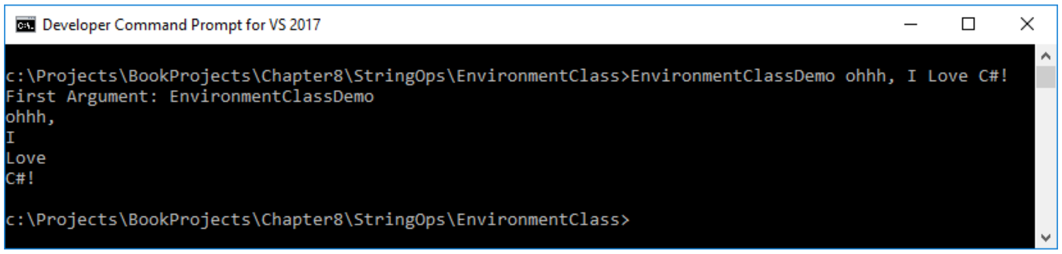
Environment Class - Extracting Command-Line Parameters

The Environment class provides quick access to a program's executable environment information. To get a good understanding of everything the Environment class has to offer, I refer you to the Microsoft's Environment class documentation page. Example 8.22 shows you how to extract command-line parameters using the Environment.GetCommandLineArgs() method.

8.22 EnvironmentClassDemo.cs

```
using System;

public class EnvironmentClassDemo {

  public static void Main(){
    string[] args = Environment.GetCommandLineArgs();
    Console.WriteLine("First Argument: " + args[0]);
    for(int i = 1; i<args.Length; i++){
      Console.WriteLine(args[i]);
    }
  }// end Main()
}// end EnvironmentClassDemo
```

Referring to example 8.22 — Note that the no argument version of the Main() method is used in this example. Figure 8-34 shows the results of running this program using the command-line arguments — Ohhh, I Love C#!.



```
Developer Command Prompt for VS 2017

c:\Projects\BookProjects\Chapter8\StringOps\EnvironmentClass>EnvironmentClassDemo ohhh, I Love C#!
First Argument: EnvironmentClassDemo
ohhh,
I
Love
C#!

c:\Projects\BookProjects\Chapter8\StringOps\EnvironmentClass>
```

Figure 8-34: Results of Running Example

Quick Review

Strings can be treated like arrays using array indexing notation to access each character in the string. Common operations on strings, like obtaining the starting index of substrings, requires an understanding of arrays.

A handful of special characters must be escaped in ordinary string literals using the backslash character '\'. Verbatim strings begin with the '@' character and will let you embed special characters within the string avoiding the need to use an escape sequence.

The '+' operator is used to concatenate strings. Favor the use of the StringBuilder.Append() method if you find you're concatenating lots of strings at runtime.

Interpolated strings begin with the '$' character and are a more convenient form of composite strings.

You can use the Environment class to get information about a program's executable environment, including command-line parameters.

The foreach Statement

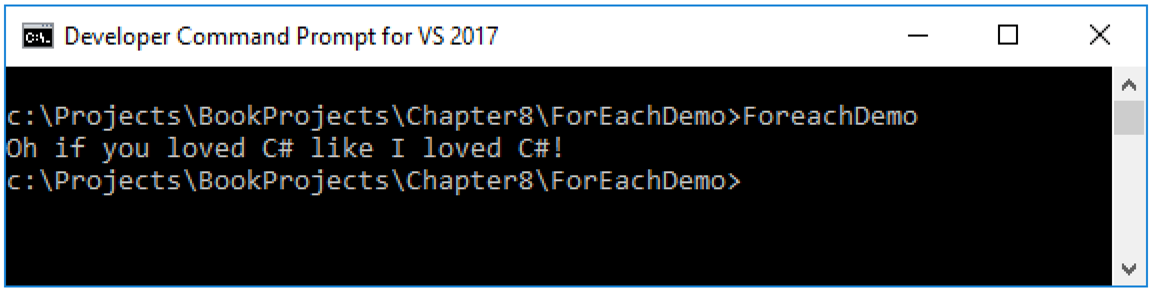
I've used the foreach statement in several examples without formal discussion, so I'd like to take the time now to correct my mistake and introduce you to your new best friend.

