Chapter 4

Contax T

Rain Walkers

Arrays

Learning Objectives

- *Understand the relationship between arrays and collections*
- *Understand the functionality provided by the IList interface*
- *Declare and use single-dimensional arrays*
- *Declare and use multi-dimensional arrays*
- *Understand how arrays are represented in memory*
- *Describe the functionality provided by System.Array class*
- *Use the System.Array class to sort the contents of an array*
- *Use the System.Array class to reverse the contents of an array*

Introduction

Arrays are closely related to collections in many ways. They serve as the underlying foundational data structure for several collection types including ArrayList, List<T>, and Hashtable, and fill auxiliary roles in others. Arrays also implement the ICollection interface and partially implement the IList interface. Thus, cultivating a thorough working knowledge of arrays leads to a better understanding of collections in general. In fact, it's often desirable to convert a collection onto an array for streamlined manipulation and many collection types provide a ToArray() method for just this purpose.

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. 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.

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 4-1 illustrates these concepts.

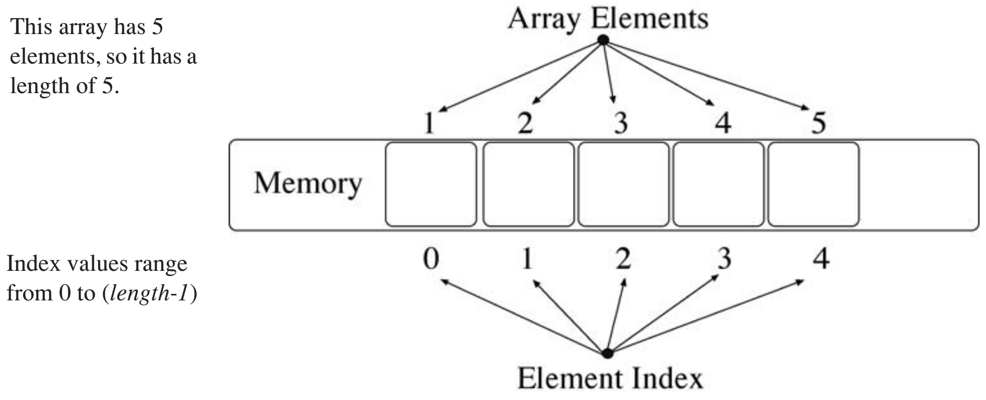

This array has 5 elements, so it has a length of 5.
Array Elements
1
2
3
4
5
Memory
0
1
2
3
4
Index values range from 0 to (*length-1*)
Element Index

Figure 4-1: Array Elements are Contiguous and Homogeneous

Figure 4-1 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 4-1 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]*).

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 4-2 illustrates this concept through the use of the array declaration and allocation syntax.

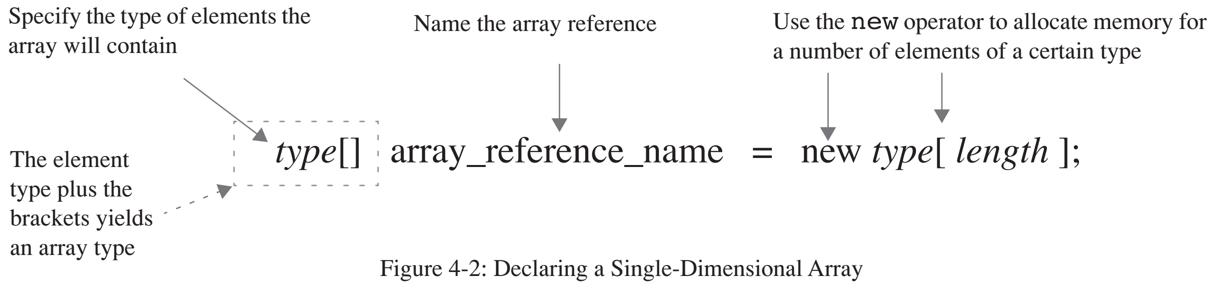

Specify the type of elements the array will contain
Name the array reference
Use the new operator to allocate memory for a number of elements of a certain type
The element type plus the brackets yields an array type
type[] array_reference_name = new type[length];

