Chapter 5

# Arrays and Collections

**In this chapter:**



An *array* is a collection of related instances, either value or reference types. Arrays possess an immutable structure, in which the number of dimensions and size of the array are fixed at instantiation. C# supports single-dimensional, multidimensional, and jagged arrays. Single-dimensional arrays, sometimes called *vectors*, consist of a single row. Multidimensional arrays are rectangular and consist of rows and columns. A jagged array also consists of rows and columns, but is irregularly shaped.

Arrays are represented in most programming languages. Most developers have some familiarity with the concept of arrays. Arrays are employed in a variety of ways. A personnel program would contain an array of employees. Graphic programs might have one array for each type of geometric object, such as ellipses, rectangles, or triangles. An accounting and scheduling application for automobile repair likely would have an array of automobile repair tickets. The Space Shuttle program would have an array of astronauts.

Arrays are intrinsic to the language. Other collections, such as *Stack*, *Queue*, and *Hashtable*, are not native to the language. As such, ease of use is one of the benefits of arrays. Another benefit is familiarity. Arrays are available and functionally similar in almost every programming language. Few developers have not worked with arrays.

An array is a container, which is an abstraction of a data structure. As a container, an array holds data items called *elements*. Elements of an array are always related, such as an array of apples or an array of oranges. An array might consist of *SalariedEmployee*, *HourlyEmployee*, and *CommissionedEmployee* instances. However, an array of apples and employees is probably invalid because those objects are probably unrelated. In addition, an array is a data structure that is a composite of elements that reside in contiguous memory.

Arrays are reference types, and the memory for the array is allocated on the managed heap. Even an array of value types is allocated on the managed heap and not on the stack. An array of 30 integer values would have the same number of 32-bit slots allocated in contiguous memory for the array elements. With arrays of reference types, the objects are not stored in contiguous memory; the references are stored in contiguous memory. However, the objects themselves are

stored in noncontiguous memory, which is pointed to by the reference. Figure 5-1 shows the difference in memory allocation between arrays of reference versus value types.

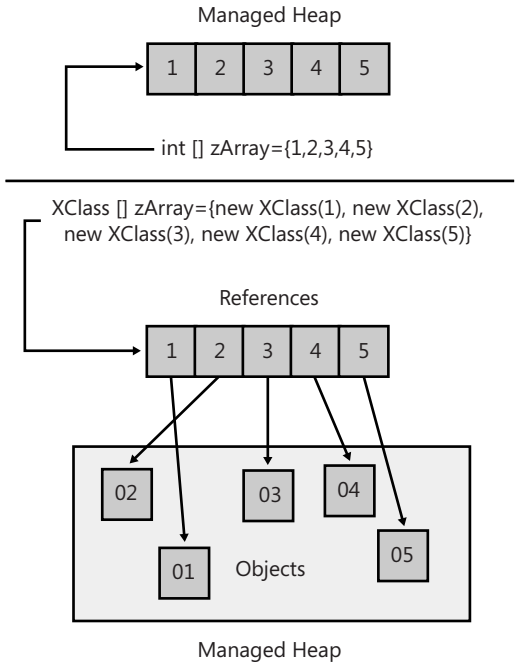


**Figure 5-1** Array of reference types versus value types

Elements are relative to the beginning of the array and are identified with indexes, which are either integer or long types. Indexes are also commonly called indices or subscripts, and placed inside the indexing operator ([ ]). Arrays are zero-based where the index is actually an offset. Array indexes are offsets from the beginning of the array to a particular element. Therefore, the first element is at the start of the array, which is an offset of zero. For an array of five elements, a proper index is from zero to four. Therefore, the last element of the array is at index *n -- 1*. This is a common cause of fencepost errors. Fencepost errors occur when referring to array elements with indexes outside the bounds of an array. If an array has five elements, accessing element six is a fencepost error.

As mentioned, arrays are immutable. This means that an array is statically sized, and the dimensions cannot be changed at run time. The *System.Array.Resize* method, which is a generic method, seemingly resizes an array. However, appearances can be deceiving. *Array.Resize* creates an entirely new array that is the new size. The new array is initialized with the elements of the source array. Afterward, the original array is discarded.

Single-dimensional arrays are indigenous to the run time. There are specific MSIL instructions for vectors, including *newarr*, *ldelem*, *ldlen*, and *stelem*. There are no built-in instructions for multidimensional arrays. This direct manipulation of single-dimensional arrays makes them more efficient. In addition, some of the members of the *System.Array* type, which is the underlying

type for all arrays, cannot be applied to multidimensional arrays. Conversely, all the methods and properties of the *System.Array* type are applicable to single-dimensional arrays.

The *System.Array* type is the underpinning of all arrays. It is an abstraction of an array. Instances of arrays are instances of the *System.Array* type. Thus, arrays are implicitly reference types. Arrays can access many of the instance methods and properties of the *System.Array* type. *System.Array* is a combination of original methods and the implementation of a series of interfaces. Table 5-1 lists the interfaces that *System.Array* implements.

**Table 5-1 Interfaces Implemented at System.Array**

| Interface | Description |
|---|---|
| *ICloneable* | This interface defines a method to clone an instance of an array. |
| *ICollection* | This interface defines methods to count the number of elements of an array and for thread synchronization. |
| *IEnumerable* | This interface defines a method that enumerates the elements of a collection. |
| *IList* | The interface defines methods to access an index-based collection. Some members of this interface are not publicly implemented. |

Create arrays as local variables or fields. Use arrays also as method parameters and return types. The rationale of using an array as a local variable, field, method parameter, or return type is the same as any type. Arrays are passable into or out of a method as a parameter. The semantics of an array parameter are the same as any parameter. Because an array is a reference type, it is passed by reference. Therefore, the content of the array can be changed in the called method. Parameter modifiers, such as the ref modifier, are assignable to array parameters.

# Arrays

Following is the syntax for declaring a one-dimensional array. A vector has a single index. When declaring an array, the indexing operator on the right-hand side sets the size of the array.

*type* [] *arrayname*[1];

*typea* [] *arrayname*[2]=new *typeb*[*n*];

*typea* [] *arrayname*[3]=new *typeb*[*n*] {*ilist*};

*typea*[] *arrayname*[4]=new *typeb*[] {*ilist*};

*typea*[] *arrayname*[5]={*ilist*};

The first syntax declares a reference to an array that is not initialized. You can later initialize the array reference using the right hand side (RHS) syntax of declaring an array. This is sample code of the first syntax of declaring an array. It declares an integer array named *zArray*. The array is then assigned an array of 10 integers that initializes the references. However, the array elements of 10 integers are not initialized. Array elements default to zero or null: zero for

value types and null for reference type elements. In this code, the array elements are set to zeroes:

```
int [] zArray;
zArray=new int[10];
```

The second syntax declares and initializes a reference to an array. The array reference is assigned a reference to the new instance. An array of a value types must be initialized to an array of the same value type. Otherwise, *typea* and *typeb* must be identical. Even if *typea* and *typeb* are convertible, the declaration is not allowed. Bytes are convertible to integers. However, an array of bytes is not convertible to an array of integers. The size of the array is *n*. The elements are indexed from zero to *n – 1*.

This is sample code of the second syntax:

```
byte aValue=10;
int bValue= aValue;         // convertible
int [] zArray=new byte[5];  // not convertible
int [] yArray=new int[5];   // convertible
```

Unlike value types, an array of reference types can be initialized to an array of the same or derived types. In the declaration, *typeb* must be the same or a derivation of *typea*, which is demonstrated in the following code:

```
public class Starter{
    public static void Main(){
        XBase [] obj=new XDerived[5];  // base    <- derived
        XDerived [] obj2=new XBase[5]; // derived <- base [invalid]
    }
}

public class XDerived: XBase {
}

public class XBase {
}
```

The third syntax declares, initializes, and assigns values to the array elements. The initialization list (*ilist*) contains the initial value for the elements of the array, where the values are comma-delimited. The number of values in the list should match the number of elements in the array exactly—no more and no less.

This is an example of the third syntax for declaring an array:

```
int [] zArray=new int[3] {1,2,3};  // valid
int [] yArray=new int[3] {1,2};    // invalid
ZClass [] xArray=new ZClass[3] {   // valid
    new ZClass(5), new ZClass(10),
    new ZClass(15) };
```

The fourth syntax also declares, initializes, and assigns values to array elements. However, in this circumstance, the initialization list sets the number of elements. The array size is not

stipulated in this syntax. The compiler counts the number of items in the initialization list to set the length of the array.

This is an example of the fourth syntax:

```
int [] zArray=new int[] {1,2,3,4,5};  // 5 elements
```

The fifth syntax is an abbreviation of the fourth syntax, where the array type and number of elements are inferred from the initialization list.

This is an example of the fifth syntax:

```
int [] yArray={1,2,3,4,5};
```

As in a local variable or field definition, multiple declarations can be combined:

```
public class ZClass{
    private int [] first={1,2,3},
                   second={4,5,6},
                   third={7,8,9};

    // Remainder of class...
}
```

## Array Elements

Indexing operators refer to elements of an array. With an index, the indexing operator returns a specific element of the array. When an indexing operator is used on the left hand side (LHS), the element value is changed. On the RHS, the indexing operator returns the value of the element.

The following *for* loop lists the elements of an array. The element and indexing operator appear as an *l*-value to total the array. It is used as an *r*-value to display each element:

```
int [] zArray={1,2,3,4,5};
int total=0;
for(int count=0;count<zArray.Length;++count) {
    total+=zArray[count];              // l-value
    int number=zArray[count];          // r-value
    Console.WriteLine(number);
}
Console.WriteLine("\nThe total is {0}.",
    total);
```

## Multidimensional Arrays

You are not limited to one-dimensional arrays. Multidimensional arrays are rectangular arrays and have multiple dimensions and indices. Two-dimensional arrays, which consist of rows and columns, are the most prevalent kind of multidimensional array. Each row contains the same number of columns, thus making the array rectangular. From a geometric perspective,

the x-axis consists of rows and the y-axis consists of columns. Multidimensional arrays are stored in row-major order and are processed in row-major order.