If you need to iterate over the elements of an array and **do not need to modify the elements**, you can do so with the foreach statement as is shown in example 8.23.

8.23 ForeachDemo.cs

```
1   using System;
2
3   public class ForeachDemo {
4     public static void Main(){
5       string[] string_array = { "Oh", "if", "you", "loved", "C#",
6                                 "like", "I", "loved", "C#!" };
7
8       foreach(string s in string_array){
9         Console.Write(s + " ");
10      }
11    }
12  }
```

Referring to example 8.23 — I declare and initialize an array of strings on line 5 and step through each element of the array with a foreach statement on line 8. This results of running this program are shown in figure 8-35.



```
c:\Projects\BookProjects\Chapter8\ForEachDemo>ForeachDemo
Oh if you loved C# like I loved C#!
c:\Projects\BookProjects\Chapter8\ForEachDemo>
```

Figure 8-35: Results of Running Example 8.23

There are a few things to keep in mind when using the foreach statement. First, you can't modify the elements of the array. If you need to step through an array and modify the elements, then you'll need to use a regular for statement and use array indexing to access each element. Finally, you can use a foreach statement to step through the elements of any object that implements IEnumerable or IEnumerable<T>. This includes arrays and collections. I cover collections in detail in chapter 14, but you'll be introduced to collections and see the foreach statement in action in the following section on LINQ.

Introduction To Language Integrated Query (LINQ)

In this section, I offer a brief introduction to LINQ and show you how you can use it to query arrays. I'll cover LINQ to XML in chapter 14, and LINQ to SQL in chapter 20.

Language Integrated Query, or LINQ, has been available since C# 3.0 and Microsoft.NET 3.5. LINQ provides general purpose constructs, built into the language, that facilitate queries over objects and data sources that implement the IEnumerable<T> or IQueryable<T> interfaces.

LINQ deals with *sequences* and *elements*. Any object that implements IEnumerable<T> is a sequence. If you refer back to figure 8.3 you'll notice that System.Array implements IEnumerable, not IEnumerable<T>, but if you examine the documentation for System.Array you'll see that it has an extension method called AsQueryable(IEnumerable), which returns an IQueryable<T> object that queries the array in terms of methods defined by IEnumerable. Let's look at an example.

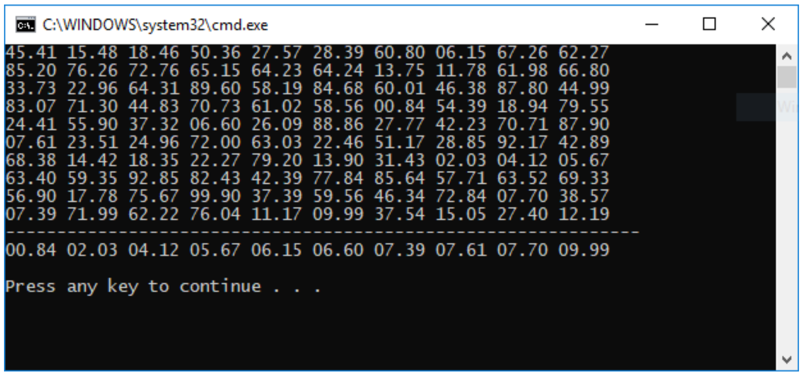
Suppose you have an array of 100 random double values and you'd like to search the array for all elements based on some criteria. You can use LINQ to perform the query as shown in example 8.24.

8.24 LinqDemo.cs

```
1   using System;
2   using System.Collections.Generic;
3   using System.Linq;
4
5   public class LinqDemo {
6     static void Main(string[] args){
7       // Declare and initialize an array of 100 doubles
8       double[] double_array = new double[100];
9       Random random = new Random();
10
11      // Initialize each array element with a random double value
12      for (int i = 0; i < double_array.GetLength(0); i++){
13        double_array[i] = random.NextDouble() * 100;
14      }
15
16      // Print the array of doubles as a nicely formatted grid
17      for (int i = 1; i <= double_array.GetLength(0); i++){
18        Console.Write(double_array[i - 1].ToString("00.00") + " ");
19      if ((i % 10) == 0){
20          Console.WriteLine();
21        }
22      }
23
24      Console.WriteLine("----------------------------------------------------------");
25
26      // Create the query using LINQ. Sort results in ascending order
27      IEnumerable<double> query = double_array.Where(n => n <= 10).OrderBy(n => n);
28
29      // Write the results of the query to the console
30      foreach (double d in query){
31        Console.Write(d.ToString("00.00") + " ");
32      }
33
34      Console.WriteLine("\n");
35
36    } // end Main
37  } // end class
```