Figure 4-2: Declaring a Single-Dimensional Array

Figure 4-2 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 API, 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. You will add a set of brackets for each additional *dimension* or *rank* you want the 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 4-2 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 code 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];
```

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 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 4-2. The array you create automatically inherits the functionality provided by the System.Array class, which itself extends from the System.Object class. Figure 4-3 shows the UML inheritance diagram for an array type.

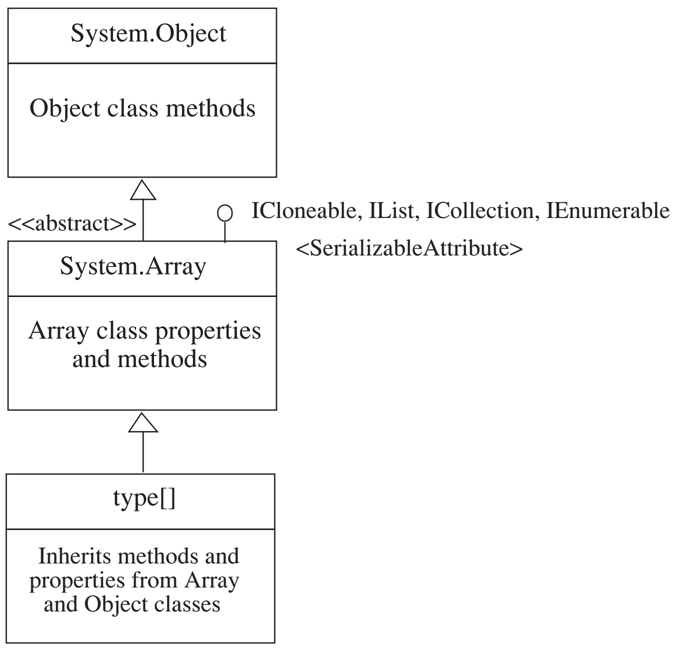

System.Object
Object class methods
<<abstract>>
ICloneable, IList, ICollection, IEnumerable
<SerializableAttribute>
System.Array
Array class properties and methods
type[]
Inherits methods and properties from Array and Object classes

Figure 4-3: Array-Type Inheritance Hierarchy

Referring to Figure 4-3 — 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 to be accessed via the Array class itself. 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 MSDN 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

The Table 4-1 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 re-sized 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);`
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 4-1: C# Array Properties

Quick Review

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

Creating And Using Single-Dimensional Arrays

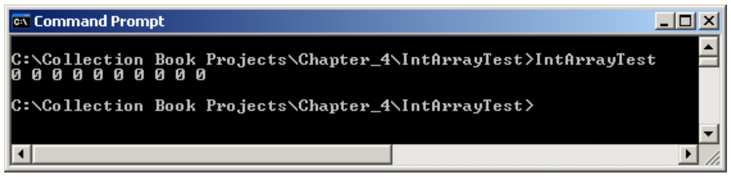
This section shows you how to declare, create, and use single-dimensional arrays of both value types and reference types. Once you know how a single-dimensional array works, you can easily apply the concepts to multidimensional arrays.

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 4.1 shows an array of integers being declared, created, and utilized in a short program. Figure 4-4 shows the results of running this program.

4.1 IntArrayTest.cs

```
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     }
11   }
```



```
C:\Collection Book Projects\Chapter_4\IntArrayTest>IntArrayTest
0 0 0 0 0 0 0 0 0 0

C:\Collection Book Projects\Chapter_4\IntArrayTest>
```

Figure 4-4: Results of Running Example 4.1

Referring to Example 4.1 — 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. As you can see from looking at Figure 4-4, 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 4-5 shows how the integer array int_array declared and created in Example 4.1 is represented in memory. The name of the array, int_array, is a reference to an object in memory of type *System.Int32[]*. 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 4.1 by assigning some values to the int_array elements. Example 4.2 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.

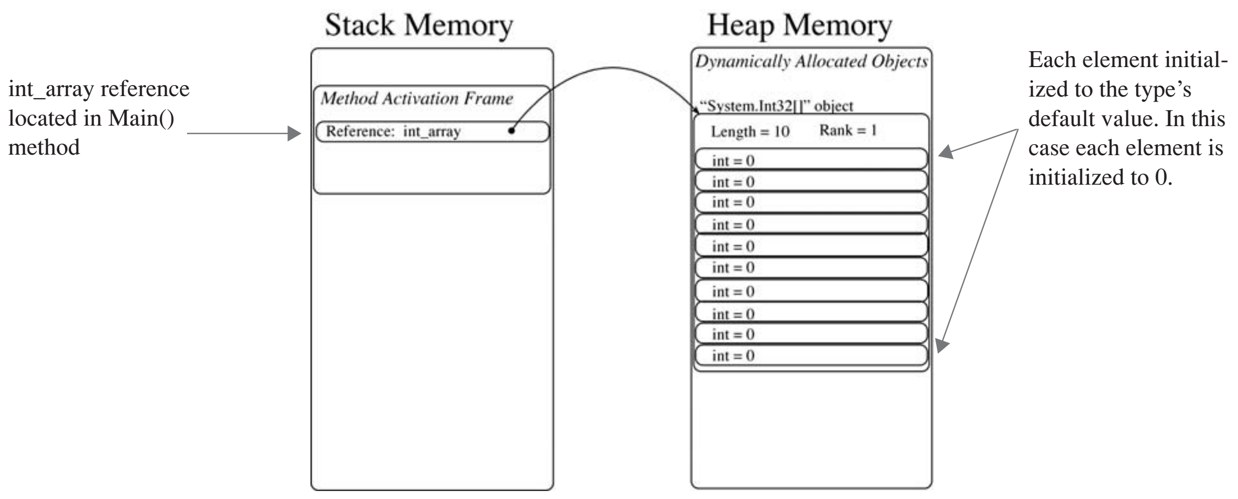
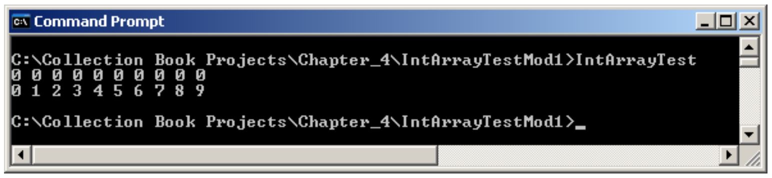


Figure 4-5: Memory Representation of Value Type Array int_array Showing Default Initialization

4.2 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 4.2 — 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 4-6 shows the results of running this program. Figure 4-7 shows the memory representation of int_array after its elements have been assigned their new values.



