CHAPTER SIX

INTRODUCTION TO COLLECTIONS AND LINQ

6.1 Introduction

This chapter discusses one of the most important parts of the .NET Framework: collections. In C#, a collection is a group of objects. The .NET - Framework contains a large number of interfaces and classes that define and implement various types of collections. Collections simplify many programming tasks because they provide off-the-shelf solutions to several common, but sometimes tedious-to-develop. Finally, we discuss the fundamental of LINQ.

6.2 Introduction to Collections

The .NET Framework Class Library provides several classes, called collections, used to store groups of related objects. These classes provide efficient methods that organize, store and retrieve your data without requiring knowledge of how the data is being stored. This reduces application-development time.

You've used arrays to store sequences of objects. Arrays do not automatically change their size at execution time to accommodate additional elements—you must do so manually by creating a new array or by using the Array class's Resize method The collection class List<T> (from namespace System.Collections.Generic) provides a convenient solution to this problem. The T is a placeholder—when declaring a new List, replace it with the type of elements that you want the List to hold. This is similar to specifying the type when declaring an array. For example,

```
List< int > list1;
```

declares list1 as a List collection that can store only int values, and

```
List< string > list2;
```

declares list2 as a List of strings. Classes with this kind of placeholder that can be used with any type are called generic classes. Figure 6.1 shows some common methods and properties of class List<T>.

Method or property	Description
`Add`	Adds an element to the end of the `List`.
`Capacity`	Property that gets or sets the number of elements a `List` can store without resizing.
`Clear`	Removes all the elements from the `List`.
`Contains`	Returns `true` if the `List` contains the specified element; otherwise, returns `false`.
`Count`	Property that returns the number of elements stored in the `List`.
`IndexOf`	Returns the index of the first occurrence of the specified value in the `List`.
`Insert`	Inserts an element at the specified index.
`Remove`	Removes the first occurrence of the specified value.
`RemoveAt`	Removes the element at the specified index.
`RemoveRange`	Removes a specified number of elements starting at a specified index.
`Sort`	Sorts the `List`.
`TrimExcess`	Sets the `Capacity` of the `List` to the number of elements the `List` currently contains (`Count`).

Fig. 6.1 | Some methods and properties of class `List<T>`.

Figure 6.2 demonstrates dynamically resizing a List object. The Add and Insert methods add elements to the List (lines 13–14). The Add method appends its argument to the end of the List. The Insert method inserts a new element at the specified position. The first argument is an index—as with arrays, collection indices start at zero. The second argument is the value that's to be inserted at the specified index. All elements at the specified index and above are shifted up by one position. This is usually slower than adding an element to the end of the List.

```
1  // Fig. 6.2: ListCollection.cs
2  // Generic List collection demonstration.
3  using System;
4  using System.Collections.Generic;
5
6  public class ListCollection
7  {
8     public static void Main( string[] args )
9     {
10       // create a new List of strings
11       List< string > items = new List< string >();
12
13       items.Add( "red" ); // append an item to the List
14       items.Insert( 0, "yellow" ); // insert the value at index 0
15
16       // display the colors in the list
17       Console.Write(
18       "Display list contents with counter-controlled loop:" );
19       for ( int i = 0; i < items.Count; i++ )
20          Console.Write( " {0}", items[ i ] );
21
22       // display colors using foreach
23       Console.Write(
24       "\nDisplay list contents with foreach statement:" );
25       foreach ( var item in items )
26              Console.Write( " {0}", item );
27
28       items.Add( "green" ); // add "green" to the end of the List
29       items.Add( "yellow" ); // add "yellow" to the end of the List
30
31       // display the List
32       Console.Write( "\nList with two new elements:" );
33       foreach ( var item in items )
34              Console.Write( " {0}", item );
35
36       items.Remove( "yellow" ); // remove the first "yellow"
37
38       // display the List
39       Console.Write( "\nRemove first instance of yellow:" );
40       foreach ( var item in items )
41              Console.Write( " {0}", item );
42
43       items.RemoveAt( 1 ); // remove item at index 1
44
45       // display the List
46       Console.Write( "\nRemove second list element (green):" );
47       foreach ( var item in items )
48              Console.Write( " {0}", item );
49
50       // check if a value is in the List
51       Console.WriteLine( "\n\"red\" is {0}in the list",
52          items.Contains( "red" ) ? string.Empty : "not" );
53
54       // display number of elements in the List
55       Console.WriteLine( "Count: {0}", items.Count );
```

```
56
57          // display the capacity of the List
58       Console.WriteLine( "Capacity: {0}", items.Capacity );
59    } // end Main
60 } // end class ListCollection
```

```
Display list contents with counter-controlled loop: yellow red
Display list contents with foreach statement: yellow red
List with two new elements: yellow red green yellow
Remove first instance of yellow: red green yellow
Remove second list element (green): red yellow
"red" is in the list
Count: 2
Capacity: 4
```