Referring to example 8.24 — In this program, I declare an array of doubles and initialize each element to a random double value with the help of the System.Random class. I then write the array of doubles to the console as a grid with 10 elements per line. On line 27, I create a LINQ query that returns all the elements in the array that are less than 10 sorted in ascending order. I then use the query in the foreach statement that begins on line 30 to write the results to the console.

Note that the Where() and OrderBy() query operators are extension methods. You won't find them defined for the System.Array class in the documentation, rather, you'll see an extension method named AsQueryable(IEnumerable), which converts the array into an IQueryable object that provides the Where() and OrderBy() methods. Figure 8-36 shows the results of running this program.



```
45.41 15.48 18.46 50.36 27.57 28.39 60.80 06.15 67.26 62.27
85.20 76.26 72.76 65.15 64.23 64.24 13.75 11.78 61.98 66.80
33.73 22.96 64.31 89.60 58.19 84.68 60.01 46.38 87.80 44.99
83.07 71.30 44.83 70.73 61.02 58.56 00.84 54.39 18.94 79.55
24.41 55.90 37.32 06.60 26.09 88.86 27.77 42.23 70.71 87.90
07.61 23.51 24.96 72.00 63.03 22.46 51.17 28.85 92.17 42.89
68.38 14.42 18.35 22.27 79.20 13.90 31.43 02.03 04.12 05.67
63.40 59.35 92.85 82.43 42.39 77.84 85.64 57.71 63.52 69.33
56.90 17.78 75.67 99.90 37.39 59.56 46.34 72.84 07.70 38.57
07.39 71.99 62.22 76.04 11.17 09.99 37.54 15.05 27.40 12.19
---------------------------------------------------------------
00.84 02.03 04.12 05.67 06.15 06.60 07.39 07.61 07.70 09.99

Press any key to continue . . .
```

Figure 8-36: Results of Running Example 8.24

Referring to figure 8-36 — Since double_array is initialized with random double values, you'll get different results every time you run the program.

I could have been more explicit with the query definition on line 27 as the following code snippet illustrates:

```
IEnumerable<double> query = double_array
                           .AsQueryable()
                           .Where(n => n <= 10)
                           .OrderBy(n => n);
```

In this case, I'm explicitly calling the AsQueryable() method on double_array, the result of which is an object that implements the IQueryable interface, which supplies the Where() and OrderBy() methods.

LINQ Operators and Lambda Expressions

In example 8.24 above, the Where() and OrderBy() methods are query operators. A query operator is a method that transforms a sequence. The arguments to each of these operators are lambda expressions. Lambda expressions takes a bit of getting used to, but with time you'll get the swing of it.

The '=>' is referred to as the lambda operator. The lambda expression "n => n <= 10", which is passed to the Where() method on line 27 can be read "The element n that is less than or equal to 10". The results of the call to Where(n => n <= 10) is a sequence of elements that satisfy the condition n <= 10.

Note that query operators can be chained together. Looking again at the LINQ query given on line 27:

```
IEnumerable<double> query = double_array.Where(n => n <= 10).OrderBy(n => n);
```

The call to the Where() query operator returns a sequence of elements that satisfy the condition specified by the lambda expression: "n => n <=10". The OrderBy() query operator orders the sequence in ascending order. The lambda expression "n => n" is pretty cryptic in this case because numeric values have a natural ordering. If, on the other hand, you were dealing with a sequence of Person objects, you could order them according to this lambda expression: "n => n.Age", assuming Persons objects had a public property called Age. To order a sequence by descending values use the OrderByDescending() query operator.

