Chapter 5

Arrays and Collections

In this chapter:

The Microsoft .NET Framework doesn't merely include classes for managing system objects, such as files, directories, processes, and threads. It also exposes objects, such as complex data structures (queues, stacks, lists, and hash tables), that help developers organize information and solve recurring problems. Many real-world applications use arrays and collections, and the .NET Framework support for arrays and collection-like objects is really outstanding. It can take you a while to get familiar with the many possibilities that the Common Language Runtime (CLR) offers, but this effort pays off nicely at coding time.

Arrays and collections have become even richer and more powerful in .NET Framework version 2.0 with the introduction of generics, both because many types have been extended with generics methods and because you can create strong-typed collections much more easily in this new version of the framework.

Note To avoid long lines, code samples in this chapter assume that the following using statements are used at the top of each source file:

```
using System.Collections;
using System.Collections.Generic;
using System.Collections.ObjectModel;
using System.Collections.Specialized;
using System.Diagnostics;
using System.IO;
using System.Text.RegularExpressions;
```

The Array Type

By default, .NET arrays have a zero-based index. One-dimensional arrays with a zero lower index are known as *SZArrays* or *vectors* and are the fastest type of arrays available to developers. .NET also supports arrays with a different lower index, but they aren't CLS-compliant, aren't very efficient, and aren't recommended. In practice, you never need an array with a different lower index, and I won't cover them in this book.

The Array class constructor has a protected scope, so you can't directly use the new keyword with this class. This isn't a problem because you create an array using the standard Microsoft Visual C# syntax and you can even use initializers:

```
// An array initialized with the powers of 2
int[] intArr = new int[]{1, 2, 4, 8, 16, 32, 64, 128, 256, 512};
// Shortened syntax
int[] intArr = {1, 2, 4, 8, 16, 32, 64, 128, 256, 512};
// Noninitialized (empty) two-dimensional array
long[,] lngArr;
```

You can also create an array and initialize it on the fly, which is sometimes useful for passing an argument or assigning a property that takes an array without having to create a temporary array. To see why this feature can be useful, consider the following code:

```
// Create a temporary array.
int[] tmp = {2, 5, 9, 13};
// The obj.ValueArray property takes an array of Int32.
obj.ValueArray = tmp;
// Clear the temporary variable.
tmp = null;
```

The ability to create and initialize an array in a single statement can make the code more concise:

```
obj.ValueArray = new int[] {2, 5, 9, 13};
```

You get an error if you access a null array, which is an array variable that hasn't been initialized yet. You can test this condition using a plain == operator:

```
if ( lngArr == null )
{
   lngArr = new long[10,20];
}
```

You can query an array for its rank (that is, the number of dimensions) by using its Rank property, and you can query the total number of its elements by means of its Length property:

```
// …(Continuing the first example in this chapter)…
int res = lngArr.Rank;                       // => 2
// lngArr has 10*20 elements.
res = lngArr.Length;                         // => 200
```

Starting with version 1.1, the .NET Framework supports 64-bit array indexes, so an array index can also be a long value. To support these huge arrays, the Array class has been expanded with a LongLength property that returns the number of elements as an Int64 value.

The GetLength method returns the number of elements along a given dimension, whereas GetLowerBound and GetUpperBound return the lowest and highest indexes along the specified dimension. (For all the arrays you can create in C#, the GetLowerBound method returns 0.) As usual in the .NET Framework, the dimension number is zero-based:

```
// …(Continuing previous example)…
res = lngArr.GetLength(0);                  // => 11
res = lngArr.GetLowerBound(1);              // => 0
res = lngArr.GetUpperBound(1);              // => 20
```

You can visit all the elements of an array using a single foreach loop and a strongly typed variable. This technique also works with multidimensional arrays, so you can process all the elements in a two-dimensional array with just one loop:

```
string[] strArr = {{"00", "01", "02"}, {"10", "11", "12"}};
foreach (string s in strArr)
{
    Console.Write(s + ",");       // => 00,01,02,10,11,12
}
```

Notice that a foreach loop on a multidimensional array visits array elements in a row-wise order (all the elements in the first row, then all the elements in the second row, and so on). Pay attention when migrating legacy applications because in most languages that preceded .NET this loop worked in column-wise order.

The Array class supports the ICloneable interface, so you can create a shallow copy of an array using the Clone instance method. (See the section titled "The ICloneable Interface" in Chapter 3, "Interfaces," for a discussion about shallow and deep copy operations.)

```
string[,] arr2 = (string[,]) strArr.Clone();
```

The CopyTo method enables you to copy a one-dimensional array to another one-dimensional array; you decide the starting index in the destination array:

```
// Create and initialize an array (10 elements).
int[] sourceArr = {1, 2, 3, 5, 7, 11, 13, 17, 19, 23};
// Create the destination array (must be same size or larger).
int[] destArr = new int[20];
// Copy the source array into the second half of the destination array.
sourceArr.CopyTo(destArr, 10);
```

Sorting Elements

The Array class offers several static methods for processing arrays quickly and easily. In Chapter 3, you learned that you can sort arrays of objects using an arbitrary group of keys by means of the Array.Sort method and the IComparable and IComparer interfaces. But the Array.Sort method is even more flexible than what you've seen so far. For example, it can sort just a portion of an array:

```
// Sort only elements [10,99] of the targetArray.
// Second argument is starting index; last argument is length of the subarray.
Array.Sort(targetArray, 10, 90);
```

You can also sort an array of values using another array that holds the sorting keys, which enables you to sort arrays of structures or objects even if they don't implement the IComparable interface. To see how this overloaded version of the Sort method works, let's start defining a structure:

```
public struct Employee
{
   public string FirstName;
   public string LastName;
   public DateTime HireDate;

   public Employee(string firstName, string lastName, DateTime hireDate)
   {
      this.FirstName = firstName;
      this.LastName = lastName;
      this.HireDate = hireDate;
   }

   // A function to display an element's properties easily
   public string Description()
   {
      return string.Format("{0} {1} (hired on {2})", FirstName, LastName,
         HireDate.ToShortDateString());
   }
}
```

The following code creates a main array of Employee structures, creates an auxiliary key array that holds the hiring date of each employee, and finally sorts the main array using the auxiliary array:

```
// Create a test array.
Employee[] employees = { new Employee("John", "Evans", new DateTime(2001, 3, 1)),
   new Employee("Robert", "Zare", new DateTime(2000, 8, 12)),
   new Employee("Ann", "Beebe", new DateTime(1999, 11, 1)) };
// Create a parallel array of hiring dates.
DateTime[] hireDates = new DateTime[employees.Length];
for ( int i = 0; i < employees.Length; i++ )
{
   hireDates[i] = employees[i].HireDate;
}
// Sort the array of Employees using HireDates to provide the keys.
Array.Sort(hireDates, employees);
// Prove that the array is sorted on the HireDate field.
foreach (Employee em in employees)
{
   Console.WriteLine(em.Description());
}
```

Interestingly, the key array is sorted as well, so you don't need to initialize it again when you add another element to the main array:

```
// Add a fourth employee.
Array.Resize<Employee>(ref employees, 4);
```

```
employees[3] = new Employee("Chris", "Cannon", new DateTime(2000, 5, 9));
// Extend the key array as well, no need to reinitialize it.
Array.Resize<DateTime>(ref hireDates, 4);
hireDates[3] = employees[3].HireDate;
// Re-sort the new, larger array.
Array.Sort(hireDates, employees);
```

(Read on for the description of the Array.Resize<T> generic method.) An overloaded version of the Sort method enables you to sort a portion of an array of values for which you provide an array of keys. This is especially useful when you start with a large array that you fill only partially:

```
// Create a test array with a lot of room.
Employee[] employees = new Employee[1000];
// Initialize only its first four elements.
…
// Sort only the portion actually used.
Array.Sort(hireDates, employees, 0, 4);
```

All the versions of the Array.Sort method that you've seen so far can take an additional IComparer object, which dictates how the array elements or keys are to be compared with one another. (See the section titled "The IComparer Interface" in Chapter 3.)

The Array.Reverse method reverses the order of elements in an array or in a portion of an array, so you can apply it immediately after a Sort method to sort in descending order:

```
// Sort an array of Int32 in reverse order.
Array.Sort(intArray);
Array.Reverse(intArray);
```

You pass the initial index and number of elements to reverse only a portion of an array:

```
// Reverse only the first 10 elements in intArray.
Array.Reverse(intArray, 0, 10);
```

You have a special case when you reverse only two elements, which is the same as swapping two consecutive elements, a frequent operation when you're working with arrays:

```
// Swap elements at indexes 5 and 6.
Array.Reverse(intArray, 5, 2);
```

Clearing, Copying, and Moving Elements

You can clear a portion of an array by using the Clear method, without a for loop:

```
// Clear elements [10,99] of an array.
Array.Clear(arr, 10, 90);
```

The Array.Copy method enables you to copy elements from a one-dimensional array to another. There are two overloaded versions for this method. The first version copies a given number of elements from the source array to the destination array:

```
int[] intArr = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int[] intArr2 = new int[20];
// Copy the entire source array into the first half of the target array.
Array.Copy(intArr, intArr2, 10);
for (int i = 0; i < 20; i++)
{
   Console.Write("{0} ", intArr2[i]);
      // => 1 2 3 4 5 6 7 8 9 10 0 0 0 0 0 0 0 0 0 0
}
```

The second version lets you decide the starting index in the source array, the starting index in the destination array (that is, the index of the first element that will be overwritten), and the number of elements to copy:

```
// Copy elements at indexes 5-9 to the end of intArr2.
Array.Copy(intArr, 5, intArr2, 15, 5);
// This is the first element that has been copied.
Console.WriteLine(intArr2[15]);                      // => 6
```

You get an exception of type ArgumentOutOfRangeException if you provide wrong values for the indexes or the destination array isn't large enough, and you get an exception of type RankException if either array has two or more dimensions.

The Copy method works correctly even when source and destination arrays have elements of different types, in which case it attempts to cast each individual source element to the corresponding element in the destination array. The actual behavior depends on many factors, though, such as whether the source or the destination is a value type or a reference type. For example, you can always copy from any array to an Object array, from an Int32 array to an Int64 array, and from a float array to a double array because they are widening conversions and can't fail. Copy throws an exception of type TypeMismatchException when you attempt a narrowing conversion between arrays of value types, even though individual elements in the source array might be successfully converted to the destination type:

```
int[] intArr3 = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
// This Copy operation succeeds even if array types are different.
long[] lngArr3 = new long[20];
Array.Copy(intArr3, lngArr3, 10);

// This Copy operation fails with ArrayTypeMismatchException.
//   (But you can carry it out with an explicit for loop.)
short[] shoArr3 = new short[20];
Array.Copy(intArr3, shoArr3, 10);
```

Conversely, if you copy from and to an array of reference type, the Array.Copy method attempts the copy operation for each element; if an InvalidCastException object is thrown for an element, the method copies neither that element nor any of the values after the one that raised the error. This behavior can cause a problem because your code now has an array that is only partially filled.

The ConstrainedCopy method, new in .NET Framework 2.0, solves the issue I just mentioned, sort of. If an exception occurs when using this method, all changes to the destination array are undone in an orderly manner, so you can never end up with an array that has been copied or converted only partially. However, the ConstrainedCopy method can't really replace the Copy method in the previous code snippet because it requires that no form of boxing, unboxing, casting, widening conversion, or narrowing conversion occurs. In practice, you should use the ConstrainedCopy method only in critical regions where an unexpected exception, including a .NET internal error, might compromise your data.