```
C:\Collection Book Projects\Chapter_4\IntArrayTestMod1>IntArrayTest
0 0 0 0 0 0 0 0 0 0
0 1 2 3 4 5 6 7 8 9

C:\Collection Book Projects\Chapter_4\IntArrayTestMod1>_
```

Figure 4-6: Results of Running Example 4.2

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

Study the code shown in Example 4.3, paying particular attention to lines 6 through 10.

4.3 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
```

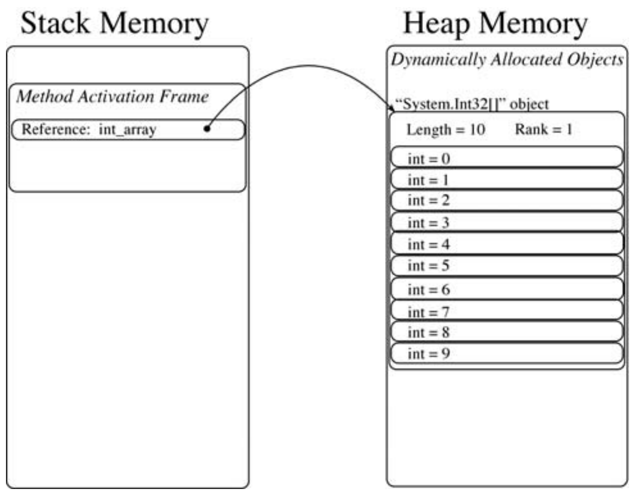
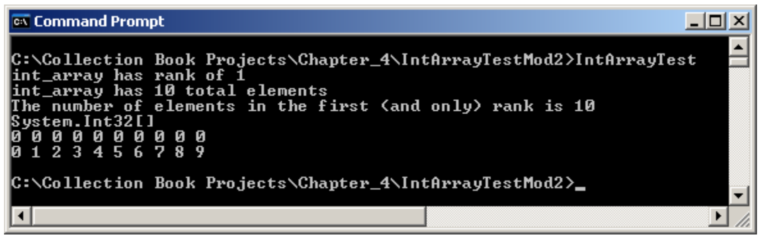


Figure 4-7: Element Values of int_array After Initialization Performed by Second `for` Loop

```
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 4.3 — 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 4-8 gives the results of running this program.



```
C:\Collection Book Projects\Chapter_4\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:\Collection Book Projects\Chapter_4\IntArrayTestMod2>_
```

Figure 4-8: Results of Running Example 4.3

Creating Single-Dimensional Arrays Using Array Literal Values

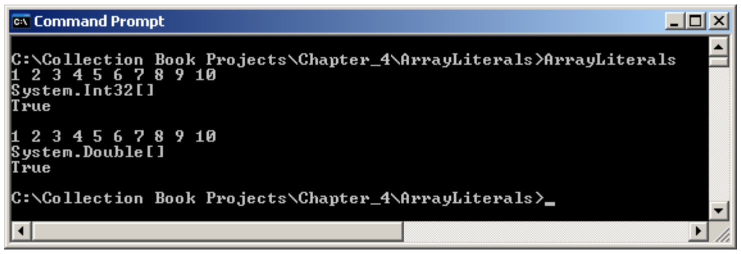
Up to this point you have seen how memory for an array can be allocated using the `new` operator. Another way to allocate memory for an array and initialize its elements at the same time is to specify the contents of the array using *array literal* values. The length of the array is determined by the number of literal values appearing in the declaration. Example 4.4 shows two arrays being declared and created using literal values.

4.4 ArrayLiterals.cs

```
1    using System;
2
3    public class ArrayLiterals {
4      static void Main(){
5        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 4.4 — 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 4-9 shows the results of running this program.



```
C:\Collection Book Projects\Chapter_4\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:\Collection Book Projects\Chapter_4\ArrayLiterals>_
```

Figure 4-9: Results of Running Example 4.4

Differences Between Arrays Of Value Types And Arrays Of Reference Types

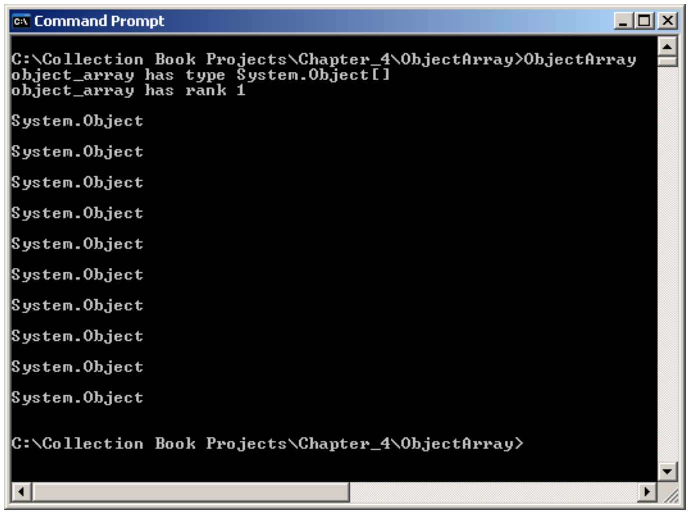
The key difference between arrays of value types and arrays of reference types is that value-type values can be directly assigned to value-type array elements. The same is not true for reference type elements. In an array of reference types, each element is a reference to an object in memory. When you 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 4.5 gives the code for a short program that creates and uses an array of Objects.

4.5 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());
    Console.WriteLine("object_array has rank " + object_array.Rank);
    Console.WriteLine();

    object_array[0] = new Object();
    Console.WriteLine(object_array[0].GetType());
    Console.WriteLine();

    object_array[1] = new Object();
    Console.WriteLine(object_array[1].GetType());
    Console.WriteLine();

    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 4-10 shows the results of running this program.



```
C:\Collection Book Projects\Chapter_4\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:\Collection Book Projects\Chapter_4\ObjectArray>
```

Figure 4-10: Results of Running Example 4.5

Referring to Example 4.5 — 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 4-11.