The total number of elements in a multidimensional array is the product of the indices. For example, an array of 5 rows and 6 columns has 30 elements. The *Array.Length* property returns the total number of elements in the array. The *Array.GetLength* method returns the number of elements per index. The indices are numbered from zero. For a two-dimensional array, row is the zero dimension, whereas column is the one dimension. *GetLength(0)* would then return the number of rows in the multidimensional array.

This is the syntax to declare a two-dimensional array. Notice the indexing operator. Row and column indexes in the indexing operator are delimited with a comma.

*type* [,] *arrayname*[1];

*typea* [,] *arrayname*[2]=new *typeb*[*r,c*];

*typea* [,] *arrayname*[3]=new *typeb*[*r,c*] {*ilist*};

*typea*[,] *arrayname*[4]=new *typeb*[,] {*ilist*};

*typea*[,] *arrayname*[5]={*ilist*};

The following code shows various declarations of multidimensional arrays. The initialization list of a multidimensional array includes nested initialization lists for each row. If an array has two rows, the initialization list includes two nested initialization lists. This is the syntax of a nested initialization list:

{ {*nlist*}, {*nlist*}, {*nlist*} ...}

The following code shows the various declaration syntaxes, including nested initialization lists:

```
int [,] array1;                // syntax 1
array1=new int[1,2];
int [,] array2=new int[2,3];   // syntax 2
int [,] array3=new int[2,3] {  // syntax 3
    {1,2,3}, {4,5,6} };
int [,] array4=new int[,] {    // syntax 4
    {1,2,3}, {4,5,6} };
int [,] array5= {              // syntax 5
    {1,2,3}, {4,5,6} };
```

To access an element of a multidimensional array, specify a row and column index in the indexing operator. It can be used on the LHS and RHS to set or get the value of an element, respectively. The following code calculates the total of the elements. This requires enumerating all the elements of a multidimensional array.

```
int [,] zArray=new int[2,3] {
    {1,2,3}, {4,5,6} };
int total=0;
```

```
    for(int row=0;row<zArray.GetLength(0);++row) {
        for(int col=0;col<zArray.GetLength(1);++col) {
            total+=zArray[row, col];              // LHS
            int number=zArray[row, col];          // RHS
            Console.WriteLine(number);
        }
    }
    Console.WriteLine("\nThe total is {0}.",
        total);
```

We have been focusing on two-dimensional arrays. However, arrays can have more than two dimensions. In fact, there is no limit to the number of dimensions. Three- and four-dimensional arrays are less common than two-dimensional arrays, but are seen nonetheless. More dimensions are rarely enunciated. Most developers find multidimensional arrays beyond two indices mind-numbing to manage and manipulate. Additional dimensions require added comma-delimited indexes when the array is, declared, defined, and used.

This is an example of a four-dimensional array:

```
            int [,,,] array=new int[1,2,3,2]
                {{{{1,2}, {1,2},{1,2}},{{1,2},{1,2},{1,2}}}};
```

How is the preceding code interpreted? A multidimensional array can be viewed as a hierarchical array that consists of layers. Each layer represents a different level of array nesting. The previous example defines a single-dimensional array, which aggregates two nested arrays. The nested arrays each contain three other arrays. Each of these lower nested arrays contains two elements.

This is a diagram of the array hierarchy:

{ 1 }; // layer 1

{{ 1 },{ 2 }}; // layer 2

{{{ 1 }, { 2 },{ 3 }},{{{ 1 }, { 2 },{ 3 }}; // layer 3

{{{{1,2}, {1,2},{1,2}} , {{1,2},{1,2},{1,2}}}}; // layer 4

The following code demonstrates a practical use of a multidimensional array. The program maintains the grades of students. Each student attends two classes. Each class has a class name and grade. Object elements are defined to make the array generic. Strings, integers, reference types, or anything else can be placed in an object array. Everything is derived from *System.Object* in managed code. The downside is boxing and unboxing of grades, which are value types.

```
using System;

namespace Donis.CSharpBook{
    public class Starter{
        public static void Main(){
```

```
            string [] names={"Bob", "Ted", "Alice"};
            object [,,] grades=new object[3,2,2]
                {{{"Algebra",85},  { "English",75}},
                 {{"Algebra",95},  { "History",70}},
                 {{"Biology",100}, { "English",92}}};

            for(int iName=0;iName<names.Length;++iName) {
                Console.WriteLine("\n{0}\n", names[iName]);
                for(int iCourse=0;iCourse<2;++iCourse) {
                    Console.WriteLine("{0} {1}",
                        grades[iName, iCourse, 0],
                        grades[iName, iCourse, 1]);
                }
            }
        }
    }
}
```

## Jagged Arrays

The most frequently found definition of a jagged array is an array of arrays. More specifically, a jagged array is an array of vectors. Although other arrays are rectangular, a jagged array, as the name implies, is jagged as each vector in the array can be of different length. With jagged arrays, first define the number of rows or vectors in the jagged array. Second, declare the number of elements in each row.

The syntax for declaring a jagged array is similar to a multidimensional array. Instead of a single bracket ([*r*,*c*]), jagged arrays have two brackets ([*r*][]). When declaring a jagged array, the number of columns is not specified and is omitted. The number of elements in each row is set individually. This is the syntax of a jagged array:

*type* [][] *arrayname*[1];

*typea* [][] *arrayname*[2]=new *typeb*[*r*][];

*typea* [][] *arrayname*[3]=new *typeb*[*r*][] {*ilist*};

*typea*[][] *arrayname*[4]=new *typeb*[][] {*ilist*};

*typea*[][] *arrayname*[5]={*ilist*};

This is sample code for declaring a jagged array:

```
int [][] zArray;                    // syntax 1
int [][] yArray=new int[3][];       // syntax 2
int [][] xArray=new int[3][]        // syntax 3
    {new int [] {1,2,3},
     new int[] {1,2},
     new int[] {1,2,3,4}};
int [][] wArray=new int[][]{        // syntax 4
     new int [] {1,2,3},
     new int[] {1,2},
```

```
            new int[] {1,2,3,4}};
        int [][] wArray={                    // syntax 5
            new int [] {1,2,3},
            new int[] {1,2},
            new int[] {1,2,3,4}};
```

The rows of the jagged array are initialized to one-dimensional arrays. Because the rows are assigned distinct arrays, the length of each row may vary. Therefore, a jagged array is essentially an array of vectors:

```
        jarray[row]=new type[elements];
```

Here is sample code that employs a jagged array. Each row of the jagged array has an increasing number of elements. The first nested loop creates and initializes each row of the jagged array. At the end, the values of each row are totaled and displayed.

```
using System;

namespace Donis.CSharpBook{
    public class Starter{
        public static void Main(){
            int [][] jagged=new int [7][];
            int count=0;
            for(int row=0;row<jagged.GetLength(0);++row) {
                Console.Write("\nRow {0}:", row);
                jagged[row]=new int[row+1];
                for(int index=0; index<row+1; ++index) {
                    ++count;
                    jagged[row][index]=count;
                    Console.Write(" {0}", count);
                }
            }
            Console.WriteLine("\n\nTotals");
            for(int row=0;row<jagged.GetLength(0);++row) {
                int total=0;
                for(int index=0; index<jagged[row].GetLength(0);
                        ++index) {
                    total+=jagged[row][index];
                }
                Console.Write("\nRow {0}: {1}",
                    row, total);
            }

        }
    }
}
```

Vectors and multidimensional arrays are both collections. As such, arrays implement a combination of an array and some collection-specific interfaces, which are encapsulated in the *System.Array* type. *System.Array* implements the *ICollection*, *IEnumerable*, *IList*, and *ICloneable* interfaces. *System.Array* also implements array-specific behaviors, such as the *Array.GetLength* method.

## System.Array

The *System.Array* type houses the fundamental methods and properties that are essential to an array. This includes sorting, reversing, element count, synchronization, and much more. Table 5-2 lists the methods of the *System.Array* type. Many of the methods are static, which is noted in the syntax. In addition, some methods are for single-dimensional arrays and are not usable with multidimensional arrays. This fact is noted in the method description in Table 5-2.

**Table 5-2 *System.Array* Members**

| Description | Syntax |
|---|---|
| *AsReadOnly*<br>This is a generic method that returns a read-only wrapper for an array. | `static ReadOnlyCollection<T>`<br>`  AsReadOnly<T>(`<br>`  T[] sourceArray)` |
| *BinarySearch*<br>This method conducts a binary search for a specific value in a sorted one-dimensional array.<br>There are several overloads for this method. The two more common overloads are shown. | `static int BinarySearch(`<br>`  Array sourceArray,`<br>`  object searchValue)`<br><br>`static int BinarySearch<T>(`<br>`  T[] sourceArray,`<br>`  T value)` |
| *Clear*<br>This method sets a range of elements to zero, null, or false. | `static void Clear(Array sourceArray,`<br>`  int index, int length)` |
| *Clone*<br>This method clones the current array. | `sealed object Clone()` |
| *ConstrainedCopy*<br>This method copies a range of elements from the source array into a destination array. You set the source index and destination index, where the copy is started in both arrays. | `static void ConstrainedCopy{`<br>`  Array sourceArray,`<br>`  int sourceIndex`<br>`  Array destinationArray,`<br>`  int destinationIndex,`<br>`  int length)` |
| *ConvertAll*<br>This is a generic method that converts the type of an array. | `static <destinationType>`<br>`  ConvertAll<sourceType,`<br>`  destinationType>(`<br>`  sourceType sourceArray,`<br>`  Converter<sourceType,`<br>`  destinationType> converter)` |
| *Copy*<br>This method copies elements from the source array to the destination array. The specified number of elements is copied.<br>There are four overloads to this method. The two more common overloads are listed. | `static void Copy(`<br>`  Array sourceArray,`<br>`  Array destinationArray,`<br>`  int length)`<br><br>`static void Copy(`<br>`  Array sourceArray,`<br>`  int sourceIndex,`<br>`  Array destinationArray,`<br>`  int destinationIndex,`<br>`  int length)` |