The Array.Copy method can even copy a portion of an array over itself. In this case, the Copy method performs a "smart copy" in the sense that elements are copied correctly in ascending order when you're copying to a lower index and in reverse order when you're copying to a higher index. So you can use the Copy method to delete one or more elements and fill the hole that would result by shifting all subsequent elements one or more positions toward lower indexes:

```
long[] lngArr4 = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
// Delete element at index 4.
Array.Copy(lngArr4, 5, lngArr4, 4, 5);
// Complete the delete operation by clearing the last element.
Array.Clear(lngArr4, lngArr4.Length - 1, 1);
// Now the array contains: {1, 2, 3, 4, 6, 7, 8, 9, 10, 0}
```

You can use this code as the basis for a reusable method that works with any type of array:

```
public static void ArrayDeleteElement(Array arr, int index)
{
   // This method works only with one-dimensional arrays.
   if (arr.Rank != 1)
   {
      throw new ArgumentException("Invalid rank");
   }
   // Shift elements from arr[index+1] to arr[index].
   Array.Copy(arr, index + 1, arr, index, arr.Length - index - 1);
   // Clear the last element.
   Array.Clear(arr, arr.Length - 1, 1);
}
```

Inserting an element is also easy, and again you can create a routine that works with arrays of any type:

```
public static void ArrayInsertElement(Array arr, int index, object newValue)
{
   // This method works only with one-dimensional arrays.
   if (arr.Rank != 1)
   {
      throw new ArgumentException("Invalid rank");
   }
   // Shift elements from arr[index] to arr[index+1] to make room.
   Array.Copy(arr, index, arr, index + 1, arr.Length - index - 1);
   // Assign the element using the SetValue method.
   arr.SetValue(newValue, index);
}
```

The Array class exposes the SetValue and GetValue methods to assign and read elements. You don't use these methods often in regular programming, but they turn out to be useful in methods that work with any type of array. You can also use generics to make your code even more concise, more robust, and faster:

```
public static void ArrayDeleteElement<T>(T[] arr, int index)
{
   Array.Copy(arr, index + 1, arr, index, arr.Length - index - 1);
   arr[index] = default(T);
}

public static void ArrayInsertElement<T>(T[] arr, int index, T newValue)
{
   Array.Copy(arr, index, arr, index + 1, arr.Length - index - 1);
   arr[index] = newValue;
}
```

You can also use the Copy method with multidimensional arrays, in which case the array is treated as if it were a one-dimensional array with all the rows laid down in memory, one after the other. This method works only if the source and destination arrays have the same rank, even if they can have a different number of rows and columns.

You can do some interesting tricks with the Buffer type, which exposes static methods that perform byte-by-byte operations on one-dimensional arrays. The elements in the two arrays don't need to be the same size; thus, for example, you can inspect the individual bytes of a Double array as follows:

```
// Inspecting the bytes of a Double array
double[] values = { 123, 456, 789 };           // 3 Doubles = 24 bytes
byte[] bytes = new byte[24];
Buffer.BlockCopy(values, 0, bytes, 0, 24);
foreach (byte b in bytes)
{
   Console.Write("{0} ", b);
   // => 0 0 0 0 0 192 94 64 0 0 0 0 0 128 124 64 0 0 0 0 0 168 136 64
}
```

Other methods of the Buffer type allow you to read and write individual bytes inside an array. For security reasons, the Buffer class works only with arrays of primitive types, such as Boolean, Char, and all numeric types. Arrays of other types cause an exception of type ArgumentException to be thrown.

Searching Values

The IndexOf method searches an array for a value and returns the index of the first element that matches or –1 if the search fails:

```
string[] strArr = {"Robert", "Joe", "Ann", "Chris", "Joe"};
int index = Array.IndexOf(strArr, "Ann");               // => 2
// Note that string searches are case sensitive.
index = Array.IndexOf(strArr, "ANN");                   // => -1
```

You can also specify a starting index and an optional ending index; if an ending index is omitted, the search continues until the end of the array. You can use the following approach to find all the values in the array with a given value:

```
// Search for all the occurrences of the "Joe" string.
index = Array.IndexOf(strArr, "Joe");
while ( index >= 0 )
{
   Console.WriteLine("Found at index {0}", index);
   // Search next occurrence.
   index = Array.IndexOf(strArr, "Joe", index + 1);
}
```

The LastIndexOf method is similar to IndexOf except that it returns the index of the last occurrence of the value. Because the search is backward, you must pass a start index equal to the end index:

```
// A revised version of the search loop, which searches
// from higher indexes toward the beginning of the array.
index = Array.LastIndexOf(strArr, "Joe", strArr.Length - 1);
while ( index >= 0 )
{
   Console.WriteLine("Found at index {0}", index);
   index = Array.LastIndexOf(strArr, "Joe", index - 1);
}
```

The IndexOf and LastIndexOf methods perform a linear search, so their performance degrades linearly with larger arrays. You deliver much faster code if the array is sorted and you use the BinarySearch method:

```
// Binary search on a sorted array
string[] strArr2 = {"Ann", "Chris", "Joe", "Robert", "Sam"};
index = Array.BinarySearch(strArr2, "Chris");              // => 1
```

If the binary search fails, the method returns a negative value that's the bitwise complement of the index of the first element that's larger than the value being searched for. This feature enables you to determine where the value should be inserted in the sorted array:

```
index = Array.BinarySearch(strArr2, "David");
if ( index >= 0 )
{
   Console.WriteLine("Found at index {0}", index);
}
else
{
   // Negate the result to get the index for the insertion point.
   index = ~index;
   Console.WriteLine("Not Found. Insert at index {0}", index);
      // => Not found. Insert at index 2
}
```

You can pass a start index and the length of the portion of the array in which you want to perform the search, which is useful when you're working with an array that's only partially filled:

```
index = Array.BinarySearch(strArr2, 0, 3, "Chris");         // => 1
```

Finally, both syntax forms for the BinarySearch method support an IComparer object at the end of the argument list; this argument lets you determine how array elements are to be compared. In practice, you can use the same IComparer object that you passed to the Sort method when you sorted the array.

Jagged Arrays

C# also supports arrays of arrays, that is, arrays whose elements are arrays. Arrays of arrays—also known as *jagged arrays*—are especially useful when you have a two-dimensional matrix with rows that don't have the same length. You can render this structure by using a standard two-dimensional array, but you'd have to size it to accommodate the row with the highest number of elements, which would result in wasted space. The arrays of arrays concept isn't limited to two dimensions only, and you might need three-dimensional or four-dimensional jagged arrays. Here is an example of a "triangular" matrix of strings:

```
"a00"
"a10"  "a11"
"a20"  "a21"  "a22"
"a30"  "a31"  "a32"  "a33"
```

The next code snippet shows how you can render the preceding structure as a jagged array, and then process it by expanding its rows:

```
// Initialize an array of arrays.
string[][] arr = { new string[] { "a00" },
   new string[] { "a10", "a11" },
   new string[] { "a20", "a21", "a22" },
   new string[] { "a30", "a31", "a32", "a33" } };

// Show how you can reference an element.
string elem = arr[3][1];                          // => a31
// Assign an entire row.
arr[0] = new string[] { "a00", "a01", "a02" };
// Read an element just added.
elem = arr[0][2];                                 // => a02
```

```
// Expand one of the rows.
Array.Resize<string>(ref arr[1], arr[1].Length + 2);
// Assign the new elements. (Currently they are null.)
arr[1][2] = "a12";
arr[1][3] = "a13";
// Read back one of them.
elem = arr[1][2];                                   // => a12
```

An obvious advantage of jagged arrays is that they can take less memory than regular multidimensional arrays do. Just as interesting, the JIT compiler produces code that is up to five or six times faster when accessing a jagged array than when accessing a multidimensional array. However, keep in mind that jagged arrays aren't CLS-compliant; thus, they shouldn't appear as arguments or return values in public methods.

A great way to take advantage of the higher speed of jagged arrays while continuing to use the standard multidimensional array syntax and hiding implementation details at the same time is by defining a generic type that wraps an array of arrays:

```
public class Matrix<T>
{
   private T[][] values;

   public Matrix(int rows, int cols)
   {
      values = new T[rows][];
      bounds = new int[] { rows - 1, cols - 1 };

      for (int i = 0; i < rows; i++)
      {
         T[] row = new T[cols];
      }
   }

   public T this[int row, int col]
   {
      get { return values[row][col]; }
      set { values[row][col] = value; }
   }
}
```

Using the Matrix class is almost identical to using a two-dimensional array, the only difference is in the way you create an instance of the array:

```
Matrix<double> mat = new Matrix<double>(100, 100);
mat[10, 1] = 123.45;
Console.WriteLine(mat[10, 1]);              // => 123.45
```

Because of the way the CLR optimizes jagged arrays, the Matrix class is two to three times faster than a standard two-dimensional array is, while preserving the latter's standard syntax. Can you ask for more?

Generic Methods

In version 2.0 of the .NET Framework, the Array type has been extended with several generic methods. In general, these methods offer better type safety and, in most cases, better performance. For example, consider the following code:

```
// (Microsoft Visual C# .NET 2003 code)
// Create an array with a nonzero value in the last element.
short[] arr = new short[100000];
arr[100000] = -1;
// Search for the nonzero element.
int index = Array.IndexOf(arr, -1);
```

The standard IndexOf method must work with arrays of all kinds; thus, the search it performs isn't optimized for a specific element type. More specifically, the second argument must be boxed when you pass a value type, as in this case. To solve these issues, the Array class in .NET Framework 2.0 supports the IndexOf<T> generic method:

```
index = Array.IndexOf<short>(arr, -1);
```

The generic method appears to be from 15 to 100 times faster than the standard method is, depending on how many repetitions you execute. (Remember that each time you call the standard method, a boxing operation takes place and a temporary object is created behind the scenes.) Even with a few repetitions, the generic approach is clearly to be preferred, especially when you consider that it simply requires adding a pair of angle brackets to existing C# code. Notice that there is no significant performance gain in using this method with a reference type, for example, a string array.

Interestingly, the Microsoft Visual C# 2005 compiler automatically selects the generic version of a method, if possible—therefore, most of your Visual C# .NET 2003 code will perform better if you simply recompile it under the current Microsoft Visual Studio version. This behavior is a consequence of the fact that you can drop the <...> clause in generic methods if no ambiguity ensues, as I explained in the section titled "Generic Methods" in Chapter 4, "Generics." More specifically, the compiler selects the generic version of the second argument if it matches perfectly the type of the array passed in the first argument. For example, consider this code:

```
int arr = new int[100000];
arr[99999] = -1;
// Next statement is compiled using the IndexOf<int> generic method.
index = Array.IndexOf(arr, -1);

short search = -1;
// Next statement is compiled using the standard IndexOf, and boxing occurs.
index = Array.IndexOf(arr, search);
```

This undocumented behavior can lead to a serious loss of performance in some cases. For example, consider this code:

```
long[] lngArr = new long[100000];
lngArr(99999) = -1;
index = Array.IndexOf(lngArr, -1);
```

Quite surprisingly, the last statement in this code snippet is compiled using a standard IndexOf method instead of the more efficient IndexOf<long> method that you might expect. The reason: the −1 argument is considered a 32-bit value and therefore doesn't match the array of long values passed in the first argument. You therefore must either explicitly use the generic method or force the type of the second argument, as follows:

```
// Two techniques to force the compiler to use the generic method
index = Array.IndexOf<long>(lngArr, -1);
index = Array.IndexOf(lngArr, -1L);
```

If you think that this is just a syntax detail and that you shouldn't care about which method is actually chosen by the compiler, well, think again. If you force the compiler to select the IndexOf<long> method instead of the IndexOf standard method, your code can run *almost two orders of magnitude faster*! The actual ratio depends on how many times you invoke the method and becomes apparent when this number is high enough to fire one or more garbage collections.

Note The code examples in the remaining portion of this chapter use the generic syntax to emphasize the generic nature of methods, even if in most cases the <…> clause might be dropped. Although there aren't any established guidelines in this field, I recommend that you use explicit angle brackets in all cases, both to make your code more readable and to force the compiler to use the generic version when a standard version of the same method is available.