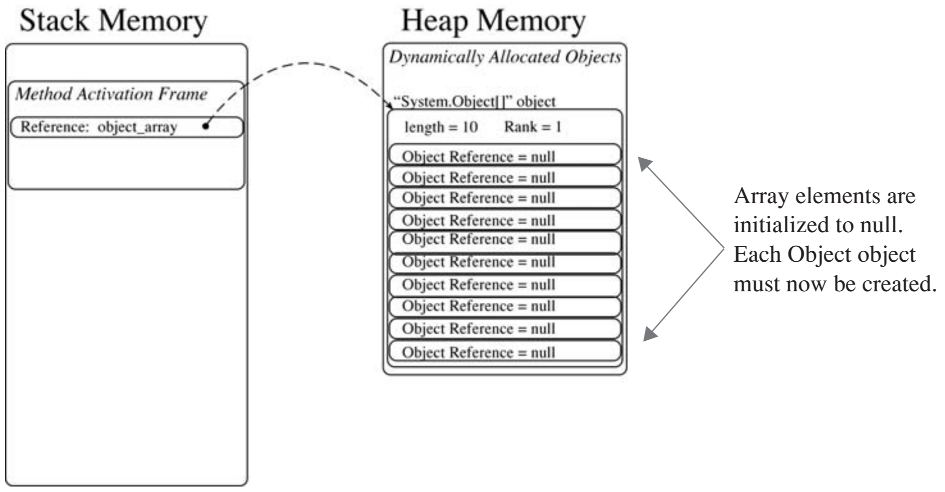


Figure 4-11: State of Affairs After Line 5 of Example 4.5 Executes

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 4-12. 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 4-13. The `for` statement on line 18 creates the remaining Object objects and assigns their memory locations to the remaining object_array reference elements. Figure 4-14 shows the memory picture after the `for` statement completes execution.

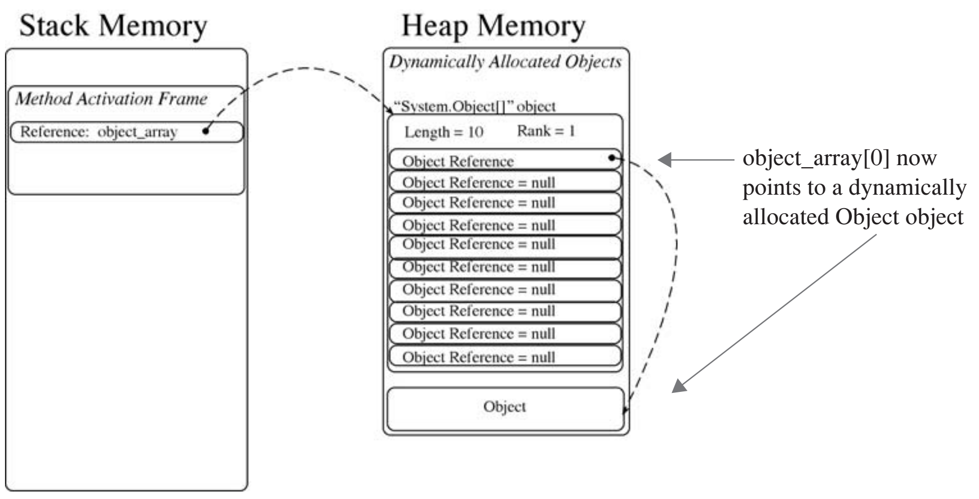


Figure 4-12: State of Affairs After Line 10 of Example 4.5 Executes.

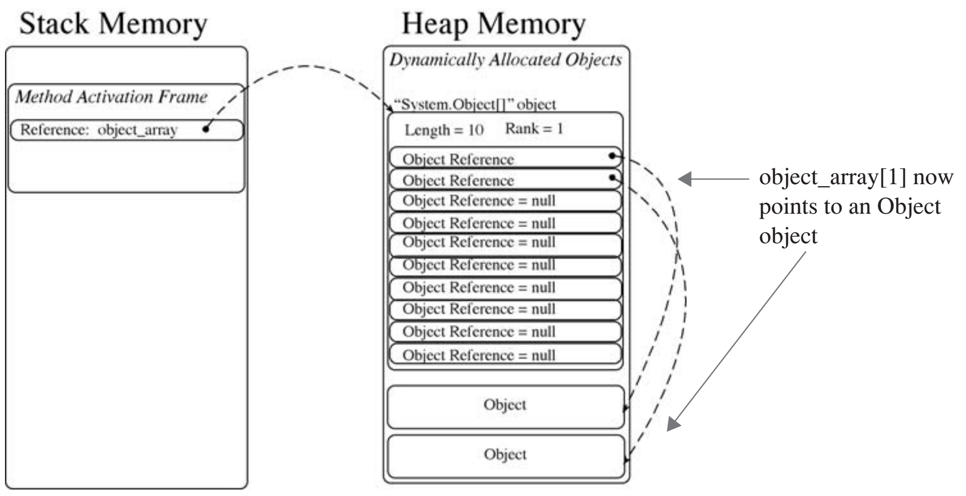


Figure 4-13: State of Affairs After Line 14 of Example 4.5 Executes

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

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 4.6 shows how such an array might be utilized.