**Table 5-2** *System.Array* **Members (Continued)**

| Description | Syntax |
|---|---|
| *CopyTo*<br>This method copies the current one-dimensional array to the destination array starting at the specified index. | `void CopyTo(Array destinationArray,`<br>`  int index)`<br>`void CopyTo(Array destinationArray,`<br>`  long index)` |
| *CreateInstance*<br>This method creates an instance of an array at run time.<br>This method has several overloads. One-dimensional and two-dimensional versions of the method are listed. | `static Array CreateInstance(`<br>`  Type arrayType,`<br>`  int length)`<br><br>`static Array CreateInstance(`<br>`  Type arrayType,`<br>`  int rows,`<br>`  int cols)` |
| *Exists*<br>This is a generic method that confirms that an element matches the conditions set in the predicate function. | `static bool Exist<T> {`<br>`  T [] sourceArray,`<br>`  Predicate<T> match)` |
| *Find*<br>This is a generic method that finds the first element that matches the conditions set in the predicate function. | `static T Find<T>(`<br>`  T[] sourceArray,`<br>`  Predicate<T> match)` |
| *FindAll*<br>This is a generic method that returns all the elements that match the conditions set in the predicate function. | `static T[] FindAll<T>(`<br>`  T[] sourceArray,`<br>`  Predicate<T> match)` |
| *FindIndex*<br>This is a generic method that returns the index to the first element that matches the conditions set in the predicate function. | `static int FindIndex<T>(`<br>`  T[] sourceArray,`<br>`  Predicate<T> match)`<br><br>`static int FindIndex<T>(`<br>`  T[] sourceArray,`<br>`  int startingIndex,`<br>`  Predicate<T> match)`<br><br>`static int FindIndex(`<br>`  T[] sourceArray,`<br>`  int startingIndex,`<br>`  int count,`<br>`  Predicate<T> match)` |
| *FindLast*<br>This is a generic method that returns the last element that matches the conditions set in the predicate function. | `static T FindLast<T>{`<br>`  T[] sourceArray,`<br>`  Predicate<T> match)` |
| *FindLastIndex*<br>This is a generic method that returns the index to the last element that matches the conditions set in the predicate function. | `static int FindLastIndex(T[] sourceArray,`<br>`  Predicate<T> match)`<br><br>`static int FindLastIndex(T[] sourceArray,`<br>`  int startingIndex,`<br>`  Predicate<T> match)`<br><br>`static int FindLastIndex(T[] sourceArray,`<br>`  int startingIndex,`<br>`  int count,`<br>`  Predicate<T> match)` |

**Table 5-2** *System.Array* **Members (Continued)**

| Description | Syntax |
|---|---|
| *ForEach*<br>This is a generic method that performs an action on each element of the array, where action refers to a function. | `public static void ForEach<T>(`<br>`  T[] array,`<br>`  Action<T> action)` |
| *GetEnumerator*<br>This method returns an enumerator that implements the enumerator pattern for collections. You can enumerate the elements of the array with the enumerator object. | `sealed IEnumerator GetEnumerator()` |
| *GetLength*<br>This method returns the number of elements for a dimension of an array. | `int GetLength(int dimension)` |
| *GetLongLength*<br>This method returns the number of elements as a 64-bit integer for a dimension of an array. | `long GetLongLength(int dimension)` |
| *GetLowerBound*<br>This method returns the lower bound of a dimension, which is usually zero. | `int GetLowerBound(int dimension)` |
| *GetUpperBound*<br>This method returns the upper bound of a dimension. | `int GetUpperBound(int dimension)` |
| *GetValue*<br>This method returns the value of an element at the specified index.<br>This method has several overloads. A one-dimensional version and a multidimensional version of the method are shown here. | `object GetValue(int index)`<br>`object GetValue(params int[] indices)` |
| *IndexOf*<br>This method returns the index of the first element in a one-dimensional array that has the specified value.<br>This method has several overloads. A generic and a nongeneric version of the method are listed. | `static int IndexOf(Array sourceArray,`<br>`  object find)`<br>Generic version:<br>`static int IndexOf<T>(T[] sourceArray,`<br>`  T value)` |
| *Initialize*<br>This method initializes every element of the array. The default constructor of each element is called. | `void Initialize()` |
| *LastIndexOf*<br>This method returns the index of the last element that matches the specified value in a one-dimensional array.<br>This method has several overloads. A generic version and a nongeneric version are listed. | `static int LastIndexOf(Array sourceArray,`<br>`  object value)`<br>Generic version:<br>`static int LastIndexOf<T>(T[] sourceArray,`<br>`  T value)` |

**Table 5-2** ***System.Array*** **Members (Continued)**

| Description | Syntax |
|---|---|
| *Resize*<br>This is a generic method that changes the size of a one-dimensional array. | `static void Resize<T>(`<br>`  ref T[] sourceArray,`<br>`  int newSize)` |
| *Reverse*<br>This method reverses the order of elements in a one-dimensional array. | `static void Reverse(Array sourceArray)`<br><br>`static void Reverse(Array sourceArray,`<br>`  int index, int length)` |
| *SetValue*<br>This method sets the value of a specific element of the current one-dimensional array.<br>This method has several overloads. Two of the overloads are listed. | `void SetValue(object value, int index)`<br><br>`void SetValue(object value,`<br>`  params int[] indices)` |
| *Sort*<br>This method sorts the elements of a one-dimensional array.<br>This method has several overloads. A nongeneric version and a generic version of the method are listed. | `static void Sort(Array sourceArray)`<br><br>`static void Sort<T>(`<br>`  T[] sourceArray)` |
| *TrueForAll*<br>This is a generic method that returns *true* if all elements of an array match the conditions set in the predicate function. | `static bool TrueForAll<T>(`<br>`  T[] array,`<br>`  Predicate<T> match)` |

The following sections offer sample code and additional descriptions for some of the *System.Array* methods.

## Array.AsReadOnly Method

The following code creates and initializes an integer array. The second element of the array is then modified, which demonstrates the read-write capability of the collection. *Array.AsReadOnly* is called to wrap the array in a read-only collection. The *ReadOnlyCollection* type is found in the *System.Collections.ObjectModel* namespace. After displaying the elements of the read-only collection, the code attempts to modify an element. Since the collection is read-only, a compile error occurs at this line:

```
using System;
using System.Collections.Generic;
using System.Collections.ObjectModel;


namespace Donis.CSharpBook{
    public class Starter{
        public static void Main(){
            int [] zArray={1,2,3,4};
            zArray[1]=10;
```

```
            ReadOnlyCollection<int> roArray=Array.AsReadOnly(zArray);
            foreach(int number in roArray) {
                Console.WriteLine(number);
            }
            roArray[1]=2;   //  compile error
        }
    }
}
```

## Array.Clone Method

In the following code, the *CommissionedEmployee* type inherits from the *Employee* type. An array of commissioned employees is defined and then cloned with the *Clone* method. The clone type is an array of *Employees*. Because *Clone* returns an object array, which is unspecific, you should cast to a specific array type. The cast from *System.Object* is not type-safe, and an incorrect cast would cause an exception. Polymorphism is employed in the *foreach* loop to call the *Pay* method of the derived class.

```
using System;
using System.Collections.Generic;


namespace Donis.CSharpBook{
    public class Starter{
        public static void Main(){
            CommissionedEmployee [] salespeople=
                {new CommissionedEmployee("Bob"),
                 new CommissionedEmployee("Ted"),
                 new CommissionedEmployee("Sally")};

            Employee [] employees=
                (Employee [])salespeople.Clone();

            foreach(Employee person in
                    employees) {
                person.Pay();
            }
        }
    }

    public class Employee {
        public Employee(string name) {
            m_Name=name;
        }

        public virtual void Pay() {
            Console.WriteLine("Paying {0}", m_Name);
        }

        private string m_Name;
    }
```

```
    public class CommissionedEmployee: Employee {
        public CommissionedEmployee(string name) :
            base(name) {
        }

        public override void Pay() {
            base.Pay();
            Console.WriteLine("Paying commissions");
        }

    }

}
```

## Array.CreateInstance Method

The following code demonstrates both the *CreateInstance* and *SetValue* methods. *CreateInstance* creates a new array at run time. This requires some degree of reflection, which will be discussed in Chapter 10, "Metadata and Reflection." This code reads the array type, method to call, and initial values for each element from the command line. *CreateInstance* creates a new array using the type name read from the command line. In the *for* loop, *Activator.CreateInstance* creates instances of the element type, where values from the command line are used to initialize the object. In the *foreach* loop, the elements of the array are enumerated while calling the method stipulated in the command-line arguments as the second parameter.

```
using System;
using System.Reflection;

namespace Donis.CSharpBook{
    public class Starter{
        public static void Main(string [] argv){
            Assembly executing=Assembly.GetExecutingAssembly();
            Type t=executing.GetType(argv[0]);
            Array zArray=Array.CreateInstance(
                t, argv.Length-2);
            for(int count=2;count<argv.Length;++count) {
                System.Object obj=Activator.CreateInstance(t, new object[] {
                    argv[count]});
                zArray.SetValue(obj, count-2);
            }
            foreach(object item in zArray) {
                MethodInfo m=t.GetMethod(argv[1]);
                m.Invoke(item, null);
            }
      }
    }

    public class ZClass {
        public ZClass(string info) {
            m_Info="ZClass "+info;
        }

        public void ShowInfo() {
```

```
            Console.WriteLine(m_Info);
        }

        private string m_Info;
    }

    public class YClass {
        public YClass(string info) {
            m_Info="YClass "+info;
        }

        public void ShowInfo() {
            Console.WriteLine(m_Info);
        }

        private string m_Info;
    }

    public class XClass {
        public XClass(string info) {
            m_Info="XClass "+info;
        }

        public void ShowInfo() {
            Console.WriteLine(m_Info);
        }

        private string m_Info;
    }
}
```

A typical command line and results of the application are shown in Figure 5-2.