A few other generic methods that mirror existing methods have been added, including BinarySearch, LastIndexOf, and Sort. The generic sort method can take one or two generic parameters, depending on whether you pass a parallel array of keys:

```
// Sort an array of integers.
Array.Sort<int>(arr);
// Sort an array of integers using a parallel array of string keys.
string[] keys = new string[arr.Length];
// Fill the array of keys.
…
// Sort the integer array using the parallel key array.
Array.Sort<string>, Integer)(keys, arr);
```

The Resize<T> method changes the number of elements in a one-dimensional array while preserving existing elements. I have already used this method previously in this chapter:

```
int[] arr = {0, 1, 2, 3, 4};
…
// Extend the array to contain 10 elements, but preserve existing ones.
Array.Resize<int>(ref arr, 10);
```

There is no Resize method to resize two-dimensional arrays, but it's easy to create one:

```
public static void Resize<T>(ref T[,] arr, int rows, int cols)
{
   if ( rows <= 0 || cols <= 0 )
   {
      throw new ArgumentException("Invalid new size");
   }
   T[,] newArr = new T[rows, cols];
   for ( int r = 0; r <= Math.Min(arr.GetUpperBound(0), rows); r++ )
   {
      for ( int c = 0; c <= Math.Min(arr.GetUpperBound(1), cols); c++ )
      {
         newArr[r, c] = arr[r, c];
      }
   }
   arr = newArr;
}
```

All the remaining generic methods in the Array class take a delegate as an argument and enable you to perform a given operation without writing an explicit for or foreach loop. These methods are especially useful in C# when used together with anonymous methods. In fact, most of the generic methods exposed by the Array type take a Predicate<T> delegate; such delegates point to a function that takes an argument of type T and returns a bool value, which is typically the result of a test condition on the argument. For example, consider the code that you should write to find the first array element that meets a given criterion, for example, the first number that is positive and divisible by 10:

```
int[] arr()= {1, 3, 60, 4, 30, 66, -10, 79, 10, -4};
int result = 0;
for ( int i = 0; i < arr.Length; i++ )
{
   if ( arr[i] > 0 && (arr[i] % 10) == 0 )
   {
      result = arr[i];
      break;
   }
}
if ( result == 0 )
{
   Console.WriteLine("Not found");
}
else
{
   Console.WriteLine("Result = {0}", result);     // => Result = 60
}
```

You'll probably agree that it's a lot of code for such a simple task. Now, see how elegant the code becomes when you use the Find<T> generic method together with an anonymous method to get rid of the for loop:

```
int res = Array.Find<int>(arr, delegate(int n)
                               { return n > 0 && (n % 10) == 0; });
```

There is also a FindLast<T> generic method that, as its name implies, returns the last element in the array that matches the condition:

```
int res = Array.FindLast<int>(arr, delegate(int n)
                                   { return n > 0 && (n % 10) == 0; });
```

Of course, the power of generics ensures that you can also use a similarly concise approach when looking for an element in a string array, a Double array, or an array of any type. If you simply want to check whether an element matching the condition exists, but you aren't interested in its value, you can use the new Exists generic method:

```
bool found = Array.Exists<int>(arr, delegate(int n)
                            { return n > 0 && (n % 10) == 0; });
if ( found )
{
   // The array contains at least one positive multiple of 10.
}
```

A limitation of the Find and FindLast methods is that they always return the default value of the type T if no match is found: null for strings and other reference types, zero for numbers, and so forth. In the preceding example, you know that the result—if found—is strictly positive, so a result equal to zero means that no match was found. If the zero or null value might be a valid match, however, you must opt for a different approach, based on the FindIndex<T> generic method:

```
int index = Array.FindIndex<int>(arr, delegate(int n)
                                     { return n > 0 && (n % 10) == 0; });
if ( index < 0 )
{
   Console.WriteLine("Element not found");
}
else
{
   Console.WriteLine("Element {0} found at index {1}", arr[index], index);
}
```

As you might expect, there is also a FindLastIndex<T> method that returns the last element that satisfies the condition:

```
index = Array.FindIndex<int>(arr, delegate(int n)
                                  { return n > 0 && (n % 10) == 0; });
```

Unlike the Find and FindLast methods, both the FindIndex and FindLastIndex methods expose two overloads that enable you to indicate the starting index and the number of elements to be searched. If you are interested in gathering all the elements that match the condition, you might therefore use these methods in a loop, until they return −1, as in the following code:

```
int index = -1;
List<int> list = new List<int>();
while ( true )
```

```
{
   // Find the next match; exit the loop if not found.
   index = Array.FindIndex<int>(arr, index + 1, delegate(int n)
                                             { return n > 0 && (n % 10) == 0; });
   if ( index < 0 )
   {
      break;
   }
   // Remember the match in the List collection.
   list.Add(arr(index))
}
// Convert the List to a strong-typed array.
int[] matches = list.ToArray();
Console.WriteLine("Found {0} matches", matches.Length);   // => Found 3 matches
```

Once again, you'll surely appreciate the conciseness that the FindAll<T> method gives you:

```
// This statement is equivalent to the previous code snippet.
int[] matches = Array.FindAll<int>(arr, delegate(int n)
                                        { return n > 0 && (n % 10) == 0; });
```

The TrueForAll<T> generic method enables you to quickly check whether all the elements in the array match the condition:

```
if ( Array.TrueForAll<int>(arr, delegate(int n)
                               { return n > 0 && (n % 10) == 0;}) )
{
   // All elements in the array are positive multiples of 10.
}
else
{
   // There is at least one element that isn't a positive multiple of 10.
}
```

(Note that there isn't a FalseForAll<T> generic method.) The ConvertAll<T, U> generic method provides a very powerful way to convert all the elements in an array into values of the same or different type. For example, here's how you can quickly convert all the elements in an Int32 array into their hexadecimal representation:

```
int[] arr = { 1, 3, 60, 4, 30, 66, -10, 79, 10, -4 };
string[] hexValues = Array.ConvertAll<int, string>(arr, delegate(int n)
                                                        { return n.ToString("X2");  });
```

The second argument for the ConvertAll<T, U> method must be a delegate that points to a method that takes an argument of type T and returns a value of type U. In some cases, you don't even need to define a separate method because you can use a static method of a type defined in the .NET Framework. For example, here's how you can convert a numeric array into a string array:

```
string[] arrStr = Array.ConvertAll<int, string>(arr, Convert.ToString);
```

Many math transformations can be achieved by passing a delegate that points to one of the static methods of the Math type, for example, to round or truncate a Double or a Decimal value to an integer. You can use any of such methods, provided that the method takes only one argument.

The ForEach method, the last generic method in this overview, enables you to execute a given action or method for each element in the array; its second argument is an Action<T> delegate, which must point to a void procedure that takes an argument of type T. Here's an example that outputs all the elements of an array to the console window, without an explicit loop:

```
Array.ForEach<string>(arrStr, Console.WriteLine);
```

Please notice that I am providing these examples mainly as a demonstration of the power of generic methods in the Array type. I am not suggesting that you should always prefer these methods to simpler (and more readable) for or foreach loops. As a matter of fact, an explicit loop is often faster than an anonymous method used with a generic method of the Array class, so you should never use these generic methods in time-critical code.

The System.Collections Namespace

The System.Collections namespace exposes many classes that can work as data containers, such as collections and dictionaries. You can learn the features of all these objects individually, but a smarter approach is to learn about the underlying interfaces that these classes might implement.

The ICollection, IList, and IDictionary Interfaces

All the collection classes in the .NET Framework implement the ICollection interface, which inherits from IEnumerable and defines an object that supports enumeration through a foreach loop. The ICollection interface exposes a read-only Count property and a CopyTo method, which copies the elements from the collection object to an array.

The ICollection interface defines the minimum features that a collection-like object should have. The .NET Framework exposes two more interfaces whose methods add power and flexibility to a collection object: IList and IDictionary.

Many classes in the .NET Framework implement the IList interface. This interface inherits from ICollection, and therefore from IEnumerable, and represents a collection of objects that can be individually indexed. All the implementations of the IList interface fall into three categories: read-only (the collection's elements can't be modified or deleted, nor can new elements be inserted), fixed size (existing items can be modified, but elements can't be added or removed), and variable size (items can be modified, added, and removed).

The IList interface exposes several members in addition to the Count property and the CopyTo method inherited from IEnumerable. The names of these methods are quite self-explanatory: Add appends an element to the end of the collection; Insert adds a value between two existing elements; Remove deletes an element given its value; RemoveAt deletes an element at the specified index; Clear removes all the elements in one operation. You can access an element at a given index by means of the Item property (this is also the indexer of the class) and check whether an element with a given value exists with the Contains method (which returns a bool) or the IndexOf method (which returns the index where the element is found, or −1 if the element isn't found). You'll see all these methods and properties in action when I discuss the ArrayList type.

The IDictionary interface defines a collection-like object that contains one or more (key, value) pairs, where the key can be any object. As for the IList interface, implementations of the IDictionary interface can be read-only, fixed size, or variable size.

The IDictionary interface inherits the Count and CopyTo members from ICollection and extends it using the following methods: Add(key, value) adds a new element to the collection and associates it with a key; Remove removes an element with a given key; Clear removes all elements; Contains checks whether an element with a given key exists. You can access items in an IDictionary object with the Item(key) property, which C# clients see as the type's indexer; the Keys and Values read-only properties return an array containing all the keys and all the values in the collection, respectively.

For a class that implements the ICollection, IList, or IDictionary interface, it isn't mandatory that you expose all the interface's properties and methods as public members. For example, the Array class implements IList, but the Add, Insert, and Remove members don't appear in the Array class interface because arrays have a fixed size. You get an exception if you invoke these methods after casting an array to an IList variable.

A trait that all the classes in System.Collections—except the BitArray and BitVector32 types—have in common is that they store Object values. This means that you can store any type of value inside them and even store instances of different types inside the same collection. In some cases, this feature is useful, but when used with value types these collections cause a lot of boxing activity and their performance is less than optimal. Also, you often need to cast values to a typed variable when you unbox collection elements. As explained in Chapter 4, you should use a strong-typed generic collection to achieve type safety and more efficient code.

The ArrayList Type

You can think of the ArrayList class as a hybrid between an array and a collection. For example, you can address elements by their indexes, sort and reverse them, and search a value sequentially or by means of a binary search as you do with arrays; you can append elements, insert them in a given position, or remove them as you do with collections.

The ArrayList object has an initial capacity—in practice, the number of slots in the internal structure that holds the actual values—but you don't need to worry about that because an ArrayList is automatically expanded as needed, as are all collections. However, you can optimize your code by choosing an initial capability that offers a good compromise between used memory and the overhead that occurs whenever the ArrayList object has to expand:

```
// Create an ArrayList with default initial capacity of 4 elements.
ArrayList al = new ArrayList();
// Create an ArrayList with initial capacity of 1,000 elements.
ArrayList al2 = new ArrayList(1000);
```

(Notice that the initial capacity was 16 in .NET version 1.1 but has changed to 4 in version 2.0.) The ArrayList constructor can take an ICollection object and initialize its elements accordingly. You can pass another ArrayList or just a regular array:

```
// Create an array on the fly and pass it to the ArrayList constructor.
ArrayList al3 = new ArrayList(new string[]{"one", "two", "three"});
```

You can modify the capacity at any moment to enlarge the internal array or shrink it by assigning a value to the Capacity property. However, you can't make it smaller than the current number of elements actually stored in the array (which corresponds to the value returned by the Count property):

```
// Make the ArrayList take only the memory that it strictly needs.
al.Capacity = al.Count;
// Another way to achieve the same result
al.TrimToSize();
```

When the current capacity is exceeded, the ArrayList object doubles its capacity automatically. You can't control the growth factor of an ArrayList, so you should set the Capacity property to a suitable value to avoid time-consuming memory allocations.

Another way to create an ArrayList object is by means of its static Repeat method, which enables you to specify an initial value for the specified number of elements:

```
// Create an ArrayList with 100 elements equal to an empty string.
ArrayList al4 = ArrayList.Repeat("", 100);
```

The ArrayList class fully implements the IList interface. You add elements to an ArrayList object by using the Add method (which appends the new element after the last item) or the Insert method (which inserts the new element at the specified index). You remove a specific object by passing it to the Remove method, remove the element at a given index by using the RemoveAt method, or remove all elements with the Clear method:

```
// Be sure that you start with an empty ArrayList.
al.Clear();
// Append the elements "Joe" and "Ann" at the end of the ArrayList.
al.Add("Joe");
al.Add("Ann");
// Insert "Robert" item at the beginning of the list. (Index is zero-based.)
al.Insert(0, "Robert");
// Remove "Joe" from the list.
al.Remove("Joe");
// Remove the first element of the list ("Robert" in this case).
al.RemoveAt(0);
```

The Remove method removes only the first occurrence of a given object, so you need a loop to remove all the elements with a given value. You can't simply iterate through the loop until you get an error, however, because the Remove method doesn't throw an exception if the element isn't found. Therefore, you must use one of these two approaches:

```
// Using the Contains method is concise but not very efficient.
while ( al.Contains("element to remove") )
{
   al.Remove("element to remove");
}

// A more efficient technique: loop until the Count property becomes constant.
int saveCount = 0;
while ( al.Count == saveCount )
{
   saveCount = al.Count;
   al.Remove("element to remove");
}
```

You can read and write any ArrayList element using the Item property. This property is the type's indexer, so you can omit it and deal with this object as if it were a standard zero-based array:

```
al[0] = "first element";
```

Just remember that an element in an ArrayList object is created only when you call the Add method, so you can't reference an element whose index is equal to or higher than the Array-List's Count property. As with all collections, the preferred way to iterate over all elements is through the foreach loop, even though you can surely use a standard for loop:

```
// These two loops are equivalent.
foreach ( object o in al )
{
   Console.WriteLine(o);
}

for ( int i = 0; i < al.Count; i++ )
{
   Console.WriteLine(al[i]);
}
```

A good reason for using a for loop is that the controlling variable in a foreach loop is read-only, and thus you can't use a foreach loop if you need to modify elements in the ArrayList.