4.6 MessageArray.cs

```
using System;

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

```

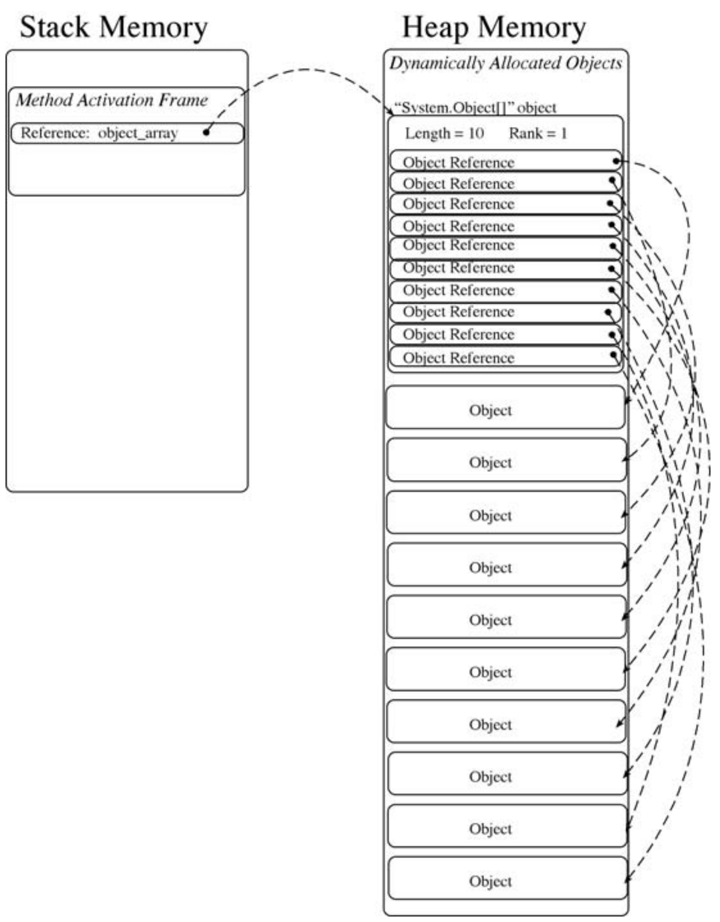
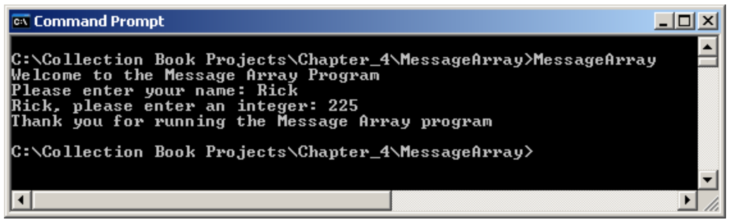


Figure 4-14: Final State of Affairs: All object_array Elements Point to an Object object

```
8        String[] messages = {"Welcome to the Message Array Program",
9                     "Please enter your name: ",
10                    ", please enter an integer: ",
11                    "You did not enter an integer!",
12                    "Thank you for running the Message Array program"};
13
14       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 4.6 — 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 4-15 shows the results of running this program.



```
C:\Collection Book Projects\Chapter_4\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:\Collection Book Projects\Chapter_4\MessageArray>
```

Figure 4-15: Results of Running Example 4.6

Calculating Averages

The program given in Example 4.7 calculates class grade averages.

4.7 Average.cs

```
1    using System;
2
3    public class Average {
4      static void Main(){
5          double[] grades      = null;
6          double   total       = 0;
7          double   average     = 0;
8          int      grade_count = 0;
9
10         Console.WriteLine("Welcome to Grade Averager");
11         Console.Write("Please enter the number of grades to enter: ");
12         try{
13             grade_count = Int32.Parse(Console.ReadLine());
14           } catch(FormatException) { Console.WriteLine("You did not enter a number!"); }
15
16          if(grade_count > 0){
17              grades = new double[grade_count];
18                 for(int i = 0; i < grade_count; i++){
19                    Console.Write("Enter grade " + (i+1) + ": ");
20                      try{
21                         grades[i] = Double.Parse(Console.ReadLine());
22                      } catch(FormatException) { Console.WriteLine("You did not enter a number!"); }
23                 } //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
```

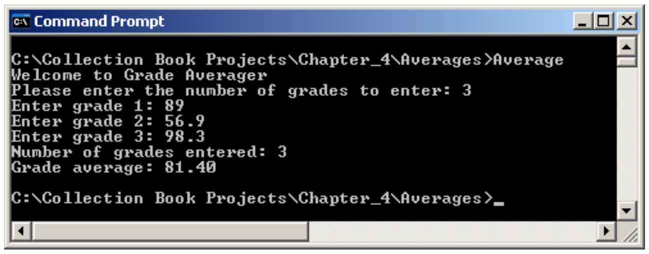
Referring to Example 4.7 — an array reference of doubles named grades is declared on line 5 and initialized to null. On lines 6 through 8, several other program variables are declared and initialized.

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 38 to properly format the double value contained in the average variable. Figure 4-16 shows the results of running this program

Histogram: Letter Frequency Counter

Letter frequency counting is an important part of deciphering messages encrypted using monalphabetic substitution. Example 4.8 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.