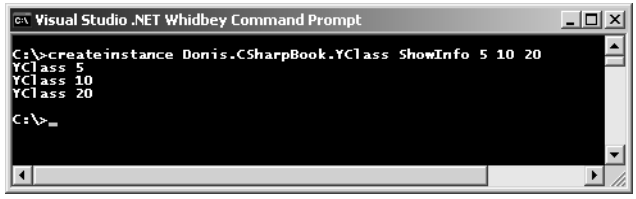


**Figure 5-2** A command line and the results from running the application

## Array.FindAll Method

Several *System.Array* methods use predicates, such as the *Exists*, *Find*, *FindAll*, and *FindLastIndex* methods. Predicates are delegates initialized with functions that find matching elements. The predicate function is called for each element of the array. A conditional test is performed in the function to isolate matching elements; *true* or *false* is returned from the predicate indicating that a match has or has not been found, respectively.

This is the syntax of the Predicate delegate:

```
delegate bool Predicate<T>(T obj)
```

Predicate methods are generic methods. The type parameter indicates the element type. The return value is the result of the comparison.

The following code finds all elements equal to three. *MethodA* is the predicate method, which compares each value to three.

```
public static void Main(){
   int [] zArray={1,2,3,1,2,3,1,2,3};
   Predicate<int> match=new Predicate<int>(MethodA<int>);
   int [] answers=Array.FindAll(zArray, match);
   foreach(int answer in answers) {
       Console.WriteLine(answer);
   }
}

public static bool MethodA<T>(T number) where T:IComparable {
    int result=number.CompareTo(3);
    return result==0;
}
```

### Array.Resize Method

The *Resize* method resizes a one-dimensional array.

Here is sample code for resizing an array. The elements added to the array are initialized to a default value.

```
using System;

namespace Donis.CSharpBook{
    public class Starter{
        public static void Main(){
            int [] zArray={1,2,3,4};
            Array.Resize<int>(ref zArray, 8);
            foreach(int number in zArray) {
                Console.WriteLine(number);
            }
        }
    }
}
```

## System.Array Properties

*System.Array* has several properties that are useful when working with arrays. Table 5-3 lists the various properties.

**Table 5-3** ***System.Array*** **Properties**

| Description | Syntax |
|---|---|
| *IsFixedSize*<br>This property returns *true* if the array is a fixed size. Otherwise, *false* is returned. Always *true* for arrays. | `virtual bool IsFixedSize{ get;}` |
| *IsReadOnly*<br>This property returns *true* if the array is read-only. Otherwise, *false* is returned. Always *false* for arrays. | `virtual bool IsReadOnly{ get;}` |
| *IsSynchronized*<br>This property returns *true* if the array is thread-safe. Otherwise, *false* is returned. Always *false* for arrays. | `virtual bool IsSynchronized{ get;}` |
| *Length*<br>This property returns the number of elements in the array. | `int Length{ get;}` |
| *LongLength*<br>This property returns the number of elements in the array as a 64-bit value. | `Long LongLength{ get;}` |
| *Rank*<br>This property returns the rank of the array, which is the number of dimensions. For example, a two-dimensional array has a rank of two. | `int Rank{ get;}` |
| *SyncRoot*<br>This property returns a synchronization object for the current array. Arrays are not inherently thread-safe. Synchronize access to the array with the synchronization object. | `virtual object SyncRoot{ get;}` |

Many of the preceding properties are used in sample code. The *SyncRoot* property is particularly important and not included in previous sample code.

## Array.SyncRoot Property

The purpose of the *SyncRoot* object is to synchronize access to an array. Arrays are unsafe data structures. As documented, the *IsSynchronized* property always returns *false* for an array. Accesses to arrays are easily synchronized with the *lock* statement, where the *SyncRoot* object is the parameter.

In the following code, the array is a field in the *Starter* class. The *DisplayForward* and *DisplayReverse* methods list array elements in forward and reverse order correspondingly. The functions are invocated at threads, in which the *SyncLock* property prevents simultaneous access in the concurrent threads.

```
using System;
using System.Threading;

namespace Donis.CSharpBook{
    public class Starter{
```

```
        public static void Main(){
            Array.Sort(zArray);
            Thread t1=new Thread(
                new ThreadStart(DisplayForward));
            Thread t2=new Thread(
                new ThreadStart(DisplayReverse));
            t1.Start();
            t2.Start();
        }

        private static int [] zArray={1,5,4,2,4,2,9,10};
        public static void DisplayForward() {
            lock(zArray.SyncRoot) {
                Console.Write("\nForward: ");
                foreach(int number in zArray) {
                    Console.Write(number);
                }
            }
        }

        public static void DisplayReverse() {
            lock(zArray.SyncRoot) {
                Array.Reverse(zArray);
                Console.Write("\nReverse: ");
                foreach(int number in zArray) {
                    Console.Write(number);
                }
                Array.Reverse(zArray);
            }
        }
    }
}
```

## Comparable Elements

*System.Array* methods require elements to be instances of comparable types.

- *Array.IndexOf*
- *Array.LastIndexOf*
- *Array.Sort*
- *Array.Reverse*
- *Array.BinarySearch*

Comparable types implement the *IComparable* interface, which requires the implementation of the *CompareTo* method. The *CompareTo* method returns zero when the current and target instances are equal. If the current instance is less than the target, a negative value is returned. A positive value is returned if the current instance is greater than the target. The preceding methods call *IComparable.CompareTo* to perform the required comparisons to sort, reverse, and otherwise access the array in an ordered manner.

A run-time error occurs in the following code when *Array.Sort* is called. Why? The *XClass* does not implement the *IComparable* interface.

```
using System;

namespace Donis.CSharpBook{
    public class Starter{
        public static void Main(){
            XClass [] objs={new XClass(5), new XClass(10),
                new XClass(1)};
            Array.Sort(objs);
        }
    }

    public class XClass {
        public XClass(int data) {
            propNumber=data;
        }

        private int propNumber;
        public int Number {
            get {
                return propNumber;
            }
        }
    }
}
```

Here is the proper code, where the *XClass* implements the *IComparable* interface. This program runs successfully.

```
using System;

namespace Donis.CSharpBook{
    public class Starter{
        public static void Main(){
            XClass [] objs={new XClass(5), new XClass(10),
                new XClass(1)};
            Array.Sort(objs);
            foreach(XClass obj in objs) {
                Console.WriteLine(obj.Number);
            }
        }
    }

    public class XClass: IComparer {
        public XClass(int data) {
            propNumber=data;
        }

        private int propNumber;
        public int Number {
            get {
                return propNumber;
            }
```

```
        }

        public int CompareTo(object obj) {
            XClass comp=(XClass) obj;
            if(this.Number==comp.Number){
                return 0;
            }
            if(this.Number<comp.Number){
                return -1;
            }
            return 1;
        }
    }
}
```

Many of the methods and properties of *System.Array* are mandated in interfaces that the type implements. The following section lists those interfaces and methods.

## ICollection Interface

The *ICollection* interface is one of the interfaces implemented in *System.Array* type; it returns the count of elements and supports synchronization of collections. The members of the *ICollection* interface are as follows:

- *CopyTo method*
- *Count property*
- *IsSynchronized property*
- *SyncRoot property*

## ICloneable Interface

*System.Array* implements the *ICloneable* interface. This is the interface for duplicating an object, such as an array. The only member of this interface is the *Clone* method.

## IEnumerable

*System.Array* type implements the *IEnumerable* interface. *IEnumerable.GetEnumerator* is the sole member of this interface. *GetEnumerator* returns an enumerator object that implements the *IEnumerator* interface. The enumerator provides a consistent interface for enumerating any collection. The benefit is the ability to write a generic algorithm, which requires enumerating the elements of a collection in a consistent manner. For example, the *foreach* statement uses an enumerator object to consistently iterate all collection types.

Chapter 7, "Iterators," will focus more on enumerators.

Here is sample code that enumerates an array using an enumerator object:

```
using System;
using System.Collections;
```

```
namespace Donis.CSharpBook{
    public class Starter{
        public static void Main(){
            int [] numbers={1,2,3,4,5};
            IEnumerator e=numbers.GetEnumerator();
            while(e.MoveNext()) {
                Console.WriteLine(e.Current);
            }
        }
    }
}
```

## IList Interface

*System.Array* type implements the *IList* interface. However, only part of this implementation is publicly available. The private implementation of some *IList* members effectively removes those members from the public interface. These are members contrary to the array paradigm, such as the *RemoveAt* method. Arrays are immutable. You cannot remove elements from the middle of an array.

Table 5-4 lists the *IList* interface and indicates the public versus private implementation in *System.Array*.

**Table 5-4 List Members**

| Member Name | Public or Private |
|---|---|
| *Add* | Private |
| *Clear* | Public |
| *Contains* | Private |
| *IndexOf* | Public |
| *Insert* | Private |
| *Remove* | Private |
| *RemoveAt* | Private |

## Indexers

You can treat instances as arrays by using indexers. Indexers are ideal for types that wrap collections, in which additional functionality is added to the type that augments the collection. Access the object with the indexing operator to get or set the underlying collection. Indexers are a combination of an array and a property. The underlying data of an indexer is often an array or a collection. The indexer defines a *set* and *get* method for the *this* reference. Indexers are considered a default property because the indexer is a nameless property. Internally, the compiler inserts the *get_Item* and *set_Item* method in support of indexers.

Indexers are fairly typical properties. However, there are some exceptions. Here are some of the similarities and differences between indexers and properties:

- Indexers can be overloaded.
- Indexers can be overridden.