The ArrayList class exposes methods that enable you to manipulate ranges of elements in one operation. The AddRange method appends to the current ArrayList object all the elements contained in another object that implements the ICollection interface. Many .NET classes other than those described in this chapter implement ICollection, such as the collection of all the items in a ListBox control and the collection of nodes in a TreeView control. The following routine takes two ArrayList objects and returns a third ArrayList that contains all the items from both arguments:

```
public static ArrayList ArrayListJoin(ArrayList al1, ArrayList al2)
{
   // Note how we avoid time-consuming reallocations.
   ArrayList res = new ArrayList(al1.Count + al2.Count);
   // Append the items in the two ArrayList arguments.
   res.AddRange(al1);
   res.AddRange(al2);
   return res;
}
```

The InsertRange method works in a similar way but enables you to insert multiple elements at any index in the current ArrayList object:

```
// Insert all the items of al2 at the beginning of al.
al.InsertRange(0, al2);
// RemoveRange deletes multiple elements in the al object:
// Delete the last four elements (assumes there are at least four elements).
al.RemoveRange(al.Count - 4, 4);
```

Adding or removing elements from the beginning or the middle of an ArrayList is an expensive operation because all the elements with higher indexes must be shifted accordingly. In general, the Add method is faster than the Insert method is and should be used if possible.

You can read or write a group of contiguous elements by means of the GetRange and SetRange methods. The former takes an initial index and a count and returns a new ArrayList that contains only the elements in the selected range; the latter takes an initial index and an ICollection object:

```
// Display only the first 10 elements to the console window.
foreach ( object o in al.GetRange(0, 10) )
{
   Console.WriteLine(o);
}
// Copy the first 20 elements from al to al2.
al2.SetRange(0, al.GetRange(0, 20));
```

You can quickly extract all the items in the ArrayList object by using the ToArray method or the CopyTo method. Both of them support one-dimensional target arrays of any compatible type, but the latter also enables you to extract a subset of ArrayList:

```
// Extract elements to an Object array (never throws an exception).
object[] objArr = al.ToArray();
// Extract elements to a String array (might throw an InvalidCastException).
string[] strArr = (string[]) al.ToArray(typeof(string));

// Same as above but uses the CopyTo method.
// (Note that the target array must be large enough.)
string[] strArr2 = new string[al.Count];
al.CopyTo(strArr2);
```

```
// Copy only items [1,2], starting at element 4 in the target array.
string[] strArr3 = {"0", "1", "2", "3", "4", "5", "6", "7", "8", "9"};
// Syntax is: CopyTo(sourceIndex, target, destIndex, count).
al.CopyTo(0, strArr3, 4, 2);
```

The ArrayList class supports other useful methods, such as Sort, Reverse, BinarySearch, Contains, IndexOf, LastIndexOf, and Reverse. I described most of these methods in the section devoted to arrays, so I won't repeat their description here.

The TrimToSize method deserves a special mention. As I explained previously, the ArrayList automatically doubles its capacity whenever it needs room for a new element. After many insertions and deletions you might end up with an ArrayList that contains many unused slots; if you don't plan to add more elements to the ArrayList, you can reclaim the unused space by means of the TrimToSize method:

```
al.TrimToSize();
```

The last feature of the ArrayList class that's worth mentioning is its Adapter and ReadOnly static methods. The Adapter method takes an IList-derived object as its only argument and creates an ArrayList wrapper around that object. In other words, instead of creating a copy of the argument, the Adapter method creates an ArrayList object that "contains" the original collection. All the changes you make on the outer ArrayList object are duplicated in the original collection, and vice versa. You might want to use the Adapter method because the ArrayList class implements several methods—Reverse, Sort, BinarySearch, ToArray, IndexOf, and LastIndexOf, just to name a few—that are missing in a simpler IList object. The following code sample demonstrates how you can use this technique to reverse (or sort, and so on) all the items in a ListBox control:

```
// Create a wrapper around the ListBox.Items (IList) collection.
ArrayList lbAdapter = ArrayList.Adapter(listBox1.Items);
// Reverse their order.
lbAdapter.Reverse();
```

If you don't plan to reuse the ArrayList wrapper, you can make this code even more concise:

```
ArrayList.Adapter(listBox1.Items).Reverse();
```

The ReadOnly static method is similar to Adapter, except it returns a ArrayList that you can't modify in any way, including adding, removing, or assigning elements. This method can be useful when you want to pass your ArrayList to a method you didn't write yourself and you want to be sure that the method doesn't mistakenly modify the ArrayList or its elements.

The Hashtable Type

The Hashtable class implements the IDictionary interface, and it behaves much like the Scripting.Dictionary object that was available to COM developers. (The Dictionary object can be found in the Microsoft Scripting Runtime library.) All objects based on the IDictionary interface manage two internal series of data—values and keys—and you can use a key to retrieve the corresponding value. The actual implementation of the interface depends on the specific type. For example, the Hashtable type uses an internal hash table, a well-known data structure that has been studied for decades by computer scientists and has been thoroughly described in countless books on algorithms.

When a (key, value) pair is added to a Hashtable object, the position of an element in the internal array is based on the numeric hash code of the key. When you later search for that key, the key's hash code is used again to locate the associated value as quickly as possible, without sequentially visiting all the elements in the hash table. The .NET Hashtable type lets you use any object as a key. Behind the scenes, the Hashtable object uses the key's GetHashCode, a method that all objects inherit from System.Object.

Depending on how the hash code is evaluated, it frequently happens that multiple keys map to the same slot (or *bucket*) in the hash table. In this case, you have a *collision*. The .NET Hashtable object uses double hashing to minimize collisions, but it can't avoid collisions completely. Never fear—collisions are automatically dealt with transparently for the programmer, but you can get optimal performance by selecting an adequate initial capacity for the hash table. A larger table doesn't speed up searches remarkably, but it makes insertions faster.

You can also get better performance by selecting a correct load factor when you create a Hashtable object. This number determines the maximum ratio between values and buckets before the hash table is automatically expanded. The smaller this value is, the more memory is allocated to the internal table and the fewer collisions occur when you're inserting or searching for a value. The default load factor is 1.0, which in most cases delivers a good-enough performance, but you can set a smaller load factor when you create the Hashtable if you're willing to trade memory for better performance. You can initialize a Hashtable object in many ways:

```
// Default load factor and initial capacity
Hashtable ht = new Hashtable();
// Default load factor and specified initial capacity
Hashtable ht2 = new Hashtable(1000);
// Specified initial capacity and custom load factor
Hashtable ht3 = new Hashtable(1000, 0.8);
```

You can also initialize the Hashtable by loading it with the elements contained in any other object that implements the IDictionary interface (such as another Hashtable or a SortedList object). This technique is especially useful when you want to change the load factor of an existing hash table:

```
// Decrease the load factor of the current Hashtable.
ht = new HashTable(ht, 0.5);
```

Other, more sophisticated variants of the constructor let you pass an IComparer object to compare keys in a customized fashion, an IHashCodeProvider object to supply a custom algorithm for calculating hash codes of keys, or an IEqualityComparer object if you want to change the way keys are compared with each other. (More on this later.)

Once you've created a Hashtable, you can add a key and value pair, read or modify the value associated with a given key, and remove an item using the Remove method:

```
// Syntax for Add method is Add(key, value).
ht.Add("Joe", 12000);
ht.Add("Ann", 13000);
// Referencing a new key creates an element.
ht["Robert"] = 15000;
Console.Write(ht["Joe"]);            // => 12000
ht["Ann"] = (int) ht["Ann"] + 1000;
// By default keys are compared in case-insensitive mode,
// so the following statement creates a *new* element.
ht["ann"] = 15000;
// Reading a nonexistent element doesn't create it.
Console.WriteLine(ht["Lee"]);       // Doesn't display anything

// Remove an element given its key.
ht.Remove("Chris");
// How many elements are now in the Hashtable?
Console.WriteLine(ht.Count);        // => 4

// Adding an element that already exists throws an exception.
ht.Add("Joe", 11500);              // Throws ArgumentException.
```

As I explained earlier, you can use virtually anything as a key, including a numeric value. When you're using numbers as keys, a Hashtable looks deceptively similar to an array:

```
ht[1] = 123;
ht[2] = 345;
```

But never forget that the expression between parentheses is just a key and not an index; thus, the ht[2] element isn't necessarily stored "after" the ht[1] element. As a matter of fact, the elements in a Hashtable object aren't stored in a particular order, and you should never write code that assumes that they are. This is the main difference between the Hashtable object and the SortedList object (which is described next).

The Hashtable object implements the IEnumerable interface, so you can iterate over all its elements using a foreach loop. Each element of a Hashtable is a DictionaryEntry object, which exposes a Key and a Value property:

```
foreach ( DictionaryEntry de in ht )
{
   Console.WriteLine("ht('{0}') = {1}", de.Key, de.Value);
}
```

The Hashtable's Keys and Values properties return an ICollection-based object that contains all the keys and all the values, respectively, so you can assign them to any object that implements the ICollection interface. Or you can use these properties directly in a foreach loop:

```
// Display all the keys in the Hashtable.
foreach ( object o in ht.Keys )     // Or use ht.Values for all the values.
{
   Console.WriteLine(o);
}
```

An important detail: the ICollection objects returned by the Keys and Values properties are "live" objects that continue to be linked to the Hashtable and reflect any additions and deletions performed subsequently, as this code demonstrates:

```
ht.Clear();
ICollection values = ht.Values;
ht.Add("Chris", 11000);
// Prove that the collection continues to be linked to the Hashtable.
Console.WriteLine(values.Count);            // => 1
```

By default, keys are compared in a case-sensitive way, so Joe, JOE, and joe are considered distinct keys. You can create case-insensitive instances of the Hashtable class through one of its many constructors, or you can use the CreateCaseInsensitiveHashtable static method of the System.Collections.Specialized.CollectionsUtil, as follows:

```
Hashtable ht4 = CollectionsUtil.CreateCaseInsensitiveHashtable();
```

Another way to implement a Hashtable that deals with keys in a nonstandard fashion is by providing an IEqualityComparer object to override the default comparison algorithm. For example, say that you want to create a Hashtable where all keys are Double (or convertible to Double) but are automatically rounded to the second decimal digit so that, for example, the keys 1.123 and 1.119 resolve to the same element in the Hashtable. You might perform the rounding each time you add or retrieve an element in the table, but the approach based on the IEqualityComparer interface is more elegant because it moves the responsibility into the Hashtable and away from the client:

```
public class FloatingPointKeyComparer : IEqualityComparer
{
   public int digits;

   public FloatingPointKeyComparer(int digits)
   {
      this.digits = digits;
   }

   public new bool Equals(object x, object y)
   {
      double d1 = Math.Round(Convert.ToDouble(x), digits);
      double d2 = Math.Round(Convert.ToDouble(y), digits);
      return d1 == d2;
   }
   public int GetHashCode(object obj)
   {
      double d = Math.Round(Convert.ToDouble(obj), digits);
      return d.GetHashCode();
   }
}
```

The FloatingPointKeyComparer's constructor takes the number of digits used when rounding keys, so you can use it for any precision. The following example illustrates how to use it for keys rounded to the second decimal digit.

```
ht = new Hashtable(new FloatingPointKeyComparer(2));
ht.Add(1.123, "first");
ht.Add(1.456, "second");
// Prove that keys that round to the same Double number resolve to same item.
Console.WriteLine(ht[1.119]);                       // => first
```

The SortedList Type

The SortedList object is arguably the most versatile nongeneric collection-like object in the .NET Framework. It implements the IDictionary interface, like the Hashtable object, and also keeps its elements sorted. Alas, you pay for all this power in terms of performance, so you should use the SortedList object only when your programming logic requires an object with all this flexibility.

The SortedList object manages two internal arrays, one for the values and one for the companion keys. These arrays have an initial capacity, but they automatically grow when the need arises. Entries are kept sorted by their key, and you can even provide an IComparer object to affect how complex values (a Person object, for example) are compared and sorted. The SortedList class provides several constructor methods:

```
// A SortedList with default capacity (16 entries)
SortedList sl = new SortedList();
// A SortedList with specified initial capacity
SortedList sl2 = new SortedList(1000);

// A SortedList can be initialized with all the elements in an IDictionary.
Hashtable ht = new Hashtable();
ht.Add("Robert", 100);
ht.Add("Ann", 200);
ht.Add("Joe", 300);
SortedList sl3 = new SortedList(ht);
```

As soon as you add new elements to the SortedList, they're immediately sorted by their key. Like the Hashtable class, a SortedList contains DictionaryEntry elements:

```
foreach ( DictionaryEntry de in sl3 )
{
   Console.WriteLine("sl3('{0}') = {1}", de.Key, de.Value);
}
```

Here's the result that appears in the console window:

```
sl3('Ann') = 200
sl3('Joe') = 300
sl3('Robert') = 100
```

Keys are sorted according to the order implied by their IComparable interface, so numbers and strings are always sorted in ascending order. If you want a different order, you must create an object that implements the IComparer interface. For example, you can use the following class to invert the natural string ordering:

```
public class ReverseStringComparer : IComparer
{
   public int Compare(object x, object y)
   {
      // Just change the sign of the String.Compare result.
      return -string.Compare(x.ToString(), y.ToString());
   }
}
```

You can pass an instance of this object to one of the two overloaded constructors that take an IComparer object:

```
// A SortedList that loads all its elements from a Hashtable and
// sorts them with a custom IComparer object.
SortedList sl5 = new SortedList(ht, new ReverseStringComparer());
foreach ( DictionaryEntry de in sl5 )
{
   Console.WriteLine("sl3('{0}') = {1}", de.Key, de.Value);
}
```

Here are the elements of the resulting SortedList object:

```
sl5('Robert') = 100
sl5('Joe') = 300
sl5('Ann') = 200
```

The SortedList class compares keys in case-sensitive mode, with lowercase characters coming before their uppercase versions (for example, Ann comes before ANN, which in turn comes before Bob). If you want to compare keys without taking case into account, you can create a case-insensitive SortedList object using the auxiliary CollectionsUtil object in the System .Collections.Specialized namespace:

```
SortedList sl6 = CollectionsUtil.CreateCaseInsensitiveSortedList();
```

In this case, adding two elements whose keys differ only in case throws an Argument-Exception object.