Fig. 6.2 | Generic `List<T>` collection demonstration

Lines 19–20 display the items in the List. The Count property returns the number of elements currently in the List. Lists can be indexed like arrays by placing the index in square brackets after the List variable's name. The indexed List expression can be used to modify the element at the index. Lines 25–26 output the List by using a foreach statement. More elements are then added to the List, and it's displayed again (lines 28–34). The Remove method is used to remove the first element with a specific value (line 36). If no such element is in the List, Remove does nothing. A similar method, RemoveAt, removes the element at the specified index (line 43). When an element is removed through either of these methods, all elements above that index are shifted down by one—the opposite of the Insert method.

Line 52 uses the Contains method to check if an item is in the List. The Contains method returns true if the element is found in the List, and false otherwise. The method compares its argument to each element of the List in order until the item is found, so using Contains on a large List is inefficient.

Lines 55 and 58 display the List's Count and Capacity. Recall that the Count property (line 55) indicates the number of items in the List. The Capacity property (line 58) indicates how many items the List can hold without growing. When the List grows, it must create a larger internal array and copy each element to the new array. This is a time-consuming operation. It would be inefficient for the List to grow each time an element is added. Instead, the List grows only when an element is added and the Count and Capacity properties are equal—there's no space for the new element.

6.3 LINQ Providers

The syntax of LINQ is built into C#, but LINQ queries may be used in many different contexts because of libraries known as providers. A LINQ provider is a set of classes that implement LINQ operations and enable programs to interact with data sources to perform tasks such as sorting, grouping and filtering elements.

These providers, along with LINQ to Objects, mentioned above, are included with Visual Studio and the .NET Framework. There are many providers that are more specialized, allowing you to interact with a specific website or data format.

6.4 Querying an Array of int Values Using LINQ

Figure 6.3 demonstrates querying an array of integers using LINQ. Repetition statements that filter arrays focus on the process of getting the results—iterating through the elements and checking whether they satisfy the desired criteria. LINQ specifies the conditions that selected elements must satisfy. This is known as declarative programming—as opposed to imperative programming (which we've been doing so far) in which you specify the actual steps to perform a task. The query in lines 20–22 specifies that the results should consist of all the ints in the values array that are greater than 4. It does not specify how those results are obtained—the C# compiler generates all the necessary code automatically, which is one of the great strengths of LINQ. To use LINQ to Objects, you must import the System.Linq namespace (line 4).

```
1  // Fig. 6.3: LINQWithSimpleTypeArray.cs
2  // LINQ to Objects using an int array.
3  using System;
4  using System.Linq;
5
6  class LINQWithSimpleTypeArray
7  {
8     public static void Main( string[] args )
9     {
10       // create an integer array
11       int[] values = { 2, 9, 5, 0, 3, 7, 1, 4, 8, 5 };
12
13       // display original values
14       Console.Write( "Original array:" );
```

```
15        foreach  (     var element in values )
16         Console.Write(  " {0}", element );
17
18           // LINQ query that obtains values greater than 4 from the array
19        var     filtered =
20           from value in values
21           where value > 4
22           select value;
23
24           // display filtered results
25        Console.Write( "\nArray values greater than 4:" );
26        foreach ( var element in filtered )
27        Console.Write( " {0}", element );
28
29           // use orderby clause to sort original array in ascending order
30        var sorted =
31           from value in values
32           orderby value
33           select value;
34
35           // display sorted results
36        Console.Write( "\nOriginal array, sorted:" );
37        foreach ( var element in sorted )
38        Console.Write( " {0}", element );
39
40           // sort the filtered results into descending order
41        var sortFilteredResults =
42           from value in filtered
43           orderby value descending
44           select value;
45
46        // display the sorted results
47        Console.Write(
48        "\nValues greater than 4, descending order (separately):" );
49        foreach ( var element in sortFilteredResults )
50             Console.Write( " {0}", element );
51
52           // filter original array and sort in descending order
53        var sortAndFilter =
54           from value in values
55           where value > 4
56           orderby value descending
57           select value;
58
59        // display the filtered and sorted results
60        Console.Write(
61        "\nValues greater than 4, descending order (one query):" );
62        foreach ( var element in sortAndFilter )
63        Console.Write( " {0}", element );
64
65        Console.WriteLine();
66     } // end Main
67  } // end class LINQWithSimpleTypeArray
```

```
Original array: 2 9 5 0 3 7 1 4 8 5
Array values greater than 4: 9 5 7 8 5
Original array, sorted: 0 1 2 3 4 5 5 7 8 9
Values greater than 4, descending order (separately): 9 8 7 5 5
Values greater than 4, descending order (one query): 9 8 7 5 5
```