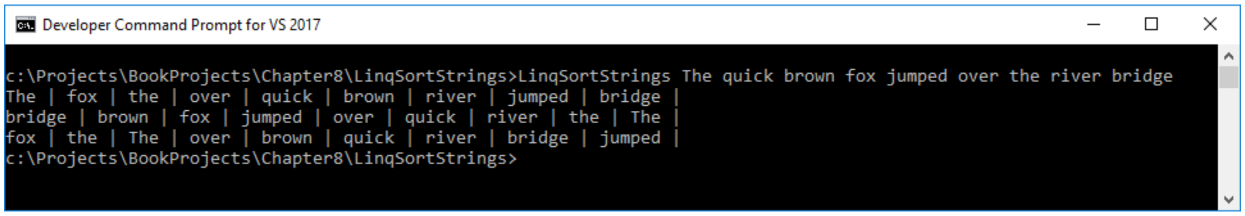
Let's look at another example. Say you wanted to order a collection of strings by length. Here's how you'd do it with LINQ.

8.25 LinqSortStrings.cs

```
using System;
using System.Linq;

```

```
4   public class LinqSortStrings {
5     public static void Main(string[] args){
6       if(args.Length > 0){
7         // Sort strings by Length in ascending order
8         var query = args.OrderBy(s => s.Length);
9
10        foreach(string s in query){
11          Console.Write(s + " | ");
12        }
13
14        Console.WriteLine();
15
16        // Sort strings alphabetically
17        query = args.OrderBy(s => s);
18
19        foreach(string s in query){
20          Console.Write(s + " | ");
21        }
22
23        Console.WriteLine();
24
25        // The sort the sorted strings by Length
26        foreach(string s in query.OrderBy(s => s.Length)){
27          Console.Write(s + " | ");
28        }
29      }
30    }
31  }
```

Referring to example 8.25 — Note first on line 8 that instead of explicitly stating the type IEnuerable<string>, I used the var keyword to declare the query reference. This is a great example of an appropriate use of var. Also on line 8, I order the string elements in the args array by the String.Length property as specified by the lambda expression: "s => s.Length". In other words, I'm sorting the strings by length. Next, on line 17, I reuse the query reference and create a new LINQ query to sort the strings alphabetically in ascending order. Strings are naturally ordered alphabetically. Finally, on line 26, I order the alphabetically sorted strings by length. Figure 8-37 shows the results of running this program.



```
c:\Projects\BookProjects\Chapter8\LinqSortStrings>LinqSortStrings The quick brown fox jumped over the river bridge
The | fox | the | over | quick | brown | river | jumped | bridge |
bridge | brown | fox | jumped | over | quick | river | the | The |
fox | the | The | over | brown | quick | river | bridge | jumped |
c:\Projects\BookProjects\Chapter8\LinqSortStrings>
```

Figure 8-37: Sorting Strings using LINQ and Lambda Expressions

In this section I presented a brief introduction to Language Integrated Query (LINQ) and lambda expressions. You'll learn more about LINQ in later chapters. A deep treatment of LINQ is beyond the scope of the book. If you're interested in learning more about this powerful feature of C#, I recommend you consult one of the references listed at the end of the chapter.

Quick Review

Language Integrated Query (LINQ) lets you write type-safe, structured queries against arrays and collections that implement the IEnumerable or IEnumerable<T> interfaces. LINQ uses query operators and lambda expressions that operate on sequences and elements.

Summary

C# array types have special functionality because of their inheritance hierarchy. C# array types directly inherit functionality from the System.Array class and implement the ICloneable, IList, ICollection, and IEnumerable interfaces. Arrays are also serializable.

Single-dimensional arrays have one dimension — length. You can get an array's length by calling the GetLength() method with an integer argument that indicates the dimension in which you are interested. You can also get the length of a single dimensional array by accessing its *Length* property. Arrays can have elements of either value or reference types. An array type is created by specifying the type name of array elements followed by one set of brackets [ ]. Use System.Array class methods and properties to get information about an array.

Each element of an array is accessed via an index value contained within a set of brackets. Value-type element values can be directly assigned to array elements. When an array of value types is created, each element is initialized to the types default value. Each element of an array of references is initialized to null. Each object that a reference element points to must either already exist or be created during program execution.