You are already familiar with the majority of the members exposed by the SortedList type because they are also exposed by the Hashtable or ArrayList types: Capacity, Count, Keys, Values, Clear, Contains, CopyTo, Remove, RemoveAt, TrimToSize. The meaning of other methods should be self-explanatory: ContainsKey returns true if the SortedList contains a given key and is a synonym for Contains; ContainsValue returns true if the SortedList contains a given value; GetKey and GetByIndex return the key or the value at a given index; SetByIndex changes the value of an element at a given index; IndexOfKey and IndexOfValue return the index of a given key or value, or −1 if the key or the value isn't in the SortedList. All these methods work as intended, so I won't provide any code examples for them.

A couple of methods require further explanation, though: GetKeyList and GetValueList. These methods are similar to the Keys and Values properties, except they return an IList object rather than an ICollection object and therefore you can directly access an element at a given index. As for the Keys and Values properties, the returned object reflects any change in the SortedList.

```
sl = new SortedList();
// Get a live reference to key and value collections.
IList alKeys = sl.GetKeyList();
IList alValues = sl.GetValueList();
// Add some values out of order.
sl.Add(3, "three");
sl.Add(2, "two");
sl.Add(1, "one");
// Display values in sorted order.
for ( int i = 0; i <= sl.Count - 1; i++ )
{
   Console.WriteLine("{0} = '{1}'", alKeys[i], alValues[i]);
}
// Any attempt to modify the IList object throws an exception.
alValues.Insert(0, "four");            // Throws NotSupportedException error.
```

As I said before, the SortedList class is the most powerful collection-like object, but it's also the most demanding in terms of resources and CPU time. To see what kind of overhead you can expect when using a SortedList object, I created a routine that adds 100,000 elements to an ArrayList object, a Hashtable object, and a SortedList object. The results were pretty interesting: the ArrayList object was about 4 times faster than the Hashtable object, which in turn was from 8 to 100 times faster than the SortedList object was. Even though you can't take these ratios as reliable in all circumstances, they clearly show that you should never use a more powerful data structure if you don't really need its features.

In general, you should never use a SortedList if you can get along with a different data structure, unless you really need to keep elements sorted *always*. In most practical cases, however, you just need to sort elements after you've read them, so you can load them into a Hashtable and, when loading has completed, pass the Hashtable to the SortedList's constructor. To illustrate this concept and show how these types can cooperate with each other, I have prepared a short program that parses a long text string (for example, the contents of a text file) into individual words and loads them into an ArrayList; then it finds unique words by loading each word in a Hashtable and finally displays the sorted list of words in alphabetical order, together with the number of occurrences of that word:

```
// Read the contents of a text file. (Change file path as needed.)
string filetext = File.ReadAllText("foreword.txt");
// Use regular expressions to parse individual words, put them in an ArrayList.
ArrayList alWords = new ArrayList();
foreach ( Match m in Regex.Matches(filetext, @"\w+") )
{
   alWords.Add(m.Value);
}
Console.WriteLine("Found {0} words.", alWords.Count);

// Create a case-insensitive Hashtable.
Hashtable htWords = CollectionsUtil.CreateCaseInsensitiveHashtable();
// Process each word in the ArrayList.
foreach ( string word in alWords )
{
   // Search this word in the Hashtable.
   object elem = htWords[word];
   if ( elem == null )
   {
      // Not found: this is the first occurrence.
      htWords[word] = 1;
   }
   else
   {
      // Found: increment occurrence count.
      htWords[word] = (int) elem + 1;
   }
}
// Sort all elements alphabetically.
SortedList slWords = new SortedList(htWords);
// Display words and their occurrence count.
foreach ( DictionaryEntry de in slWords )
{
   Console.WriteLine("{0} ({1} occurrences)", de.Key, de.Value);
}
```

Read Chapter 6, “Regular Expressions,” for more information about regular expressions.

Other Collections

Although the ArrayList, Hashtable, and SortedList types are collections you might need most frequently in your applications, the System.Collections namespace contains several other useful types. In this roundup section, I cover the Stack, Queue, BitArray, and BitVector32 types.

The Stack Type

The System.Collections.Stack type implements a last in, first out (LIFO) data structure, namely, a structure into which you can push objects and later pop them out. The last object pushed in is also the first one popped out. The three basic methods of a Stack object are Push, Pop, and Peek; the Count property returns the number of elements currently in the stack:

```
Stack st = new Stack();
// Push three values onto the stack.
st.Push(10);
st.Push(20);
st.Push(30);
// Pop the value on top of the stack, and display its value.
Console.WriteLine(st.Pop());              // => 30
// Read the value on top of the stack without popping it.
Console.WriteLine(st.Peek());             // => 20
// Now pop it.
Console.WriteLine(st.Pop());              // => 20
// Determine how many elements are now in the stack.
Console.WriteLine(st.Count);              // => 1
// Pop the only value still on the stack.
Console.WriteLine(st.Pop());              // => 10
// Check that the stack is now empty.
Console.WriteLine(st.Count);              // => 0
```

The only other methods that can prove useful are Contains, which returns true if a given value is currently on the stack; ToArray, which returns the contents of the stack as an array of the specified type; and Clear, which removes all the elements from the stack:

```
// Is the value 10 somewhere in the stack?
if ( st.Contains(10) )
{
   Console.Write("Found");
}

// Extract all the items to an array.
object[] values = st.ToArray();
// Clear the stack.
st.Clear();
```

The Stack object supports the IEnumerable interface, so you can iterate over its elements without popping them by means of a foreach loop:

```
foreach ( object o in st )
{
   Console.WriteLine(o);
}
```

The Queue Type

A first in, first out (FIFO) structure, also known as a *queue* or *circular buffer*, is often used to solve recurring programming problems. You need a queue structure when a portion of an application inserts elements at one end of a buffer and another piece of code extracts the first available element at the other end. This situation occurs whenever you have a series of elements that you must process sequentially but can't process immediately.

You can render a queue in C# by leveraging the System.Collections.Queue object. Queue objects have an initial capacity, but the internal buffer is automatically extended if the need arises. You create a Queue object by specifying its capacity and a growth factor, both of which are optional:

```
// A queue with initial capacity of 200 elements; a growth factor equal to 1.5
// (When new room is needed, the capacity will become 300, then 450, 675, etc.)
Queue qu1 = new Queue(200, 1.5F);
// A queue with 100 elements and a default growth factor of 2
Queue qu2 = new Queue(100);
// A queue with 32 initial elements and a default growth factor of 2
Queue qu3 = new Queue();
```

The key methods of a Queue object are Enqueue, Peek, and Dequeue. Check the output of the following code snippet, and compare it with the behavior of the Stack object:

```
Queue qu = new Queue(100);
// Insert three values in the queue.
qu.Enqueue(10);
qu.Enqueue(20);
qu.Enqueue(30);
// Extract the first value, and display it.
Console.WriteLine(qu.Dequeue());              // => 10
// Read the next value, but don't extract it.
Console.WriteLine(qu.Peek());                 // => 20
// Extract it.
Console.WriteLine(qu.Dequeue());              // => 20
// Check how many items are still in the queue.
Console.WriteLine(qu.Count);                  // => 1
// Extract the last element, and check that the queue is now empty.
Console.WriteLine(qu.Dequeue());              // => 30
Console.WriteLine(qu.Count);                  // => 0
```

The Queue object also supports the Contains method, which checks whether an element is in the queue, and the Clear method, which clears the queue's contents. The Queue class implements IEnumerable and can be used in a foreach loop.

The BitArray Type

A BitArray object can hold a large number of Boolean values in a compact format, using a single bit for each element. This class implements IEnumerable (and thus supports foreach), ICollection, and ICloneable. You can create a BitArray object in many ways:

```
// Provide the number of elements (all initialized to false).
BitArray ba = new BitArray(1024);
// Provide the number of elements, and initialize them to a value.
BitArray ba2 = new BitArray(1024, true);

// Initialize the BitArray from an array of bool, byte, or int.
bool[] boolArr = new bool[1024];
```

```
// Initialize the boolArr array here.
…
BitArray ba3 = new BitArray(boolArr);

// Initialize the BitArray from another BitArray object.
BitArray ba4 = new BitArray(ba);
```

You can retrieve the number of elements in a BitArray by using either the Count property or the Length property. The Get method reads and the Set method modifies the element at the specified index:

```
// Set element at index 9, and read it back.
ba.Set(9, true);
Console.WriteLine(ba.Get(9));              // => True
```

The CopyTo method can move all elements back to an array of Booleans, or it can perform a bitwise copy of the BitArray to a zero-based Byte or Integer array:

```
// Bitwise copy to an array of Integers
int[] intArr = new int[32];                 // 32 elements * 32 bits each = 1,024 bits
// Second argument is the index in which the copy begins in target array.
ba.CopyTo(intArr, 0);
// Check that bit 9 of first element in intArr is set.
Console.WriteLine(intArr[0]);               // => 512
```

The Not method complements all the bits in the BitArray object:

```
ba.Not();                                   // No arguments
```

The And, Or, and Xor methods enable you to perform the corresponding operation on pairs of Boolean values stored in two BitArray objects:

```
// Perform an AND operation of all the bits in the first BitArray
// with the complement of all the bits in the second BitArray.
ba.And(ba2.Not);
```

Finally, you can set or reset all the bits in a BitArray class using the SetAll method:

```
// Set all the bits to true.
ba.SetAll(true);
```

The BitArray type doesn't expose any methods that enable you to quickly determine how many true (or false) elements are in the array. You can take advantage of the IEnumerator support of this class and use a foreach loop:

```
int bitCount = 0;
foreach ( bool b in ba )
{
   if ( b )
   {
      bitCount += 1;
   }
}
Console.WriteLine("Found {0} True values.", bitCount);
```

The BitVector32 Type

The BitVector32 class (in the System.Collections.Specialized namespace) is similar to the BitArray class in that it can hold a packed array of Boolean values, one per bit, but it's limited to 32 elements. However, a BitVector32 object can store a set of small integers that takes up to 32 consecutive bits and is therefore useful with bit-coded fields, such as those that you deal with when passing data to and from hardware devices.

```
BitVector32 bv = new BitVector32();
// Set one element and read it back.
bv[1] = true;
Console.WriteLine(bv[1]);                    // => True
```

You can also pass a 32-bit integer to the constructor to initialize all the elements in one pass:

```
// Initialize all elements to true.
bv = new BitVector32(-1);
```

To define a BitVector32 that is subdivided into sections that are longer than 1 bit, you must create one or more BitVector32.Section objects and use them when you later read and write individual elements. You define a section by means of the BitVector32.CreateSection static method, which takes the highest integer you want to store in that section and (for all sections after the first one) the previous section. Here's a complete example:

```
bv = new BitVector32();
// Create three sections, of 4, 5, and 6 bits each.
BitVector32.Section se1 = BitVector32.CreateSection(15);
BitVector32.Section se2 = BitVector32.CreateSection(31, se1);
BitVector32.Section se3 = BitVector32.CreateSection(63, se2);

// Assign a given value to each section.
bv[se1] = 10;
bv[se2] = 20;
bv[se3] = 40;
// Read values back.
Console.WriteLine(bv[se1]);             // => 10
Console.WriteLine(bv[se2]);             // => 20
Console.WriteLine(bv[se3]);             // => 40
```

The Data property sets or returns the internal 32-bit integer; you can use this property to save the bit-coded value into a database field or to pass it to a hardware device:

```
// Read the entire field as a 32-bit value.
Console.WriteLine(bv.Data);                    // => 20810
Console.WriteLine(bv.Data.ToString("X"));      // => 514A
```

Abstract Types for Strong-Typed Collections

As I have emphasized many times in earlier sections, all the types in the System.Collections namespace—with the exception of BitArray and BitVector32—are weakly typed collections that can contain objects of any kind. This feature makes them more flexible but less robust because any attempt to assign an object of the “wrong” type can’t be flagged as an error by the compiler. You can overcome this limitation by creating a strong-typed collection class.