```
C:\Collection Book Projects\Chapter_4\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:\Collection Book Projects\Chapter_4\Averages>_
```

Figure 4-16: Results of Running Example 4.7

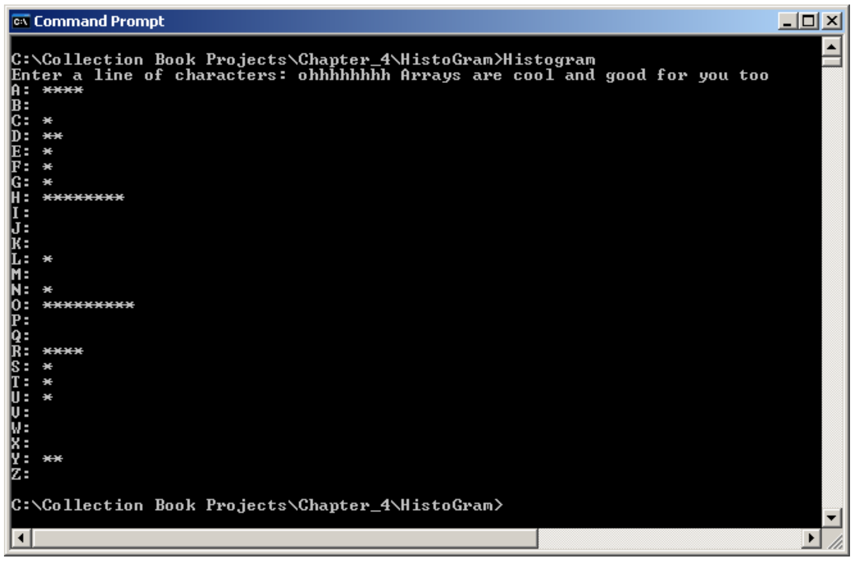
4.8 Histogram.cs

```
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;
65          case 'X': letter_frequencies[X]++;
66                    break;
67          case 'Y': letter_frequencies[Y]++;
68                    break;
69          case 'Z': letter_frequencies[Z]++;
70                    break;
71          default : break;
72         } //end switch
73       } //end for
74
75     for(int i = 0; i < letter_frequencies.Length; i++){
76        Console.Write((char)(i + 65) + ": ");
77        for(int j = 0; j < letter_frequencies[i]; j++){
78         Console.Write('*');
79        } //end for
80        Console.WriteLine();
81     } //end for
82
83    } //end if
84   } // end main
85  } // end Histogram class definition
```

Referring to Example 4.8 — 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 19 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 4-17 gives the results of running this program with a sample line of text.



```
C:\Collection Book Projects\Chapter_4\HistoGram>Histogram
Enter a line of characters: ohhhhhhhh Arrays are cool and good for you too
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:\Collection Book Projects\Chapter_4\HistoGram>
```

Figure 4-17: Results of Running Example 4.8

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 4-18 gives the rectangular array declaration syntax for a two-dimensional array.

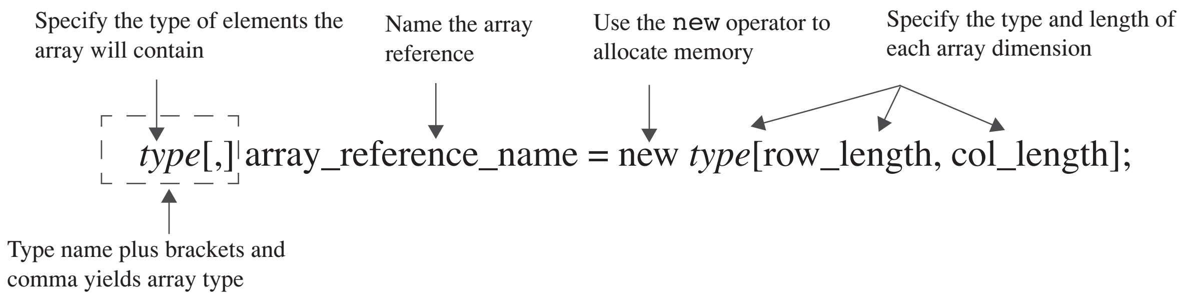

Specify the type of elements the array will contain
Name the array reference
Use the new operator to allocate memory
Specify the type and length of each array dimension
type[,] array_reference_name = new type[row_length, col_length];
Type name plus brackets and comma yields array type

Figure 4-18: Rectangular Array Declaration Syntax

Referring to Figure 4-18 — 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 4-19. 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 4-19 also includes a few more examples of two-dimensional array element access. Example 4.9 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.

4.9 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();

```

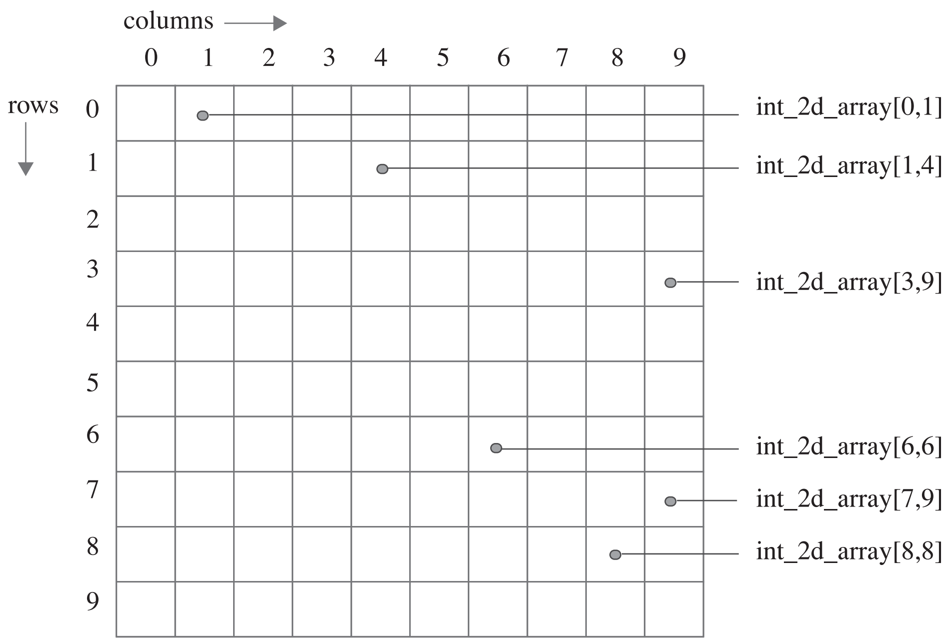
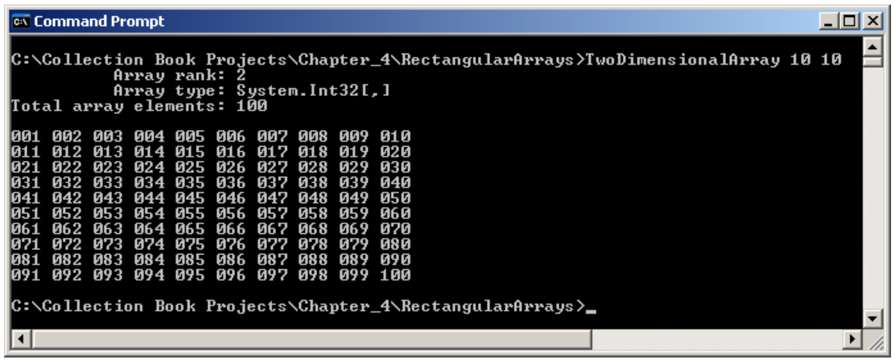