C# supports two kinds of multidimensional arrays: rectangular and ragged. A rectangular array is a multidimensional array whose shape is fixed based on the length of each dimension or rank. All of a rectangular array's dimensions must be specified when the array object is created.

A ragged array is an array of arrays. Ragged arrays can be any number of dimensions but the last, or rightmost, dimension is omitted from the array creation expression. Each rightmost array object must then be created during program execution, introducing the possibility that the array's dimensions may differ in length.

Use the built-in methods and properties of the System.Array class to perform certain array manipulations such as sorting.

Strings can be treated like arrays using array indexing notation to access each character in the string. Common operations on strings, like obtaining the starting index of substrings, requires an understanding of arrays.

A handful of special characters must be escaped in ordinary string literals using the backslash character '\'. Verbatim strings begin with the '@' character and will let you embed special characters within the string avoiding the need to use an escape sequence.

The '+' operator is used to concatenate strings. Favor the use of the StringBuilder.Append() method is you find you're concatenating lots of strings at runtime.

Interpolated strings begin with the '$' character and are a more convenient form of composite strings.

You can use the Environment class to get information about a program's executable environment, including command-line parameters.

Language Integrated Query (LINQ) lets you write type-safe, structured queries against arrays and collections that implement the IEnumerable or IEnumerable<T> interfaces. LINQ uses query operators and lambda expressions that operate on sequences and elements.

Skill-Building Exercises

1. **Further Research:** Study the System.Array class and the interfaces it implements to better familiarize yourself with the functionality it provides.

2. **Further Research:** Conduct a web search for different applications for single and multidimensional

arrays.

3. **Single-Dimensional Arrays:** Write a program that lets you create a single-dimensional array of integers of different sizes at program runtime using command-line inputs.

4. **Single-Dimensional Arrays:** Write a program that reverses the order of text entered on the command line. This will require the use of the Main() method's string array.

5. **Further Research:** Conduct a web search on different sorting algorithms and how arrays are used to implement these algorithms. Also, there are several good sources of information regarding sorting algorithms listed in the references section of this chapter.

6. **Multidimensional Arrays:** Modify example 8.9 so that it creates two-dimensional arrays of characters. Initialize each element with the character 'c'. Run the program several times to create character arrays of different sizes.

7. **Multidimensional Arrays:** Modify example 8.9 again so that the character array is initialized to the value of the first character read from the command line. **Hint:** Refer to example 8.13 to see how to access the first character of a string.

8. Environment Class: Look up the Environment class on Microsoft Docs. Research its properties and methods to get a feel for what it can help you do in future programs.

Suggested Projects

1. **Matrix Multiplication:** Given two matrices A_{ij} and B_{jk}, the product C_{ik} can be calculated with the following equation:

$$C_{ik} = \sum_{j=1}^{n} A_{ij}B_{jk}$$

Write a program that multiplies the following matrices together and stores the results in a new matrix. Print the resulting matrix values to the console.

$$\begin{bmatrix} 2 & 3 \\ 3 & 4 \\ 4 & 5 \end{bmatrix} \begin{bmatrix} 2 & 3 & 4 \\ 3 & 4 & 5 \end{bmatrix}$$

2. **Modify Histogram Program:** Modify the histogram program given in example 8.8 so that it counts the occurrence of the digits 0 through 9 and the punctuation marks period '.', comma ',', question mark '?', colon ':', and semicolon ';'.

3. **Computer Simulator:** You are a C# developer with a high-tech firm doing contract work for the Department of Defense. Your company has won the proposal to develop a proof-of-concept model for an Encrypted Instruction Set Computer System Mark 1 (EISCS Mk1). Your job is to simulate the operation of the EISCS Mk1 with a C# application.

   Supporting Information: The only language a computer understands is its machine-language instruction set. The EISCS Mk1 is no different. The EISCS machine language instruction set will consist of a four-digit integer with the two most significant digits being the *operation code* (*opcode*) and the two least significant digits being the *operand*. For example, consider the following instruction:

$$\underbrace{\underline{11}}_{\text{opcode}}\underbrace{\underline{33}}_{\text{operand}}$$

   The number 11 represents the opcode and the number 33 represents the operand. The following table lists and describes each EISCS machine instruction.