In Visual C# .NET 2003, you can implement custom strong-typed collection types by inheriting from one of the abstract base classes that the .NET Framework offers, namely these:

- **CollectionBase** For strong-typed IList-based collections, that is, types that are functionally similar to ArrayList but capable of accepting objects of a specific type only.
- **ReadOnlyCollectionBase** Like CollectionBase, except that the collection has a fixed size and you can’t add elements to or remove elements from it after it has been instantiated. (Individual items can be either read-only or writable, depending on how you implement the collection.)
- **DictionaryBase** For strong-typed IDictionary-based collections, that is, types that are functionally similar to Hashtable but capable of accepting objects of a specific type only.
- **NameObjectCollectionBase** (In the System.Collections.Specialized namespace) For strong-typed collections whose elements can be accessed by either their index or the key associated with them. (You can think of these collections as a hybrid between the ArrayList and the Hashtable types.)

The importance of these base collection types has decreased in Visual C# 2005 because generics enable you to implement strong-typed collections in a much simpler and more efficient manner. However, in some scenarios you might need to use these base types in .NET Framework 2.0 as well, for example, when implementing a collection that must accept objects of two or more distinct types and these types don’t share a common base class or interface. Another case when generics aren’t very helpful is when you need to implement the IEnumerable interface directly, as I show in Chapter 3.

The CollectionBase Type

For the sake of illustration, I will show how you can inherit from CollectionBase to create an ArrayList-like collection that can host only Person objects. Consider the following definition of a Person:

```
public class Person
{
   // These should be properties in a real-world application.
   public string FirstName;
   public string LastName;
   public Person Spouse;
   public readonly ArrayList Children = new ArrayList();

   public Person(string firstName, string lastName)
   {
      this.FirstName = firstName;
      this.LastName = lastName;
   }

   public string ReverseName()
   {
      return LastName + ", " + FirstName;
   }
}
```

The Spouse member enables you to create a one-to-one relationship between two Person objects, whereas the Children member can implement a one-to-many relationship. The problem is that the Children collection is weakly typed; thus, a client program might mistakenly add to it an object of the wrong type without the compiler being able to spot the problem. You can solve this problem by creating a class that inherits from CollectionBase and that exposes a few strong-typed members that take or return Person objects:

```
public class PersonCollection : CollectionBase
{
   public void Add(Person item)
   {
      this.List.Add(item);
   }

   public void Remove(Person item)
   {
      this.List.Remove(item);
   }

   public Person this[int index]
   {
      get { return ((Person)this.List[index]); }
      set { this.List[index] = value; }
   }
}
```

The PersonCollection type inherits most of its public members from its base class, including Count, Clear, and RemoveAt; these are the members with signatures that don't mention the type of the specific objects you want to store in the collection (Person, in this case). Your job is to provide only the remaining members, which do nothing but delegate to the inner IList object by means of the protected List property.

To make the collection behave exactly as an ArrayList, you need to implement additional members, including Sort, IndexOf, and BinarySearch. These methods aren't exposed by the protected List property, but you can reach them by using the InnerList protected member (which returns the inner ArrayList):

```
public void Sort()
{
   this.InnerList.Sort();
}
```

When you've completed the PersonCollection type, you can replace the declaration of the Children member in the Person class to implement the one-to-many relationship in a more robust manner:

```
// (In the Person class…)
public readonly PersonCollection Children = new PersonCollection();

// (In the client application…)
Person john = new Person("John", "Evans");
john.Children.Add(new Person("Robert", "Evans"));       // This works.
// *** The next statement doesn't even compile.
john.Children.Add(new Object());
```

Quite surprisingly, however, the PersonCollection isn't very robust because an application can still add non-Person objects to it by accessing its IList interface:

```
// These statements raise neither a compiler warning nor a runtime error!
(john.Children as IList)[0] = new object();
(john.Children as IList).Add(new object());
```

Unfortunately, there is no way to tell the compiler to reject the preceding statement, but at least you can throw an exception at run time by checking the type of objects being assigned or added in the OnValidate protected method:

```
// (In the PersonCollection class…)
protected override void OnValidate(object value)
{
   if ( !(value is Person) )
   {
      throw new ArgumentException("Invalid item");
   }
}
```

The CollectionBase abstract class exposes other protected methods that can be overridden to execute a piece of custom code just before or after an operation is performed on the collection: OnClear and OnClearComplete methods run before and after a Clear method; OnInsert and OnInsertComplete methods run when an item is added to the collection; OnRemove and OnRemoveComplete run when an item is removed from the collection; OnSet and OnSet-Complete run when an item is assigned; OnGet runs when an item is read. For example, you might need to override these methods when the collection must notify another object when its contents change.

The ReadOnlyCollectionBase Type

The main difference between the CollectionBase type and the ReadOnlyCollectionBase type is that the latter doesn't expose any public member that would let clients add or remove items, such as Clear and RemoveAt. For a fixed-sized collection, you shouldn't expose methods such as Add and Remove, so in most cases your only responsibility is to implement the indexer. If you mark this property with the ReadOnly key, clients can't even assign a new value to the collection's elements:

```
public class PersonCollection : ReadOnlyCollectionBase
{
   public PersonCollection()
   {
      // Initialize the inner collection here.
      …
   }

   public Person this[int index]
   {
      get { return (Person) this.List[index]; }
   }
}
```

The DictionaryBase Type

The technique to create a strong-typed dictionary is similar to what I've just showed for the CollectionBase type, except you inherit from DictionaryBase. This base class enables you to access an inner IDictionary object by means of the Dictionary protected property. Here's a PersonDictionary class that behaves much like the Hashtable object, but can contain only Person objects that are indexed by a string key:

```
public class PersonDictionary : DictionaryBase
{
   public void Add(string key, Person item)
   {
      this.Dictionary.Add(key, item);
   }

   public void Remove(string key)
   {
      this.Dictionary.Remove(key);
   }

   public Person this[string key]
   {
      get { return (Person) this.Dictionary[key]; }
      set { this.Dictionary[key] = value; }
   }

   protected override void OnValidate(object key, object value)
   {
      if ( !(key is string) )
      {
         throw new ArgumentException("Invalid key");
      }
      else if ( !(value is Person) )
      {
         throw new ArgumentException("Invalid item");
      }
   }
}
```

The NameObjectCollectionBase Type

As I mentioned previously, you can inherit from the NameObjectCollectionBase type to implement a strong-typed collection that can refer to its elements by either a key or a numeric index. This type uses an internal Hashtable structure, but it doesn't expose it to inheritors. Instead, your public methods must perform their operation by delegating to a protected Base*Xxxx* method, such as BaseAdd or BaseGet. Here's a complete example of a strong-typed collection based on the NameObjectCollectionBase abstract class:

```
public class PersonCollection2 : NameObjectCollectionBase
{
   public void Add(string key, Person p)
   {
      this.BaseAdd(key, p);
   }

   public void Clear()
   {
      this.BaseClear();
   }

   // The Remove method that takes a string key
   public void Remove(string key)
   {
      this.Remove(key);
   }

   // The Remove method that takes a numeric index
   public void Remove(int index)
   {
      this.Remove(index);
   }

   // The indexer property that takes a string key
   public Person this[string key]
   {
      get { return (Person) this.BaseGet(key); }
      set { this.BaseSet(key, value); }
   }

   // The indexer property that takes a numeric index
   public Person this[int index]
   {
      get { return (Person) this.BaseGet(index); }
      set { this.BaseSet(index, value); }
   }
}
```

Generic Collections

In Chapter 4, you saw how you can implement your own generic types. However, in most cases, you don't really need to go that far because you can simply use one of the many types defined in the System.Collections.Generic namespace, which contains both generic collection types and generic interfaces. You can use the generic collections both directly in your applications or inherit from them to extend them with additional methods. In either case, generics can make your programming much, much simpler.

For example, going back to the example in the section titled "The CollectionBase Type" earlier in this chapter, you can have the Person class expose a strong-typed collection of other Persons as easily as this code:

```
public class Person
{
   public readOnly List<Person> Children = new List<Person>();
   …
}
```

On the other hand, if you are migrating code from Visual C# .NET 2003 and don't want to break existing clients, you can use a different approach and replace the existing version of the PersonCollection strong-typed collection with this code:

```
public class PersonCollection : List<Person>
{
   // …and that's it!
}
```

Not only is the implementation of PersonCollection simpler, it is also more complete because it exposes all the methods you expect to find in a collection type, such as Sort and Reverse. Just as important, if the element type is a value type—such as a numeric type or a structure—the generic-based implementation is also far more efficient because values are never passed to an Object argument and therefore are never boxed.

The List Generic Type

If you are familiar with the ArrayList type, you already know how to use most of the functionality exposed by the List<T> type and its members: Add, Clear, Insert, Remove, RemoveAt, RemoveAll, IndexOf, LastIndexOf, Reverse, Sort, and BinarySearch. You can perform operations on multiple items by means of the GetRange, AddRange, InsertRange, and RemoveRange methods. The GetRange method returns another List<T> object, so you can assign its result in a strongly typed fashion:

```
// Create a list of persons
List<Person> persons = new List<Person>();
persons.Add(new Person("John", "Evans"));
persons.Add(new Person("Ann", "Beebe"));
persons.Add(new Person("Robert", "Zare"));
…
// Create a new collection and initialize it with 3 elements from first collection.
List<Person> persons2 = new List<Person>(persons.GetRange(0, 3));
// Add elements at indexes 10-14 from first collection.
persons2.AddRange(persons.GetRange(10, 5));
```

A List<T> collection can contain any object that derives from T; for example, a Person collection can also contain Employee objects if the Employee class derives from Person.

Interestingly, the AddRange and InsertRange methods take any object that implements the IEnumerable<T> interface; thus, you can pass them either another List object or a strong-typed array:

```
Person[] arr = new Person[] {new Person("John", "Evans"),
   new Person("Ann", "Beebe")};
// Insert these elements at the beginning of the collection.
persons.InsertRange(0, persons);
```

Because the arguments must implement the IEnumerable<T> interface, you can't pass them an Object array or a weakly typed ArrayList, even if you know for sure that the array or the ArrayList contains only objects of type T. In this case, you must write an explicit foreach loop:

```
// Add all the Person objects stored in an ArrayList.
foreach ( Person p in myArrayList )
{
   persons.Add(p);
}
```

The Remove method doesn't throw an exception if the specified element isn't in the collection; instead, it returns true if the element was successfully removed, false if the element wasn't found:

```
if ( persons.Remove(aPerson) )
{
   Console.WriteLine("The specified person was in the list and has been removed.");
}
```

The TrimExcess method enables you to reclaim the memory allocated to unused slots:

```
persons.TrimExcess();
```

This method does nothing if the list currently uses 90 percent or more of its current capability. The rationale behind this behavior is that trimming a list is an expensive operation and there is no point in performing it if the expected advantage is negligible.