Fig. 6.3 | LINQ to Objects using an `int` array.

The from Clause and Implicitly Typed Local Variables

A LINQ query begins with a from clause (line 20), which specifies a range variable (value) and the data source to query (values). The range variable represents each item in the data source (one at a time), much like the control variable in a foreach statement. We do not specify the range variable's type. Since it is assigned one element at a time from the array values, which is an int array, the compiler determines that the range variable value should be of type int. This is a C# feature called implicitly typed local variables, which enables the compiler to infer a local variable's type based on the context in which it's used.

Introducing the range variable in the from clause at the beginning of the query allows the IDE to provide IntelliSense while you write the rest of the query. The IDE knows the range variable's type, so when you enter the range variable's name followed by a dot (.) in the code editor, the IDE can display the range variable's methods and properties.

The var Keyword and Implicitly Typed Local Variables

You can also declare a local variable and let the compiler infer the variable's type based on the variable's initializer. To do so, the var keyword is used in place of the variable's type when declaring the variable. Consider the declaration

```
var x = 7;
```

Here, the compiler infers that the variable x should be of type int, because the compiler assumes that whole-number values, like 7, are of type int. Similarly, in the declaration

```
var y = -123.45;
```

the compiler infers that y should be of type double, because the compiler assumes that floating-point number values, like -123.45, are of

type double. Typically, implicitly typed local variables are used for more complex types, such as the collections of data returned by LINQ queries. We use this feature in lines 19, 30, 41 and 53 to enable the compiler to determine the type of each variable that stores the results of a LINQ query. We also use this feature to declare the control variable in the foreach statements at lines 15–16, 26–27, 37–38, 49–50 and 62–63. In each case, the compiler infers that the control variable is of type int because the array values and the LINQ query results all contain int values.

The where Clause

If the condition in the where clause (line 21) evaluates to true, the element is selected—i.e., it's included in the results. Here, the ints in the array are included only if they're greater than 4. An expression that takes an element of a collection and returns true or false by testing a condition on that element is known as a predicate.

The select Clause

For each item in the data source, the select clause (line 22) determines what value appears in the results. In this case, it's the int that the range variable currently represents. A LINQ query typically ends with a select clause.

Iterating Through the Results of the LINQ Query

Lines 26–27 use a foreach statement to display the query results. As you know, a foreach statement can iterate through the contents of an array, allowing you to process each element in the array. Actually, the foreach statement can iterate through the contents arrays, collections and the results of LINQ queries. The foreach statement in lines 26–27 iterates over the query result filtered, displaying each of its items.

LINQ vs. Repetition Statements

It would be simple to display the integers greater than 4 using a repetition statement that tests each value before displaying it. However, this would intertwine the code that selects elements and the code that displays them. With LINQ, these are kept separate, making the code easier to understand and maintain.

The orderby Clause

The orderby clause (line 32) sorts the query results in ascending order. Lines 43 and 56 use the descending modifier in the orderby clause to sort the results in descending order. An ascending modifier also exists but isn't normally used, because it's the default. Any value that can be compared with other values of the same type may be used with the orderby clause. A value of a simple type (e.g., int) can always be compared to another value of the same type; we'll say more about comparing values of reference types in Chapter 12.

The queries in lines 42–44 and 54–57 generate the same results, but in different ways. The first query uses LINQ to sort the results of the query from lines 20–22. The second query uses both the where and orderby clauses. Because queries can operate on the results of other queries, it's possible to build a query one step at a time, and pass the results of queries between methods for further processing.

More on Implicitly Typed Local Variables

Implicitly typed local variables can also be used to initialize arrays without explicitly giving their type. For example, the following statement creates an array of int values:

```
var array = new[] { 32, 27, 64, 18, 95, 14, 90, 70, 60, 37 };
```

There are no square brackets on the left side of the assignment operator, and that new[] is used to specify that the variable is an array.