- Indexers can be added to interfaces.
- Indexers support the standard access modifiers.
- Indexers cannot be a static member.
- Indexers are nameless and associated with the *this* reference.
- Indexers parameters are indices. Properties do not have indices.
- Indexers in the base class are accessed as base[*indices*], while a similar property is accessed base.*Property*.

Indexers are also similar to arrays:

- Indexers are accessed using indices.
- Indexers support non-numeric indices. Arrays support only integral indices.
- Indexers use a separate data store, whereas an array is the data store.
- Indexers can perform data validation, which is not possible with an array.

This is the syntax of an indexer:

*accessibility modifier type this*[*parameters*]

{ *attributes* get {*getbody*} *attributes* set {*setbody*} }

Except for static accessibility, indexers have the same accessibility and modifiers of a normal property. Indexers cannot be static. The parameters are a comma-delimited list of indexes. The list includes the type and name of each parameter. Indexer indices can be nonintegral types, such as string and even *Employee* types.

The following is example code for indexers. The *Names* type is a wrapper of an array of names and ages. The indexer is read-only in this example and provides access to the array field. Per the parameters of the indexer, the indexer manipulates object elements and has a single index. However, the *_names* array is two-dimensional, which requires two indexes. Flexibility is one of the benefits of indexers versus standard arrays. In the sample code, the parameter of the indexer represents the row, whereas the column is a constant.

```
using System;

namespace Donis.CSharpBook{
    public class Starter{
        public static void Main(){
            Names obj=new Names();
            Console.WriteLine(obj[1]);
        }
    }

    public class Names {
        object [,] _names={
             {"Valerie", 27},
```

```
             {"Ben", 35},
             {"Donis", 29}};

        public object this[int index] {
            get {
                 return _names[index,0]+" "+_names[index,1];
            }
        }
    }
}
```

Indexers can be overloaded based on the parameter list. Overloaded indexers should have a varying number of parameters or parameter types. The following code overloads the indexer property twice. The first property is read-only and returns the name and age information. The second overload is a read-write property that sets and gets the age of a person. This property, which uses a string parameter, also demonstrates non-numerical indexes.

```
using System;

namespace Donis.CSharpBook{
    public class Starter{
        public static void Main(){
            Names obj=new Names();
            obj["Donis"]=42;
            Console.WriteLine(obj["Donis"]);
        }
    }

    public class Names {
        object [,] _names={
             {"Valerie", 27},
             {"Ben", 35},
             {"Donis", 29}};

        public object this[int index] {
            get {
                 return _names[index,0]+" "+_names[index,1];
            }
        }

        public object this[string sIndex] {

            get {
                 int index=FindName(sIndex);
                 return _names[index, 1];
            }
            set {
                 int index=FindName(sIndex);
                 _names[index, 1]=value;
            }
        }

        private int FindName(string sIndex) {
            for(int index=0; index<_names.GetLength(0);
```

```
                ++index) {
            if((string)(_names[index,0])==sIndex) {
                return index;
            }
        }
        throw new Exception("Name not found");
    }
  }
}
```

## params Keyword

The *params* keyword is a parameter modifier, which indicates that a parameter is a one-dimensional array of any length. This defines a variable-length parameter list. The *params* modifier can be applied to only one parameter in the parameter list, which must be the final parameter. The *ref* and *out* modifiers cannot be applied to the *params* modified parameter. You have probably used methods of the Microsoft .NET Framework class library (FCL) that have variable-length parameter lists, such as *Console.WriteLine*. *Console.WriteLine* accepts a variable number of arguments. It has a *param* parameter.

Initialize the *params*-modified argument with an implicit or explicit array. This is done at the call site. For implicit initialization, the C# compiler consumes the optional parameters that follow the fixed parameter list. The fixed parameters are the parameters that precede the *params*-modified parameter in the function signature. The optional arguments are consolidated into an array. If there are six optional arguments after the fixed arguments, an array of six elements is created and initialized with the values of the optional arguments. Alternatively, an explicit array can be used. Finally, the *params* argument can be omitted. When omitted, the *params* parameter is assigned an empty array. An empty array is different from a null array. Empty arrays have no elements but are valid instances.

In the following code, *Names* is a static method, where the *params* modifier is applied to the second parameter. Therefore, the *employees* argument is a single-dimensional string array, and the *Names* method offers a variable-length parameter list.

```
public static void Names(string company,
    params string [] employees) {
    Console.WriteLine("{0} employees: ",
        company);
    foreach(string employee in employees) {
        Console.WriteLine("  {0}", employee);
    }
}
```

For a variable-length parameter list, the number of parameters, which includes the parameter array, is set at the called site. The following code shows the *Names* method being called with varying numbers of parameters. In both calls, the first parameter is consumed by the company parameter. The remaining parameters are used to create an array, which is assigned to

the *params* parameter. A three-argument array is created for the first method call, whereas the second method creates a six-argument array that is assigned to the *params* parameter.

```
Names("Fabrikam","Fred", "Bob", "Alice");
Names("Contoso", "Sally", "Al", "Julia",
    "Will", "Sarah", "Terri");
```

The following code calls the *Names* method with an explicit array. This is identical to calling the method with three optional arguments.

```
Names("Fabrikam", new string [] {"Fred", "Bob",
            "Alice"});
```

In this statement, the *Names* method is called without a *params* argument. For the omitted parameter, the compiler creates an array with no elements, which is subsequently passed to the method.

```
Names("Fabrikam");
```

Variable-length methods can be overloaded similar to any method. You can even overload a method with a fixed number of parameters with a method with a variable number of parameters. Where there is ambiguity, the method with the fixed number of parameters is preferred and called.

In the following code, the *Names* method is overloaded with three methods. The first two overloads have a variable-length parameter list, whereas the final method has a fixed-length parameter list. In *Main*, the first two calls of the *Names* method are not ambiguous. The final call is ambiguous and can resolve to either the first or third overload. Because the third overload has a fixed-length parameter list, it is called instead of the first overloaded method.

```
using System;

namespace Donis.CSharpBook{
    public class Starter{
        public static void Main(){
            Names("Fabrikam","Fred", "Bob", "Alice");
            Names("Fabrikam",1234, 5678, 9876, 4561);
            Names("Fabrikam","Carter", "Deborah");
        }

        public static void Names(string company,
            params string [] employees) {
            Console.WriteLine("{0} employees: ",
                company);
            foreach(string employee in employees) {
                Console.WriteLine("  {0}", employee);
            }
        }

        public static void Names(string company,
            params int [] emplid) {
```

```
            Console.WriteLine("{0} employees: ",
                company);
            foreach(int employee in emplid) {
                Console.WriteLine("  {0}", employee);
            }
        }

        public static void Names(string company,
            string empl1, string empl2) {
            Console.WriteLine("{0} employees: ",
                company);
            Console.WriteLine("  {0}",empl1);
            Console.WriteLine("  {0}",empl2);
        }
    }
}
```

# Array Conversion

You can cast between arrays. Arrays are implicit *System.Array* types. For that reason, regardless of type or the number of dimensions, any array can be cast to *System.Array*. All arrays are compatible with that type.

When casting or converting between arrays, the source and destination array are required to have the same dimensions. In addition, an array of value types is convertible only to arrays of the same dimension and type. Arrays of reference types are somewhat more flexible. Arrays of reference types can be converted to arrays of the same or ascendant type. This is called *array covariance*. Array reference types are covariant, whereas arrays of value types are not.

Arrays can be inserted as function parameters and returns. In these roles, proper conversion is important.

## Arrays as Function Returns and Parameters

Arrays provided as function arguments are passed by reference. This is more efficient than caching potentially a large number of elements on the stack. As a reference type, the array state can be changed in the called method. Array parameters are normal parameters and support the regular assortment of modifiers. Of course, the array argument must be convertible to the array parameter.

In the following code, the *ZClass* has two static methods, which both have an array parameter. The *ListArray* method has a *System.Array* parameter, which accepts any array argument. For this reason, the *ListArray* method is called in *Main* with different types of array arguments. The *Total* method limits array arguments to one-dimensional integer arrays.

```
using System;

namespace Donis.CSharpBook{
    public class Starter{
```

```
        public static void Main(){
            int [] zArray={10,9,8,7,6,5,4,3,2,1};
            string [] xArray={"a", "b", "c", "d"};
            Console.WriteLine("List Numbers");
            ZClass.ListArray(zArray);
            Console.WriteLine("List Letters");
            ZClass.ListArray(xArray);
            Console.WriteLine("Total Numbers");
            ZClass.Total(zArray);
        }
    }


    public class ZClass {

        public static void ListArray(Array a) {
            foreach(object element in a) {
                Console.WriteLine(element);
            }
        }

        public static void Total(int [] iArray) {
            int total=0;
            foreach(int number in iArray) {
                total+=number;
            }
            Console.WriteLine(total);
        }
    }
}
```

Arrays can also be returned from functions, which is one means of returning more than a single value. You can return multiple values as elements of an array. Returning an array gives the calling function a reference to the array, which provides direct access to the array. Arrays are not returned on the stack.

# Collections

Arrays are the most popular collection. However, the .NET FCL offers a variety of other collections with different semantics. Collections are abstractions of data algorithms. *ArrayList* is an abstraction of a dynamic array, the *Stack* collection abstracts a stack data structure, the *Queue* collection abstracts queues, the *Hashtable* collection abstracts a lookup table, and so on. Each collection exposes both unique and standard interfaces. The unique interface is specific to the collection type. For example, the *Stack* type has pop and push methods, whereas the *Queue* type has *Dequeue* and *Enqueue* methods. Collections minimally implement the *ICollection*, *IEnumerable*, and *ICloneable* interfaces. (These interfaces were described earlier in this chapter.) The nongeneric collection classes are implemented in the *System.Collections* namespace. Generic collections are reviewed in Chapter 6, "Generics."

Table 5-5 lists the nongeneric collections in the .NET FCL.