Alternatively, you can assign the Capacity property directly. By default, the initial capacity is 4, unless you pass a different value to the constructor, but this value might change in future versions of the .NET Framework:

```
persons.Capacity = persons.Count;
```

Some generic methods of the List type might puzzle you initially. For example, the Sort method works as expected if the element type supports the IComparable interface; however, you can't provide an IComparer object to it to sort according to a user-defined order, as you'd do with a weak-typed ArrayList. Instead, you must define a class that implements the strong-typed IComparer<Person> interface. For example, the following class can work as a strong-typed comparer for the Person class:

```
public class PersonComparer : IComparer<Person>
{
   public int Compare(Person x, Person y)
   {
      return x.ReverseName().CompareTo(y.ReverseName());
   }
}
```

You can then use the PersonComparer class with the Sort method:

```
// Sort a collection of persons according to the ReverseName property.
persons.Sort(new PersonComparer());
```

(You can also use the same PersonComparer class with BinarySearch for superfast searches in a sorted collection.) A welcome addition to the Sort method in ArrayList is the ability to pass a delegate of type Comparison<T>, which must point to a function that compares two T objects and returns an integer that specifies which is greater. This feature means that you don't need to define a distinct comparer class for each possible kind of sort you want to implement:

```
// This function can be used to sort in the descending direction.
private static int ComparePersonsDesc(Person p1, Person p2)
{
   // Notice that the order of arguments is reversed.
   return p2.ReverseName().CompareTo(p1.ReverseName());
}

// Elsewhere in the program…
static void SortPersonList()
{
   List<Person> persons = new List<Person>();
   …
   persons.Sort(ComparePersonsDesc);
}
```

Or you can avoid using an external method and use an anonymous method instead:

```
persons.Sort( delegate(Person p1, Person p2)
             { return p2.ReverseName().CompareTo(p1.ReverseName()); });
```

In addition to all the methods you can find in the ArrayList, the List type exposes all the new generic methods that have been added to the Array class in .NET Framework 2.0 and that expect a delegate as an argument, namely, ConvertAll, Exists, Find, FindAll, FindIndex, FindLastIndex, ForEach, and TrueForAll. For example, the TrueForAll method takes a Predicate<T> delegate, which must point to a function that tests a T object and returns a Boolean, so you can pass it the address of the String.IsNullOrEmpty static method to check whether all the elements of a List<string> object are null or empty:

```
List<string> list = new List<string>();
// Fill the list and process its elements…
…
if ( list.TrueForAll(string.IsNullOrEmpty) )
{
   Console.WriteLine("All elements are null or empty strings.");
}
```

You can also use the instance Equals method that the String type and most numeric types expose to check whether all elements are equal to a specific value:

```
string testValue = "ABC";
if ( list.TrueForAll(testValue.Equals) )
{
   Console.WriteLine("All elements are equal to 'ABC'");
}
```

Generic methods based on delegates often enable you to create unbelievably concise code, as in this example:

```
// Create two strong-typed collections of Double values.
List<double> list1 = new List<double>(new double[] { 1, 2, 3, 4, 5, 6, 7, 8, 9 });
List<double> list2 = new List<double>(new double[] { 0, 9, 12, 3, 6 });
// Check whether the second collection is a subset of the first one.
bool isSubset = list2.TrueForAll(list1.Contains);        // => False

// One statement to find all the elements in list2 that are contained in list1.
List<double> list3 = list2.FindAll(list1.Contains);
// Remove from list1 and list2 the elements that they have in common.
list1.RemoveAll(list3.Contains);
list2.RemoveAll(list3.Contains);

// Display the elements in the three lists.
list1.ForEach(Console.WriteLine);                         // => 1 2 4 5 7 8
list2.ForEach(Console.WriteLine);                         // => 0 12
list3.ForEach(Console.WriteLine);                         // => 9 3 6
```

The only method left to discuss is AsReadOnly. This method takes no arguments and returns a System.Collections.ObjectModel.ReadOnlyCollection<T> object, which, as its name suggests, is similar to the List type except you can neither add or remove objects nor modify existing items. Interestingly, the value returned by AsReadOnly is an adapter of the original list; thus, the elements in the returned list reflect any insertions and deletions performed on the original list. This feature enables you to pass a read-only reference to an external procedure that can read the most recent data added to the list but can't modify the list in any way:

```
// Get a read-only wrapper of the original list.
ReadOnlyCollection<double> roList = list1.AsReadOnly();
Console.WriteLine(roList.Count == list1.Count);              // => True
// Prove that roList reflects all the operations on the original list.
list1.Add(123);
Console.WriteLine(roList.Count == list1.Count);              // => True
```

The Dictionary Generic Type

The Dictionary<TKey,TValue> type is the generic counterpart of the Hashtable type, in that it can store (key, value) pairs in a strong-typed fashion. For example, here's a dictionary that can contain Person objects and index them by a string key (the person's complete name):

```
Dictionary<string,Person> dictPersons = new Dictionary<string,Person>();
dictPersons.Add("John Evans", new Person("John", "Evans"));
dictPersons.Add("Robert Zare", new Person("Robert", "Zare"));
```

The constructor of this generic class can take a capacity (the initial number of slots in the inner table), an object that implements the IDictionary<TKey,TValue> generic interface (such as another generic dictionary), an IEqualityComparer<T> object, or a few combinations of these three values. (In .NET Framework 2.0, the default capacity is just three elements, but might change in future versions of the .NET Framework.)

The constructor that takes an IEqualityComparer<T> object enables you to control how keys are compared. I demonstrated how to use the nongeneric IEqualityComparer interface in the section titled "The Hashtable Type" earlier in this chapter, so you should have no problem understanding how its generic counterpart works. The following class normalizes a person's name to the format "LastName,FirstName" and compares these strings in case-insensitive fashion:

```
public class NameEqualityComparer : IEqualityComparer<string>
{
   public bool Equals(string x, string y)
   {
      return string.Equals(NormalizedName(x), NormalizedName(y));
   }

   public int GetHashCode(string obj)
   {
      return NormalizedName(obj).GetHashCode();
   }

   // Helper method that returns a person name in upper case and in the
   // (LastName,FirstName) format, without any spaces.
   private string NormalizedName(string name)
   {
      // If there is a comma, assume that name is already in (last,first) format.
      if (name.IndexOf(',') < 0)
      {
         // Find first and last names.
         char[] separators = {' '};
         string[] parts = name.Split(separators, 2, StringSplitOptions.RemoveEmptyEntries);
         // Invert the two portions.
         name = parts[1] + "," + parts[0];
      }
      // Delete spaces, if any, convert to upper case, and return.
      return name.Replace(" ", "").ToUpper();
   }
}
```

You can use the NameEqualityComparer type to manage a dictionary of Persons whose elements can be retrieved by providing a name in several different formats:

```
dictPersons = new Dictionary<string, Person>(new NameEqualityComparer());
dictPersons.Add("John Evans", new Person("Joe", "Evans"));
dictPersons.Add("Robert Zare", new Person("Robert", "Zare"));
// Prove that the last element can be retrieved by providing a key in
// either the (last,first) format or the (first last) format, that spaces
// are ignored, and that character casing isn't significant.
string name;
name = dictPersons["robert zare"].ReverseName();        // => Zare, Robert
name = dictPersons["ZARE, robert"].ReverseName();       // => Zare, Robert
```

If necessary, you can retrieve a reference to the IEqualityComparer object by means of the dictionary's Comparer read-only property.

Unlike the Remove method in the Hashtable type, but similar to the Remove method in the List generic type, the Remove method of the Dictionary generic class returns true if the object was actually removed and false if no object with that key was found; therefore, you don't need to search the element before trying to remove it:

```
if (dictPersons.Remove("john evans"))
{
   Console.WriteLine("Element John Evans has been removed");
}
else
{
   Console.WriteLine("Element John Evans hasn't been found");
}
```

When a Dictionary object is used in a foreach loop, at each iteration you get an instance of the KeyValuePair<TKey,TValue> generic type, which enables you to access the dictionary elements in a strong-typed fashion:

```
foreach ( KeyValuePair<string, Person> kvp in dictPersons )
{
   // You can reference a member of the Person class in strong-typed fashion.
   Console.WriteLine("Key={0} FirstName={1}", kvp.Key, kvp.Value.FirstName);
}
```

The TryGetValue method takes a key and an object variable (which is taken by reference); if an element with that key is found, its value is assigned to the object and the method returns true. You can therefore test the presence of an element and retrieve it with a single operation by using this code:

```
Person p;
if ( dictPersons.TryGetValue("ann beebe", out p) )
{
   // The variable p contains a reference to the found element.
   Console.WriteLine("Found {0}", p.ReverseName());
}
```

```
else
{
   Console.WriteLine("Not found");
}
```

The remaining members of the Dictionary type are quite straightforward: the Keys and Values read-only properties, the Clear method, the ContainsKey and ContainsValue methods. They work exactly as do the methods with the same names of the Hashtable object (except that they are strong-typed) and I won't repeat their descriptions here.

The LinkedList Generic Type

Linked lists are data structures with elements that aren't stored in contiguous memory locations; you can visit all the elements of these lists because each element has a pointer to the next element (simple linked list) or to both the next and the previous elements (doubly linked list). Elements in such a structure are called *nodes* and cannot be referenced by means of an index or a key. You can reach a node only by following the chain of pointers, either starting at the first element and moving forward or starting at the last element and moving backward. Because there is no key, elements can have duplicate values. See Figure 5-1 for the .NET implementation of the double linked list data structure.

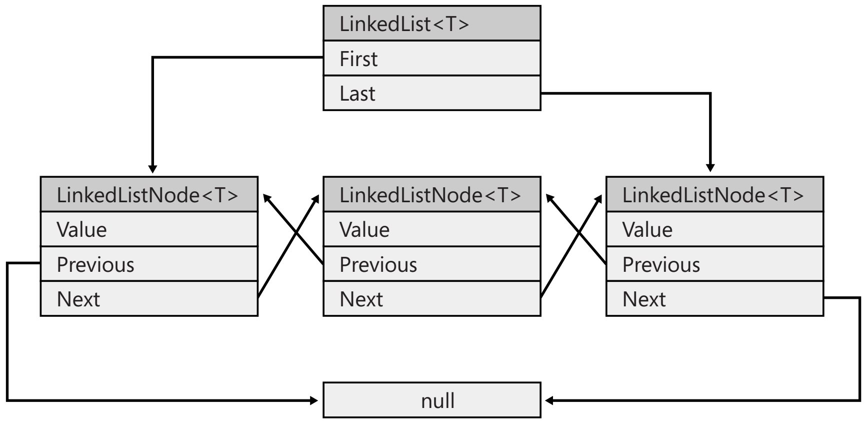

Figure 5-1 The LinkedList and LinkedListNode generic types

The LinkedList<T> type implements a strong-typed doubly linked list. Each node in a LinkedList structure is implemented as a LinkedListNode<T> object and exposes only four properties: Value (an object of type T), List (the list this node belongs to), Previous and Next (pointers to the previous and next nodes in the list). You can create a node by passing an object of type T to its constructor or by means of one of the methods in the parent LinkedList object, as you'll see in a moment. Two nodes in the list are special in that they represent the first and last nodes in the linked list. (These nodes are often called the *head* and the *tail* of the linked list.) You can get a reference to these nodes by means of the First and Last properties of the LinkedList type.

Enough theory for now. Let's see how to create and use a strong-typed generic linked list that can contain one or more Person objects. The remarks in code will help you to understand what happens inside the list:

```
LinkedList<Person> lnkList = new LinkedList<Person>();
// An empty linked list has no first or last node.
Console.WriteLine(lnkList.First == null);          // => True
Console.WriteLine(lnkList.Last == null);           // => True
```

The LinkedList type exposes four methods for adding a new node: AddFirst, AddLast, AddBefore, and AddAfter. All these methods have two overloads: the first overload takes an object of type T, wraps it into a LinkedListNode<T> object, inserts the node in the list, and returns it to the caller; the second overload takes a LinkedListNode<T> object and inserts it in the list at the desired position, but doesn't return anything. In most cases, you can write more concise code by using the former syntax and discarding the return value:

```
// Add the first node of the list.
Person p1 = new Person("John", "Evans");
lnkList.AddFirst(p1);
// When the list contains only one node, the first and last nodes coincide.
Console.WriteLine(lnkList.First == lnkList.Last);            // => True
```

Now the list isn't empty and you can add new elements using existing nodes as a reference for the AddBefore or AddAfter methods:

```
// Add a new node after the list head.
lnkList.AddAfter(lnkList.First, new Person("Ann", "Beebe"));
// The new node has become the list's tail node.
Console.WriteLine(lnkList.Last.Value.ReverseName());     // => Beebe, Ann
// Add a new node immediately before the list tail.
lnkList.AddBefore(lnkList.Last, new Person("Robert", "Zare"));
// Add a new node after the current list tail (it becomes the new tail).
lnkList.AddLast(new Person("James", "Hamilton"));
// Now the list contains four elements.
Console.WriteLine(lnkList.Count);                        // => 4
```

You can iterate over all the elements in a linked list in a couple of ways. First, you can use a traditional foreach loop:

```
foreach ( Person p in lnkList )
{
   Console.Write("{0} ", p.FirstName);         // => John Robert Ann James
}
```

Second, you can take advantage of the nature of the doubly linked list by following the chain of node pointers. This technique gives you more flexibility because you can traverse the list in both directions:

```
// Visit all nodes in reverse order.
LinkedListNode<Person> node = lnkList.Last;
while ( node != null )
```

```
{
   Console.WriteLine(node.Value.ReverseName());
   node = node.Previous;
}

// Change last name from Evans to Hamilton.
node = lnkList.First;
while ( node != null )
{
   if ( node.Value.LastName == "Evans" )
   {
      node.Value.LastName = "Hamilton";
   }
   node = node.Next;
}
```

Another good reason you might traverse the linked list manually is because you gain the ability to insert and delete nodes during the process. (Either operation would throw an exception if performed from inside a foreach loop.) Before I show how to perform this operation, let's take a step backward for a moment.

As I have already mentioned, the first thing to do on a freshly created LinkedList instance is create its first node (which also becomes its last node). This operation introduces an asymmetry between the first element added to the list and all the elements after it because adding the first element requires a different piece of code. This asymmetry makes programming a bit more complicated.