Opcode	Mnemonic	Description
Input/Output Operations		
10	READ	Reads an integer value from the console and stores it in memory location identified by the operand.
11	WRITE	Writes the integer value stored in memory location operand to the console.
Load/Store Operations		
20	LOAD	Loads the integer value stored at memory location operand into the accumulator.
21	STORE	Stores the integer value residing in the accumulator into memory location operand.
Arithmetic Operations		
30	ADD	Adds the integer value located in memory location operand to the value stored in the accumulator and leaves the result in the accumulator.
31	SUB	Subtracts the integer value located in memory location operand from the value stored in the accumulator and leaves the result in the accumulator.
32	MUL	Multiplies the integer value located in memory location operand by the value stored in the accumulator and leaves the result in the accumulator.
33	DIV	Divides the integer value stored in the accumulator by the value located in memory location operand.
Control and Transfer Operations		

Table 8-4: EISCS Machine Instructions

Opcode	Mnemonic	Description
40	BRANCH	Unconditional jump to memory location operand.
41	BRANCH_NEG	If accumulator value is less than zero jump to memory location operand.
42	BRANCH_ZERO	If accumulator value is zero then jump to memory location operand.
43	HALT	Stop program execution.

Table 8-4: EISCS Machine Instructions

Sample Program: Using the instruction set given in table 8-3, you can write simple programs that will run on the EISCS Mk1 computer simulator. The following sample program reads two numbers from the input, multiplies them together, and writes the results to the console.

Memory Location	Instruction / Contents	Action
00	1007	Read integer into memory location 07
01	1008	Read integer into memory location 08
02	2007	Load contents of memory location 07 into accumulator
03	3208	Multiply value located in memory location 08 by value stored in accumulator. Leave result in accumulator
04	2109	Store value currently in accumulator to memory location 09
05	1109	Write the value located in memory location 09 to the console
06	4010	Jump to memory location 10
07		
08		
09		
10	4300	Halt program

Basic Operation: This section discusses several aspects of the EISCS computer simulation operation to assist you in completing the project.

Memory: The machine language instructions that constitute an EISCS program must be loaded into memory before the program can be executed by the simulator. Represent the computer simulator's memory as an array of integers 100 elements long.

Instruction Decoding: Instructions are fetched one at a time from memory and decoded into opcodes and operands before being executed. The following code sample demonstrates one possible decoding scheme:
Hints:

```
instruction = memory[program_counter++];
operation_code = instruction / 100;
operand = instruction % 100;
```

- Use switch/case structure to implement the instruction execution logic.
- You may either hard code sample programs in your simulator or allow a user to enter a program into memory via the console.
- Use an array of 100 integers to represent memory.

Self-Test Questions

1. Arrays are contiguously allocated memory elements of homogeneous data types. Explain in your own words what this means.

2. What's the difference between arrays of value types vs. arrays of reference types?

3. C# array types directly inherit functionality from what class?

4. How do you determine the length of an array?

5. (True/False) An array can be resized after it has been created.

6. (True/False) One or more of the dimensions of a rectangular array can be left unspecified upon array object creation.

7. Ragged arrays are _______________ of _______________.

8. When a ragged array is created, which dimensions are optional and which dimensions are mandatory?

9. What is meant by the term "ragged array"?

10. What's the purpose of the Main() method's string array?

References

ECMA-335 Common Language Infrastructure (CLI), 6th Edition, June 2012 http://www.ecma-international.org/publications/standards/Ecma-335.htm

ECMA-334 C# Language Specification, 4th Edition, June 2006 http://www.ecma-international.org/publications/standards/Ecma-334.htm

Microsoft Developer Network (MSDN) http://www.msdn.com

Microsoft Docs https://docs.microsoft.com

Donald E. Knuth. *The Art of Computer Programming. Volume 3: Sorting and Searching*, Second Edition. Addison-Wesley. Reading Massachusetts. ISBN: 0-201-89685-0

Joseph Albahari, et. al., *LINQ Pocket Reference*, O'Reilly Media, Inc., Sebastapol, CA. ISBN: 978-0-596-51924-7

Lambda Expressions, Microsoft Docs, https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/statements-expressions-operators/lambda-expressions

Notes