**Table 5-5 Collection Types**

| Class Name | Description |
|---|---|
| *ArrayList* | Dynamic array |
| *BitArray* | Bit array |
| *Hashtable* | Lookup table of keys and values |
| *Queue* | First-in/first-out (FIFO) collection of elements |
| *SortedList* | Sorted list of elements |
| *Stack* | Last-in/first-out (LIFO) collection of elements |

What follows is a detailed explanation of each collection found in the *Collections* namespace. The explanations are complemented with sample code, which demonstrates the uniqueness of each collection.

## ArrayList Collection

An array list is a dynamic array. Although indigenous arrays are static, elements can be added or removed from an *ArrayList* at run time. Elements of the *ArrayList* are not automatically sorted. Similar to single-dimensional arrays, the elements of an *ArrayList* are accessible using the indexing operator and indices.

In addition to the standard collection interfaces, *ArrayList* implements the *IList* interface.

Table 5-6 lists the *ArrayList*-specific methods and properties. The static members of *ArrayList* are thread-safe, whereas instance members are not. The common collection interfaces—*ICollection*, *IEnumerable*, and *ICloneable*—are not discussed in detail. (These interfaces and their members were reviewed earlier in the chapter.)

**Table 5-6 ArrayList Members**

| Member Name | Syntax |
|---|---|
| *Constructors* | `ArrayList()`<br>`ArrayList(`<br>`  ICollection sourceCollection)`<br>`ArrayList(int capacity)` |
| *Adapter*<br>This method creates a wrapper for an *IList* collection. | `static ArrayList Adapter(`<br>`  IList list)` |
| *Add*<br>This method adds an element to the end of the *ArrayList* collection. | `virtual int Add(object value)` |

**Table 5-6 ArrayList Members (Continued)**

| Member Name | Syntax |
|---|---|
| *AddRange*<br>This method adds a range of elements to the *ArrayList* collection. The elements are input from an *ICollection* type, such as a regular array. | `virtual void AddRange(`<br>`  ICollection elements)` |
| *BinarySearch*<br>This method performs a binary search for a specific value in a sorted array. | `virtual int BinarySearch(`<br>`  object value)`<br><br>`virtual int BinarySearch(`<br>`  object value,`<br>`  IComparer comparer)`<br><br>`virtual int BinarySearch(`<br>`  int index,`<br>`  int count,`<br>`  object value,`<br>`  IComparer comparer)` |
| *Capacity*<br>This property gets or sets the number of properties allowed in the collection. The default capacity is 16 elements. Capacity is different from the number of elements. The capacity is automatically increased as elements are added. When the number of elements exceeds the current capacity, the capacity doubles. Capacity is for better memory management of elements in the collection. | `virtual int Capacity {`<br>`get; set}` |
| *Clear*<br>This method removes all the elements of the collection. | `virtual void Clear()` |
| *Contains*<br>This method returns *true* if the specified item is found in a collection. If the item is not found, *false* is returned. | `virtual bool Contains(object item)` |
| *Count*<br>This property returns the number of elements in the collection. | `virtual int Count{`<br>`  get; }` |
| *FixedSize*<br>This method creates a wrapper for an *ArrayList* or *IList* collection, in which elements cannot be added or removed. | `static ArrayList FixedSize(`<br>`  ArrayList sourceArray)`<br><br>`static IList FixedSize(`<br>`  IList sourceList)` |
| *GetRange*<br>This method returns a span of elements from the current array. The result is stored in a destination *ArrayList*. | `virtual ArrayList GetRange(`<br>`  int index, int count)` |

**Table 5-6 ArrayList Members (Continued)**

| Member Name | Syntax |
|---|---|
| *IndexOf*<br>This method returns the index of the first matching element in the collection. | `virtual int IndexOf(`<br>`  object value)`<br><br>`virtual int IndexOf(object value,`<br>`  int startIndex)`<br><br>`virtual int IndexOf(object value,`<br>`  int startIndex,`<br>`  int count)` |
| *Insert*<br>This method inserts an element into the collection at the specified index. | `virtual void Insert(int index,`<br>`  object value)` |
| *InsertRange*<br>This method inserts multiple elements into the collection at the specified index. | `virtual void InsertRange(`<br>`  int index,`<br>`  ICollection sourceCollection)` |
| *IsFixedSize*<br>This property returns *true* if the collection is fixed-length. Otherwise, the property returns *false*. | `virtual bool IsFixedSize{`<br>`  get;}` |
| *IsReadOnly*<br>This property returns *true* if the collection is read-only. Otherwise, the property returns *false*. | `virtual bool IsReadOnly{`<br>`  get;}` |
| *Item*<br>This property gets or sets the element at the index. | `virtual object this[int index] {`<br>`  get;set;}` |
| *LastIndexOf*<br>This method returns the index of the last matching element in the collection. | `virtual int LastIndex(`<br>`  object value)`<br><br>`virtual int LastIndexOf(object value,`<br>`  int startIndex)`<br><br>`virtual int LastIndexOf(object value,`<br>`  int startIndex,`<br>`  int count)` |
| *ReadOnly*<br>This method creates a read-only wrapper for an *IList* object. | `static ArrayList ReadOnly(`<br>`  ArrayList sourceArray)`<br><br>`static IList ReadOnly(`<br>`  IList sourceList)` |
| *Remove*<br>This method removes the first element in the collection that matches the value. | `virtual void Remove(`<br>`  object value)` |

**Table 5-6 ArrayList Members (Continued)**

| Member Name | Syntax |
|---|---|
| *RemoveAt*<br>This method removes the element at the index from the collection. | `virtual void RemoveAt(`<br>`  int index)` |
| *RemoveRange*<br>This method removes a range of elements from a collection. | `virtual void RemoveRange(`<br>`  int index,`<br>`  int count)` |
| *Repeat*<br>This method returns an *ArrayList* with each element initialized to the same value. Count is the number of times to replicate the value. | `static ArrayList Repeat(`<br>`  object value,`<br>`  int count)` |
| *Reverse*<br>This method reverses the order of elements in the collection. | `virtual void Reverse()`<br><br>`virtual void Reverse(`<br>`  int beginIndex,`<br>`  int endingIndex)` |
| *SetRange*<br>This method copies elements from a collection into the same elements in the *ArrayList* collection. | `virtual void SetRange(`<br>`  int index,`<br>`  ICollection sourceCollection)` |
| *Sort*<br>This method sorts an *ArrayList*. | `virtual void Sort()`<br><br>`virtual void Sort(`<br>`  IComparer comparer}`<br><br>`virtual void Sort(int index,`<br>`  int count,`<br>`  IComparer comparer)` |
| *Synchronized*<br>This method returns a thread-safe wrapper of an *ArrayList* or *IList* object. | `static ArrayList Synchronized(`<br>`  ArrayList sourceArray)`<br><br>`static IList Synchronized(`<br>`  IList sourceList)` |
| *ToArray*<br>This method copies elements from the current array into a new collection. | `virtual object [] ToArray()`<br><br>`virtual Array ToArray(Type type)` |
| *TrimToSize*<br>This method sets the capacity to the number of elements in the collection. | `virtual void TrimToSize()` |
| *IEnumerable* members | *GetEnumerator* |
| *ICloneable* members | *Clone* |
| *ICollection* members | *CopyTo*, *Count*, *IsSynchronized*, and *SyncRoot* |

The following code uses various *ArrayList* methods and properties. It creates a new *ArrayList*, which is then initialized with command-line parameters. The *Add* method is called to add elements to the *ArrayList*. The *ArrayList* is then sorted and cloned. Then the values at the elements of the cloned *ArrayList* are doubled. Afterward, the cloned *ArrayList* is enumerated and every element is displayed.

```
using System;
using System.Collections;

namespace Donis.CSharpBook{
    public class Starter{
        public static void Main(string [] argv){

            ArrayList al1=new ArrayList();
            foreach(string arg in argv) {
                al1.Add(int.Parse(arg));
            }
            al1.Sort();
            ArrayList al2=(ArrayList)al1.Clone();
            for(int count=0;count<al2.Count;++count) {
                al2[count]=((int)al2[count])*2;
            }
            foreach(int number in al2) {
                Console.WriteLine(number);

            }
         }
    }
}
```

## BitArray Collection

The *BitArray* collection is a composite of bit values. Bit values are 1 and 0, where 1 is *true* and 0 *false*. This collection provides an efficient means of storing and retrieving bit values.

Table 5-7 list the *BitArray*-specific methods and properties. The static members of the *BitArray* are thread-safe, whereas instance members are not.

**Table 5-7 BitArray Members**

| Member Name | Syntax |
|---|---|
| *Constructor*<br>The *BitArray* constructor is overloaded. These are some of the overloaded constructors. | `BitArray(bool [] bits)`<br>`BitArray(int [] bits)`<br>`BitArray(int count, bool default)` |
| *And*<br>This method performs a bitwise *And* on the current and *BitArray* parameter. The result is placed in the returned *BitArray*. | `BitArray And(BitArray value)` |

**Table 5-7 BitArray Members (Continued)**

| Member Name | Syntax |
|---|---|
| *Get*<br>This method returns a specific bit in the *BitArray* collection. | `bool Get(int index)` |
| *IsReadOnly*<br>This property returns *true* if the collection is read-only. Otherwise, the property returns *false*. | `virtual bool IsReadOnly{ get;}` |
| *Item*<br>This property gets or sets the bit at the index. | `virtual object this[int index] { get;set;}` |
| *Length*<br>This property gets or sets the number of bits in the collection. | `public int Length{ get; set; }` |
| *Not*<br>This method negates the bits of the *BitArray* collection. The result is placed in the returned *BitArray*. | `BitArray Not()` |
| *Or*<br>This method performs a bitwise *Or* on the current and *BitArray* parameter. The result is placed in the returned *BitArray*. | `BitArray Or(BitArray value)` |
| *Set*<br>This method sets a specific bit in the collection. | `Void Set(int index, bool value)` |
| *SetAll*<br>This method sets all the bits of the collection to *true* or *false*. | `void SetAll(bool value)` |
| *Xor*<br>This method performs an exclusive *OR* on the current collection and the *BitArray* parameter. | `BitArray Xor( BitArray value)` |
| *IEnumerable* members | *GetEnumerator* |
| *ICloneable* members | *Clone* |
| *ICollection* members | *CopyTo*, *Count*, *IsSynchronized*, and *SyncRoot* |