You can avoid the asymmetry and simplify programming by assuming that the first node in the linked list is a special node that is always present but contains no meaningful value:

```
LinkedList<Person> aList = new LinkedList<Person>();
aList.AddFirst(new LinkedListNode<Person>(null));
// You can now add all nodes with a plain AddLast method.
aList.AddLast(new Person("John", "Evans"));
aList.AddLast(new Person("Ann", "Beebe"));
aList.AddLast(new Person("Robert", "Zare"));
```

You must take the "dummy" first node into account when iterating over all the elements:

```
// We are sure that the first node exists; thus, the next statement can never throw.
LinkedListNode<Person> aNode = aList.First.Next;
while ( aNode != null )
{
   Console.WriteLine(aNode.Value.ReverseName());
   aNode = aNode.Next;
}
```

The first empty node simplifies programming remarkably because you don't need to take any special case into account. For example, here's a loop that removes all the persons that match a given criterion:

```
aNode = aList.First.Next;
while ( aNode != null )
```

```
{
   // Remove this node if the last name is Evans.
   if ( aNode.Value.LastName == "Evans" )
   {
      // Backtrack to previous node and remove the node that was current.
      // (We can be sure that the previous node exists.)
      aNode = aNode.Previous;
      aList.Remove(aNode.Next);
   }
   aNode = aNode.Next;
}
```

The LinkedList type also exposes the RemoveFirst and RemoveLast methods to remove the list's head and tail nodes, respectively. There is also an overload of the Remove method that takes an object of type T and returns true if the object was found and removed, false otherwise.

A last note about the technique based on the dummy first node: remember to take this dummy node into account when you display the number of elements in the list:

```
Console.WriteLine("The list contains {0} persons", aList.Count - 1);
```

The only methods I haven't covered yet are Find and FindLast, which take an object of type T and return the LinkedListNode<T> object that holds that object, or null if the search fails. Keep in mind, however, that searching a node is a relatively slow operation because these methods have to start at one of the list's ends and traverse each element until they find a match.

Other Generic Collections

The System.Collections.Generic namespace contains a few other generic collections, namely these:

- **Stack<T>** The strong-typed version of the Stack collection class
- **Queue<T>** The strong-typed version of the Queue collection class
- **SortedDictionary<TKey,TValue>** The strong-typed version of the SortedList, which uses keys of type TKey associated with values of type TValue

The SortedDictionary type exposes many of the methods of the Dictionary type, but it also keeps all its elements in sorted order. To keep elements sorted, the constructor can take an object that implements the IComparer<TKey> generic interface. For example, suppose that your sorted list uses a filename as a key and you want to sort the elements according to file extension. First, define a comparer class that implements the IComparer<string> interface:

```
public class FileExtensionComparer : IComparer<string>
{
   public int Compare(string x, string y)
   {
      // Compare the extensions of filenames in case-insensitive mode.
      int res = string.Compare(Path.GetExtension(x), Path.GetExtension(y), true);
      if ( res == 0 )
      {
         // If extensions are equal, compare the entire filenames.
         res = string.Compare(Path.GetFileName(x), Path.GetFileName(y), true);
      }
      return res;
   }
}
```

You can then define a sorted dictionary that contains the text associated with a series of text files and keeps the entries sorted on the file extensions:

```
SortedDictionary<string, string> fileDict = new SortedDictionary<string, string>();
// Load some elements.
fileDict.Add("foo.txt", "contents of foo.txt...");
fileDict.Add("data.txt", "contents of data.txt...");
fileDict.Add("data.doc", "contents of data.doc...");
// Check that files have been sorted on their extension.
foreach ( KeyValuePair<string, string> kvp in fileDict )
{
   Console.Write("{0}, ", kvp.Key);       // => data.doc, data.txt, foo.txt,
}
```

Like the method with the same name of the Dictionary type, the TryGetValue method enables you to check whether an element with a given key exists and to read it using a single operation:

```
string text;
if ( fileDict.TryGetValue("foo.txt", out text) )
{
   // The text variable contains the value of the "foo.txt" element.
   Console.WriteLine(text);
}
```

Even if elements in a SortedDictionary can be accessed in order, you can't reference them by their index. If you need to read the key or the value of the *N*th element, you must first copy the Keys or Values collection to a regular array:

```
// Display the value for the first item.
string[] values = new string[fileDict.Count];
fileDict.Values.CopyTo(values, 0);
Console.WriteLine(values[0]);
```

A Notable Example

I won't cover the Stack<T> and Queue<T> generic types: if you are familiar with their weak-typed counterparts, you already know how to use them. However, I do provide an example that demonstrates that these classes can help you write sophisticated algorithms in an efficient manner and with very little code. More precisely, I show you how to implement a complete Reverse Polish Notation (RPN) expression evaluator.

First a little background, for those new to RPN. An RPN expression is a sequence of operands and math operators in postfix notation. For example, the RPN expression "12 34 +" is equivalent to 12 + 34, whereas the expression "12 34 + 56 78 - *" is equivalent to (12 + 34) * (56 − 78). The RPN notation was used in programmable calculators in the 1980s and in programming languages such as Forth, but it is useful in many cases even today. (For example, a compiler must translate an expression to RPN to generate the actual IL or native code.) The beauty of the RPN notation is that you never need to assign a priority to operators and therefore you never need to use parentheses even for the most complex expressions. In fact, the rules for evaluating an RPN expression are quite simple:

1. Extract the tokens from the RPN expression one at a time.
2. If the token is a number, push it onto the stack.
3. If the token is an operator, pop as many numbers off the stack as required by the operator, execute the operation, and push the result onto the stack again.
4. When there are no more tokens, if the stack contains exactly one element, this is the result of the expression, else the expression is unbalanced.

Thanks to the String.Split method and the Stack<T> generic type, implementing this algorithm requires very few lines of code:

```
public static double EvalRPN(string expression)
{
   Stack<double> stack = new Stack<double>();
   // Split the string expression into tokens.
   string[] operands = expression.ToLower().Split(
      new char[] {' '}, StringSplitOptions.RemoveEmptyEntries);
   foreach ( string op in operands )
   {
      switch ( op )
      {
         case "+":
            stack.Push(stack.Pop() + stack.Pop());
            break;
         case "-":
            stack.Push(-stack.Pop() + stack.Pop());
            break;
         case "*":
            stack.Push(stack.Pop() * stack.Pop());
            break;
         case "/":
            stack.Push( (1 / stack.Pop()) * stack.Pop() );
            break;
         case "sqrt":
            stack.Push( Math.Sqrt(stack.Pop()) );
            break;
         default:
            // Assume this token is a number, throw if the parse operation fails.
```

```
                stack.Push(double.Parse(op));
                break;
            }
        }
        // Throw if stack is unbalanced.
        if ( stack.Count != 1 )
        {
            throw new ArgumentException("Unbalanced expression");
        }
        return stack.Pop();
    }
```

Here are a few usage examples:

```
double res = EvalRPN("12 34 + 56 78 - *");         // => -1012
res = EvalRPN("123 456 + 2 /");                    // => 289.5
res = EvalRPN("123 456 + 2 ");                     // => Exception: Unbalanced expression
res = EvalRPN("123 456 + 2 / *");                  // => Exception: Stack empty
```

The System.Generic.ObjectModel Namespace

Previously, I showed that you can inherit from a generic type and implement either a standard class or another generic type. For example, the following code implements a typed collection of Person objects:

```
public class PersonCollection : List<Person>
{
}
```

The next code defines a new generic type that is similar to a sorted dictionary except it enables you to retrieve keys and values by their numeric index, thus solving one of the limitations of the SortedDictionary generic type:

```
public class IndexableDictionary<TKey, TValue> : SortedDictionary<TKey, TValue>
{
   // Retrieve a key by its index.
   public TKey GetKey(int index)
   {
      // Retrieve the N-th key.
      TKey[] keys = new TKey[this.Count - 1 + 1];
      this.Keys.CopyTo(keys, 0);
      return keys[index];
   }

   public TValue this[int index]
   {
      get { return this[GetKey(index)]; }
      set { this[GetKey(index)] = value; }
   }
}
```

(Notice that the IndexableDictionary class has suboptimal performance because finding the key with a given index is a relatively slow operation.) Here's how you can use the Indexable-Dictionary type:

```
IndexableDictionary<string, Person> idPersons =
   new IndexableDictionary<string, Person>();
idPersons.Add("Zare, Robert", new Person("Robert", "Zare"));
idPersons.Add("Beebe, Ann", new Person("Ann", "Beebe"));
Console.WriteLine(idPersons[0].ReverseName());          // => Beebe, Ann
```

Even if inheriting from concrete types such as List and Dictionary is OK in most cases, sometimes you can write better inherited classes by deriving from one of the abstract generic types defined in the System.Collections.ObjectModel namespace:

- **Collection<T>** Provides a base class for generic collections that can be extended by adding or removing elements
- **ReadOnlyCollection<T>** Provides a base class for generic read-only collections
- **KeyedCollection<TKey,TValue>** Provides a base class for generic dictionaries

The main difference between regular generic types and the preceding abstract types is that the latter expose several protected methods that you can override to get more control of what happens when an element is modified or added to or removed from the collection. For example, the Collection<T> class exposes the following protected methods: ClearItems, InsertItem, RemoveItem, and SetItems, plus an Items protected property that enables you to access the inner collection of items.

Here's an example of a collection that can store a set of IComparable objects and that exposes an additional pair of read-only properties that returns the minimum and maximum values in the collection:

```
public class MinMaxCollection<T> : Collection<T>
   where T : IComparable
{
   private T min; private T max;
   private bool upToDate;

   public T MinValue
   {
      get
      {
         if ( !upToDate )
         {
            UpdateValues();
         }
         return min;
      }
   }

   public T MaxValue
   {
      get
      {
         if ( !upToDate )
         {
            UpdateValues();
         }
         return max;
      }
   }

   protected override void InsertItem(int index, T item)
   {
      base.InsertItem(index, item);
      if ( this.Count == 1 )
      {
         UpdateValues();
      }
      else if ( upToDate )
      {
         // If values are up-to-date, adjusting the min/max value is simple.
         if ( min.CompareTo(item) > 0 )
         {
            min = item;
         }
         if ( max.CompareTo(item) < 0 )
         {
            max = item;
         }
      }
   }

   protected override void SetItem(int index, T item)
   {
      // Check whether we're assigning the slot holding the min or max value.
      if ( min.CompareTo(this[index]) == 0 || max.CompareTo(this[index]) == 0 )
      {
         upToDate = false;
      }
      base.SetItem(index, item);
   }

   protected override void RemoveItem(int index)
   {
      // Check whether we're removing the min or max value.
      if ( min.CompareTo(this[index]) == 0 || max.CompareTo(this[index]) == 0 )
      {
         upToDate = false;
      }
      base.RemoveItem(index);
   }

   protected override void ClearItems()
   {
      base.ClearItems();
      upToDate = false;
   }

   // Helper method that updates the min/max values
```

```
   private void UpdateValues()
   {
      if ( this.Count == 0 )
      {
         // Assign default value of T if collection is empty.
         min = default(T); max = default(T);
      }
      else
      {
         // Else evaluate the min/max value.
         min = this[0]; max = this[0];
         foreach ( T item in this )
         {
            if ( min.CompareTo(item) > 0 )
            {
               min = item;
            }
            if ( max.CompareTo(item) < 0 )
            {
               max = item;
            }
         }
      }
      // Signal that min/max values are now up-to-date.
      upToDate = true;
   }
}
```

The noteworthy aspect of the MinMaxCollection is that it keeps an up-to-date value of the MinValue and MaxValue properties if possible, as long as the client program just adds new elements. If the client changes or removes an element that is currently the minimum or the maximum value, the upToDate variable is set to false so that the MinValue and MaxValue properties are recalculated the next time they are requested. This algorithm is quite optimized, yet it's generic enough to be used with any numeric type (more precisely: any type that supports the IComparable interface):

```
MinMaxCollection<double> col = new MinMaxCollection<double>();
// MinValue and MaxValue are updated during these insertions.
col.Add(123); col.Add(456); col.Add(789); col.Add(-33);
// This removal doesn't touch MinValue and MaxValue.
col.Remove(456);
Console.WriteLine("Min={0}, Max={1}", col.MinValue, col.MaxValue); // => Min=-33, Max=789

// This statement does affect MinValue and therefore sets upToDate=False.
col.Remove(-33);
// The next call to MinValue causes the properties to be recalculated.
Console.WriteLine("Min={0}, Max={1}", col.MinValue, col.MaxValue); // => Min=123, Max=789
```

The ReadOnlyCollection<T> abstract class is similar to Collection<T>, except that it doesn't expose any method for changing, adding, or removing elements after the collection has been created. (It is therefore the strong-typed counterpart of the ReadOnlyCollectionBase nongeneric abstract class.) Because the only operation that is supported after creation is enumeration, this base class doesn't expose any overridable protected methods.