Figure 4-19: Accessing Two-Dimensional Array Elements

```
16        for(int i = 0, element = 1; i<int_2d_array.GetLength(0); i++){
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 4.9 — 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 4-20 gives the results of running this program.



```
C:\Collection Book Projects\Chapter_4\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:\Collection Book Projects\Chapter_4\RectangularArrays>_
```

Figure 4-20: Results of Running Example 4.9

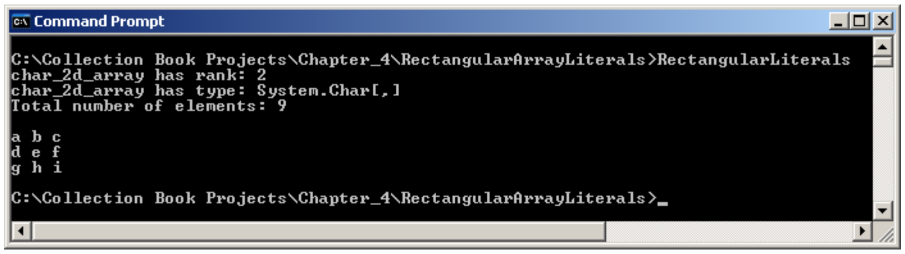
Initializing Rectangular Arrays With Array Literals

Rectangular arrays can be initialized using literal values in an array initializer expression. Study the code offered in Example 4.10.

4.10 RectangularLiterals.cs

```
1    using System;
2
3    public class RectangularLiterals {
4      static void Main(){
5        char[,] char_2d_array = {{'a', 'b', 'c'},
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 4.10 — 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 4-21 shows the results of running this program.



```
C:\Collection Book Projects\Chapter_4\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:\Collection Book Projects\Chapter_4\RectangularArrayLiterals>_
```

Figure 4-21: Results of Running Example 4.10

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. Figure 4-22 shows the ragged array declaration syntax for a two-dimensional ragged array. Example 4.11 gives a short program showing the use of a ragged array.

4.11 Ragged2dArray.cs

```
using System;

public class Ragged2dArray {
  static void Main(){
    int[][] ragged_2d_array = new int[10][];

    Console.WriteLine("ragged_2d_array has rank: " + ragged_2d_array.Rank);
    Console.WriteLine("ragged_2d_array has type: " + ragged_2d_array.GetType());
    Console.WriteLine("Total number of elements: " + ragged_2d_array.Length);
    Console.WriteLine();

    for(int i = 0; i<ragged_2d_array.GetLength(0); i++){
      ragged_2d_array[i] = new int[i+1];
    }
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    }
```

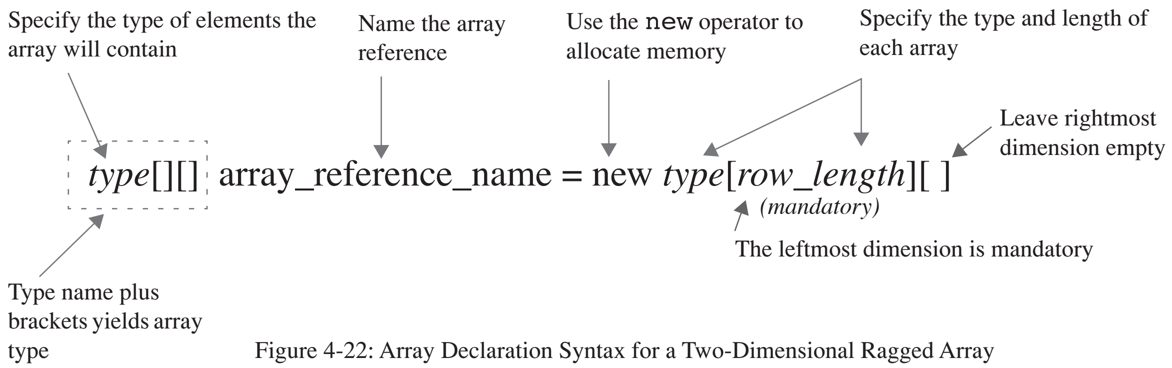
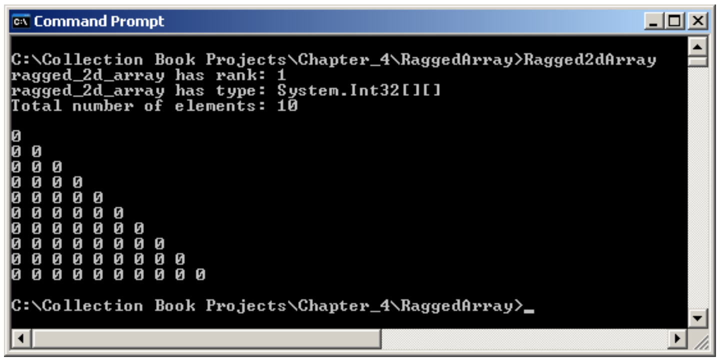