The following code demonstrates the *BitArray* collection. The *Employee* class contains a *BitArray* collection that tracks employee enrollment in various programs, such as the health plan and credit union. This is convenient because enrollment is either *true* or *false* and never maybe. In the *Employee* class, properties are provided to set and get enrollment in various programs.

```
using System;
using System.Collections;

namespace Donis.CSharpBook{
    public class Starter{
        public static void Main(){
            Employee ben=new Employee();
            ben.InProfitSharing=false;
```

```
        ben.InHealthPlan=false;
        Employee valerie=new Employee();
        valerie.InProfitSharing=false;
        Participation("Ben", ben);
        Participation("Valerie", valerie);
    }

    public static void Participation(string name, Employee person) {
        Console.WriteLine(name+":");
        if(person.InProfitSharing) {
            Console.WriteLine("   Participating in"+
                " Profit Sharing");
        }
        if(person.InHealthPlan) {
            Console.WriteLine("   Participating in"+
                " Health Plan");
        }
        if(person.InCreditUnion) {
            Console.WriteLine("   Participating in"+
                " Credit Union");
        }
    }
}

public class Employee {

    public Employee() {
        eflags.SetAll(true);
    }

    private BitArray eflags=new BitArray(3);

    public bool InProfitSharing{
        set {
            eflags.Set(0, value);
        }
        get {
            return eflags.Get(0);
        }
    }

    public bool InHealthPlan{
        set {
            eflags.Set(1, value);
        }
        get {
            return eflags.Get(1);
        }
    }

    public bool InCreditUnion{
        set {
            eflags.Set(2, value);
        }
        get {
```

```
                return eflags.Get(2);
            }
        }
    }
}
```

## Hashtable Collection

The *Hashtable* collection is a collection of key/value pairs. Entries in this collection are instances of the *DictionaryEntry* type. *DictionaryEntry* types have a *Key* and *Value* property to get and set keys and values.

In addition to the standard collection interfaces, the *Hashtable* collection implements the *IDictionary*, *ISerializable*, and *IDeserializationCallback* interfaces.

The entries are stored and retrieved in order based on a hash code of the key.

Table 5-8 lists the members of the *Hashtable* collection.

**Table 5-8 Hashtable Members**

| Member Name | Syntax |
|---|---|
| *Constructor*<br>The *Hashtable* constructor is overloaded. These syntaxes are some of the overloaded constructors. | `Hashtable()`<br>`Hashtable(int capacity)`<br>`Hashtable(int capacity, float loadFactor)` |
| *Add*<br>This method adds an element to the collection. | `virtual void Add(object key, object value)` |
| *Contains*<br>This method returns *true* if the key is found in the collection. If the key is not present, *false* is returned. | `virtual bool Contains(object key)` |
| *ContainsKey*<br>This method returns *true* if the key is found in the collection. If the key is not present, *false* is returned. Identical to the *Contains* method. | `virtual bool ContainsKey(object key)` |
| *ContainsValue*<br>This method returns *true* if the value is found in the collection. If the value is not present, *false* is returned. | `virtual bool ContainsValue(object value)` |
| *EqualityComparer* | `IEqualityComparer EqualityComparer {get;}.` |
| *GetHash*<br>This method returns the hash code for the specified key. | `virtual int GetHash(object key)` |

**Table 5-8 Hashtable Members (Continued)**

| Member Name | Syntax |
|---|---|
| *GetObjectData*<br>This is the method implemented to serialize the collection. | `virtual void GetObjectData(`<br>`  SerializationInfo info,`<br>`  StreamingContext context)` |
| *IsFixedSize*<br>This property returns *true* if the collection is fixed size. Otherwise, *false* is returned. | `virtual bool IsFixedSize{`<br>`  get;}` |
| *IsReadOnly*<br>This property returns *true* if the collection is read-only. Otherwise, *false* is returned. | `virtual bool IsReadOnly{`<br>`  get;}` |
| *IsSynchronized*<br>This property returns *true* if the collection is synchronized. | `virtual bool IsSynchronized{`<br>`  get;}` |
| *Item*<br>This property gets or sets a value related to a key. | `virtual object this[object key]{`<br>`  get; set;}` |
| *KeyEquals*<br>This method compares a key to a value. If equal, *true* is returned. Otherwise, *false* is returned. This method is primarily used to compare two keys. | `virtual bool KeyEquals(`<br>`  object item,`<br>`  object key)` |
| *Keys*<br>Returns a collection that contains the keys of the *Hashtable*. | `virtual ICollection Keys{`<br>`  get;}` |
| *OnDeserialization*<br>This method is called when deserialization is completed. | `virtual void OnDeserialization(`<br>`object sender)` |
| *Remove*<br>This method removes an element with the specified key from the collection. | `virtual void Remove(`<br>`  object key)` |
| *Synchronized*<br>This method returns a thread-safe wrapper of the collection. | `static Hashtable Synchronized(`<br>`  Hashtable sourceTable)` |
| *Values*<br>Returns a collection that has the values of the *Hashtable*. | `virtual ICollection values{`<br>`  get;}` |
| *IEnumerable* members | *GetEnumerator* |
| *ICloneable* members | *Clone* |
| *ICollection* members | *CopyTo*, *Count*, *IsSynchronized*, and *SyncRoot* |

*Hashtable.GetEnumerator* implements *IDictionary.GetEnumerator*, which returns an *IDictionary Enumeator*. *IDictionaryEnumerator* implements the *IEnumerator* interface. It also adds three properties: *Entry*, *Key*, and *Value*.

The following code extends the previous sample code for the *BitArray* collection. The program creates a *Hashtable*, in which employee identifiers are the keys. The values associated with the keys are instances of the *Employee* type.

```
using System;
using System.Collections;

namespace Donis.CSharpBook{
    public class Starter{
        public static void Main(){
            Hashtable employees=new Hashtable();
            employees.Add("A100", new Employee(
                "Ben", true, false, true));
            employees.Add("V100", new Employee(
                "Valerie", false, false, true));
            Participation((Employee) employees["A100"]);
            Participation((Employee) employees["V100"]);
        }

        public static void Participation(Employee person) {
            Console.WriteLine(person.Name+":");
            if(person.InProfitSharing) {
                Console.WriteLine("   Participating in"+
                    " Profit Sharing");
            }
            if(person.InHealthPlan) {
                Console.WriteLine("   Participating in"+
                    " Health Plan");
            }
            if(person.InCreditUnion) {
                Console.WriteLine("   Participating in"+
                    " Credit Union");
            }
        }
    }

    public class Employee {

        public Employee(string emplName) {
            propName=emplName;
            eflags.SetAll(true);
        }

        public Employee(string emplName,
                        bool profitSharing,
                        bool healthPlan,
                        bool creditUnion) {
            propName=emplName;
            InProfitSharing=profitSharing;
            InHealthPlan=healthPlan;
```

```
            InCreditUnion=creditUnion;
        }

        private BitArray eflags=new BitArray(3);

        public bool InProfitSharing{
            set {
                eflags.Set(0, value);
            }
            get {
                return eflags.Get(0);
            }
        }

        public bool InHealthPlan{
            set {
                eflags.Set(1, value);
            }
            get {
                return eflags.Get(1);
            }
        }

        public bool InCreditUnion{
            set {
                eflags.Set(2, value);
            }
            get {
                return eflags.Get(2);
            }
        }

        private string propName;
        public string Name {
            get {
                return propName;
            }
        }
    }
}
```

This is sample code of the *IDictionaryEnumerator* enumerator:

```
using System;
using System.Collections;

namespace Donis.CSharpBook{
    public class Starter{
        public static void Main(){
            Hashtable zHash=new Hashtable();
            zHash.Add("one", 1);
            zHash.Add("two", 2);
            zHash.Add("three", 3);
            zHash.Add("four", 4);
            IDictionaryEnumerator e=
```

```
            zHash.GetEnumerator();
        while(e.MoveNext()){
            Console.WriteLine(
                "{0} {1}",
                e.Key, e.Value);
        }
    }
  }
}
```

## Queue Collection

*Queue* collections abstract FIFO data structures. The initial capacity is 32 elements. *Queue* collections are ideal for messaging components.

Table 5-9 lists the member of the *Queue* collection.

**Table 5-9 Queue Members**

| Member Name | Syntax |
|---|---|
| *Constructor* | `public Queue()`<br>`public Queue(`<br>`  ICollection sourceCollection)`<br><br>`public Queue(int capacity)`<br><br>`public Queue(int capacity,`<br>`  float factor)` |
| *Clear*<br>This method removes all the elements of the collection. | `virtual void Clear()` |
| *Contains*<br>This method returns *true* if the specified value is found in the collection. If the value is not found, *false* is returned. | `virtual bool Contains(object value)` |
| *Dequeue*<br>This method removes and returns the first element of the queue. | `virtual object Dequeue()` |
| *Enqueue*<br>This method adds an element to the queue. | `virtual void Enqueue(`<br>`  object element)` |
| *Peek*<br>This method returns the first element of the queue without removing it. | `virtual object Peek()` |
| *Synchronized*<br>This method returns a thread-safe wrapper for a queue object. | `static Queue Synchronized(`<br>`  Queue sourceQueue)` |

**Table 5-9 Queue Members (Continued)**

| Member Name | Syntax |
|---|---|
| *ToArray*<br>This method creates a new array initialized with the elements of the queue. | `virtual object[] ToArray()` |
| *TrimToSize*<br>This method sets the capacity to the number of elements in the collection. | `virtual void TrimToSize()` |
| *IEnumerable* members | *GetEnumerator* |
| *ICloneable* members | *Clone* |
| *ICollection* members | *CopyTo*, *Count*, *IsSynchronized*, and *SyncRoot* |

This is sample code of the *Queue* collection. Customers are added to the queue and then displayed.

```
using System;
using System.Collections;

namespace Donis.CSharpBook{
    public class Starter{
        public static void Main(){
            Queue waiting=new Queue();
            waiting.Enqueue(new Customer("Bob"));
            waiting.Enqueue(new Customer("Ted"));
            waiting.Enqueue(new Customer("Kim"));
            waiting.Enqueue(new Customer("Sam"));

            while(waiting.Count!= 0) {
                Customer cust=
                    (Customer) waiting.Dequeue();
                Console.WriteLine(cust.Name);
            }
        }

        public class Customer{
            public Customer(string cName) {
                propName=cName;
            }

            private string propName;
            public string Name {
                get {
                    return propName;
                }
            }
        }
    }
}
```

## SortedList

The *SortedList* collection is a combination of key/value entries and an *ArrayList* collection, where the collection is sorted by the key. The collection is accessible per the key or an index.