Specify the type of elements the array will contain
Name the array reference
Use the new operator to allocate memory
Specify the type and length of each array
Leave rightmost dimension empty
type[][] array_reference_name = new type[row_length][ ]
(mandatory)
The leftmost dimension is mandatory
Type name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

Figure 4-22: Array Declaration Syntax for a Two-Dimensional Ragged Array

Referring to Example 4.11 — 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 4-23 shows the results of running this program.



```
C:\Collection Book Projects\Chapter_4\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:\Collection Book Projects\Chapter_4\RaggedArray>_
```

Figure 4-23: Results of Running Example 4.11

Multidimensional Arrays In Action

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

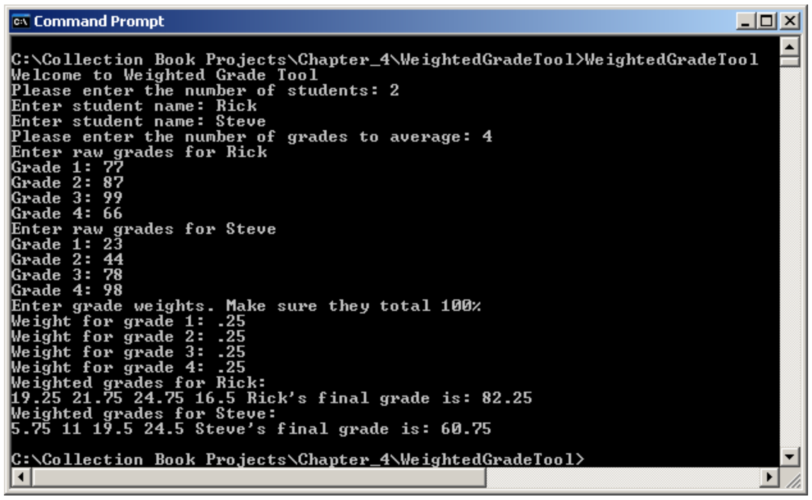
Weighted Grade Tool

Example 4.12 gives the code for a class named WeightedGradeTool. The program calculates a student's final grade based on weighted grades.

4.12 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] = 25.0;
                }
            }

            /***************** 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 4-24 shows the results of running this program.



```
C:\Collection Book Projects\Chapter_4\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:\Collection Book Projects\Chapter_4\WeightedGradeTool>
```

Figure 4-24: Results of Running Example 4.12

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.

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

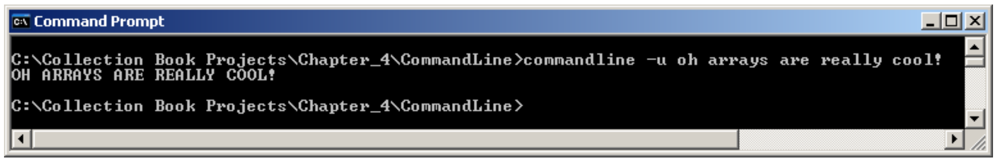
The purpose of the Main() method's string array is to enable C# applications to accept and act upon command-line arguments. The csc compiler is an example of a program that takes command-line arguments, the most important of which is the name of the file to compile. This chapter and the previous chapter 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 4.13 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.

4.13 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 4-25 shows the results of running this program.



```
C:\Collection Book Projects\Chapter_4\CommandLine>commandline -u oh arrays are really cool!
OH ARRAYS ARE REALLY COOL!

C:\Collection Book Projects\Chapter_4\CommandLine>
```

Figure 4-25: Results of Running Example 4.13

Manipulating Arrays With The System.Array Class

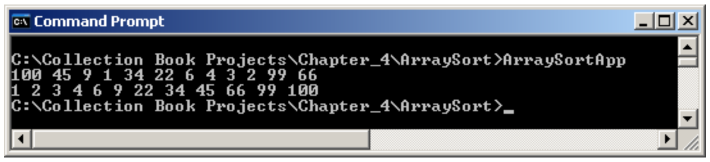
The .NET platform makes it easy to perform common array manipulations such as searching and sorting with the System.Array class. Example 4.14 offers a short program that shows the Array class in action sorting an array of integers.

4.14 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 4-26 shows the results of running this program.



```
C:\Collection Book Projects\Chapter_4\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:\Collection Book Projects\Chapter_4\ArraySort>_
```

Figure 4-26: Results of Running Example 4.14

Non Supported IList Operations

Although the System.Array class implements the IList interface, and array is a fixed size and does not grow automatically to accept new elements. Because of this fixed-size characteristic, there are four members of the IList interface that are not supported and any attempt to use them will throw a NonSupportedException. These include IList.Add(), IList.Insert(), IList.Remove(), and IList.RemoveAt().

Summary

C# array types have special functionality because of their special 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.

References

ECMA-335 Common Language Infrastructure (CLI), 4th Edition, June 2006 [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]

Rick Miller. C# For Artists: The Art, Philosophy, and Science of Object-Oriented Programming. ISBN-13: 978-1-932504-07-1. Pulp Free Press

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

Notes