Table 5-10 include the members of the *SortedList* collection.

**Table 5-10 SortedList Members**

| Member Name | Syntax |
|---|---|
| *Constructor*<br>The *SortedList* constructor is overloaded. These are some of the overloaded constructors. | `SortedList()`<br>`SortedList(IComparer comparer)`<br>`SortedList(`<br>`  IDictionary sourceCollection)` |
| *Add*<br>This method adds an element to the collection. | `virtual void Add(object key,`<br>`  object value)` |
| *Capacity*<br>This property gets or sets the capacity of the collection. | `virtual int Capacity{`<br>`  get; set;}` |
| *Clear*<br>This method removes all the elements of the collection. | `virtual void Clear()` |
| *Contains*<br>This method returns *true* if the specified value is found in the collection. If the value is not found, *false* is returned. | `virtual bool Contains(object value)` |
| *ContainsKey*<br>This method returns *true* if the key is found in the collection. If the key is not present, *false* is returned. Identical to the *Contains* method. | `virtual bool ContainsKey(`<br>`  object key)` |
| *ContainsValue*<br>This method returns *true* if the value is found in the collection. If the value is not present, *false* is returned. | `virtual bool ContainsValue(`<br>`  object value)` |
| *GetByIndex*<br>This method returns the value at the index. | `virtual object GetByIndex(`<br>`  int index)` |
| *GetKey*<br>This method returns the key at the specified index. | `virtual object GetKey(`<br>`  int index)` |
| *GetKeyList*<br>This method returns all the keys in the collection. | `virtual IList GetKeyList()` |
| *GetValueList*<br>This method returns all the values of the *SortedList* in a new collection. | `virtual IList GetValueList()` |

**Table 5-10 SortedList Members (Continued)**

| Member Name | Syntax |
|---|---|
| *IndexOfKey*<br>This method returns the index of a key found in the collection. | `virtual int IndexOfKey(`<br>`  object key)` |
| *IndexOfValue*<br>This method returns the index to the first instance of this value in the collection. | `virtual int IndexOfValue(`<br>`  object value)` |
| *IsFixedSize*<br>This property returns *true* if the collection is fixed size. Otherwise, *false* is returned. | `virtual bool IsFixedSize{`<br>`  get;}` |
| *IsReadOnly*<br>This property returns *true* if the collection is read-only. Otherwise, *false* is returned. | `virtual bool IsReadOnly{`<br>`  get;}` |
| *Item*<br>This property gets or sets the value of this key. | `virtual object this[object key]`<br>`  {get; set;}` |
| *Keys*<br>This property returns the keys of the *SortedList* collection. | `public virtual ICollection Keys{`<br>`  get;}` |
| *Remove*<br>This method removes an element, which is identified by the key, from the collection. | `virtual void Remove(`<br>`  object key)` |
| *RemoveAt*<br>This method removes an element at the specific index. | `virtual void RemoveAt(`<br>`  int index)` |
| *SetByIndex*<br>This method set the value of the element at the specified index. | `virtual void SetByIndex(`<br>`  int index, object value)` |
| *Synchronized*<br>This method returns a thread-safe wrapper for a queue object. | `static SortedList Synchronized(`<br>`  SortedList sourceList)` |
| *TrimToSize*<br>This method trims the capacity to the actual number of elements in the collection. | `virtual void TrimToSize()` |
| *Values*<br>This property returns the values of the collection. | `virtual ICollection Values{`<br>`  get;}` |
| *IEnumerable* members | *GetEnumerator* |
| *ICloneable* members | *Clone* |
| *ICollection* members | *CopyTo*, *Count*, *IsSynchronized*, and *SyncRoot* |

This following program is an application that tracks auto repair tickets. Each ticket, which is an instance of the *AutoRepairTicket*, is added to a sorted list. The key is the customer name. The value is the actual ticket. After populating the *SortedList* type, the *CustomerReport* method lists the open tickets.

```
using System;
using System.Collections;

namespace Donis.CSharpBook{
    public class Starter{
        public static void Main(){
            SortedList tickets=new SortedList();
            AutoRepairTicket ticket=NewTicket("Ben");
            tickets.Add(ticket.Name, ticket);
            ticket=NewTicket("Donis");
            tickets.Add(ticket.Name, ticket);
            ticket=NewTicket("Adam");
            tickets.Add(ticket.Name, ticket);
            CustomerReport(tickets);
        }

        public static AutoRepairTicket NewTicket(
                string customerName) {
            return new AutoRepairTicket(customerName,
                DateTime.Now);
        }

        public static void CustomerReport(SortedList list) {
            foreach(DictionaryEntry entry in list) {
                int nextTag=((AutoRepairTicket) entry.Value).Tag;
                string nextTime=((AutoRepairTicket)
                    entry.Value).Time.ToShortTimeString();
                Console.WriteLine("Customer: {0} Ticket: {1} Time: {2}",
                    entry.Key, nextTag, nextTime);
            }
        }

    }

    public class AutoRepairTicket{
        public AutoRepairTicket(string customerName,
                DateTime ticketTime) {
            propName=customerName;
            propTime=ticketTime;
            propTag=++count;
        }

        private string propName;
        public string Name {
            get {
                return propName;
            }
        }
```

```
        private DateTime propTime;
        public DateTime Time {
            get {
                return propTime;
            }
        }

        private int propTag;
        public int Tag {
            get {
                return propTag;
            }
        }

        private static int count=1000;
    }
}
```

## Stack Collection

*Stack* collections abstract LIFO data structures in which the initial capacity is 32 elements.

Table 5-11 lists the member of the *Stack* collection.

**Table 5-11 Stack Members**

| Member Name | Syntax |
|---|---|
| *Clear*<br>This method removes all the elements of the collection. | `virtual void Clear()` |
| *Contains*<br>This method returns *true* if the specified value is found in the collection. If the value is not found, *false* is returned. | `virtual bool Contains(object value)` |
| *Peek*<br>The *Peek* method previews the last element on the stack. The element is returned without removal from the stack. | `virtual object Peek()` |
| *Pop*<br>This method returns and removes the top element of the stack. | `virtual object Pop()` |
| *Push*<br>This method pushes another element on the stack. | `virtual void Push(object obj)` |
| *Synchronized*<br>This method returns a thread-safe wrapper for the *Stack* collection. | `static Stack Synchronized(`<br>`  Stack sourceStack)` |
| *ToArray*<br>This method returns the *Stack* collection as a regular array. | `virtual object[] ToArray()` |

The following code adds numbers to a *Stack* collection. The values of the collection are then enumerated and displayed.

```
using System;
using System.Collections;

namespace Donis.CSharpBook{
    public class Starter{
        public static void Main(){
            Stack numbers=new Stack(
                new int [] {1,2,3,4,5,6});
            int total=numbers.Count;
    for(int count=0;count<total;++count) {
                Console.WriteLine(numbers.Pop());
            }
        }
    }
}
```

## Specialized Collections

In addition to the common collections that most developers use, the .NET FCL offers specialized collections. These collections are found in the *System.Collections.Specialized* namespace. Although these collections are used infrequently, they are valuable in certain circumstances.

Table 5-12 lists the specialized collections.

**Table 5-12 Specialized Collections**

| Member Name | Description |
|---|---|
| *BitVector32* | This is an array of 32 bits. It is similar to a *BitArray*, but limited to 32 bits. Because of this array's refined use, *BitVector32* structures are more efficient then a *BitArray* collection. |
| *HybridDictionary* | This collection is a combination of a *ListDictionary* and *Hashtable*. It operates as a *ListDictionary* when containing a small number of elements. For optimum performance, the collection switches to a *Hashtable* as the elements increased. *ListDictionary* of 10 elements or fewer is recommended. |
| *NameValueCollection* | This is a collection of keys and values, in which both the key and value are strings. The collection is accessible via an index or key. A key can refer to multiple values. |
| *OrderedDictionary* | Not documented at the time of this book. |
| *StringCollection* | This is a collection of strings. |
| *StringDictionary* | This is a combination of the *Hashtable* and *StringCollection* collections, in which both the key and value are strings. |

# Generics

The next chapter is about Generics, which improves upon the performance and other factors related to nongeneric collections. The collections reviewed in this chapter manipulate object types. For this reason, when populating collections with value types, boxing is required. Excessive boxing is both a performance and memory issue. Another problem is type-safeness. You often cast elements that are object types to a specific type. The cast is not type-safe and can cause run-time errors when done incorrectly.

Generics types are classes with type parameters. The type parameter acts as a placeholder for a future type. When a generic type is declared, type arguments are substituted for the type parameters. The type argument is the actual type. At that time, the generic is type-specific, which resolves many of the problems of a nongeneric type. Generic methods, like generic types, have type parameters. However, the type argument can be inferred from the way the method is called.

A complete explanation of generic types and methods follows in Chapter 6.