6.5 Querying an Array of Employee Objects Using LINQ

LINQ is not limited to querying arrays of primitive types such as ints. It can be used with most data types, including strings and user-defined classes. It cannot be used when a query does not have a defined meaning—for example, you cannot use orderby on objects that are not comparable. Figure 6.4 presents the Employee class and uses LINQ to query an array of Employee objects.

```
1     // Fig. 6.4: LINQWithArrayOfObjects.cs
2     // LINQ to Objects using an array of Employee objects.
3     using System;
4     using System.Linq;
5
6     public class LINQWithArrayOfObjects
7     {
8       public struct Employee
9       {
10          public decimal monthlySalaryValue; // monthly salary of employee
11          public string FirstName;
12          public string LastName;
13         // constructor initializes first name, last name and monthly salary
14          public Employee( string first, string last, decimal salary )
15          {
16              FirstName = first;
17              LastName = last;
18              MonthlySalary = salary;
19          } // end constructor
20      }   // end struct
21       public static void Main( string[] args )
22       {
23          // initialize array of employees
24          Employee[] employees = {
25             new Employee( "Jason", "Red", 5000M ),
26             new Employee( "Ashley", "Green", 7600M ),
27             new Employee( "Matthew", "Indigo", 3587.5M ),
28             new Employee( "James", "Indigo", 4700.77M ),
29             new Employee( "Luke", "Indigo", 6200M ),
30             new Employee( "Jason", "Blue", 3200M ),
31             new Employee( "Wendy", "Brown", 4236.4M ) };
32
33          // display all employees
34          Console.WriteLine( "Original array:" );
35          foreach ( var element in employees )
36             Console.WriteLine( element );
37
38          // filter a range of salaries using && in a LINQ query
39          var between4K6K =
40             from e in employees
41             where e.MonthlySalary >= 4000M && e.MonthlySalary <= 6000M
42             select e;
```

```
43
44          // display employees making between 4000 and 6000 per month
45          Console.WriteLine( string.Format(
46             "\nEmployees earning in the range {0:C}-{1:C} per month:",
47             4000, 6000 ) );
48          foreach ( var element in between4K6K )
49             Console.WriteLine( element );
50
51          // order the employees by last name, then first name with LINQ
52          var nameSorted =
53             from e in employees
54             orderby e.LastName, e.FirstName
55             select e;
56
57          // header
58          Console.WriteLine( "\nFirst employee when sorted by name:" );
59
60          // attempt to display the first result of the above LINQ query
61          if (nameSorted.Any() )
62             Console.WriteLine(nameSorted.First() );
63          else
64             Console.WriteLine( "not found" );
65
66          // use LINQ to select employee last names
67          var lastNames =
68             from e in employees
69             select e.LastName;
70
71          // use method Distinct to select unique last names
72          Console.WriteLine( "\nUnique employee last names:" );
73          foreach ( var element in lastNames.Distinct() )
74             Console.WriteLine( element );
75
76          // use LINQ to select first and last names
77          var names =
78             from e in employees
79             select new { e.FirstName, Last = e.LastName };
80
81          // display full names
82          Console.WriteLine( "\nNames only:" );
83          foreach ( var element in names )
84             Console.WriteLine( element );
85
86          Console.WriteLine();
87       } // end Main
88    } // end class LINQWithArrayOfObjects
```

Fig. 6.4 | LINQ to Objects using an array of `Employee` objects

The output of previous program is:

```
Original array:
Jason      Red         $5,000.00
Ashley     Green       $7,600.00
Matthew    Indigo      $3,587.50
James      Indigo      $4,700.77
Luke       Indigo      $6,200.00
Jason      Blue        $3,200.00
Wendy      Brown       $4,236.40

Employees earning in the range $4,000.00-$6,000.00 per month:
Jason      Red         $5,000.00
James      Indigo      $4,700.77
Wendy      Brown       $4,236.40

First employee when sorted by name:
Jason      Blue        $3,200.00

Unique employee last names:
Red
Green
Indigo
Blue
Brown

Names only:
{ FirstName = Jason, Last = Red }
{ FirstName = Ashley, Last = Green }
{ FirstName = Matthew, Last = Indigo }
{ FirstName = James, Last = Indigo }
{ FirstName = Luke, Last = Indigo }
{ FirstName = Jason, Last = Blue }
{ FirstName = Wendy, Last = Brown }
```

6.6 Querying a Generic Collection Using LINQ

You can use LINQ to Objects to query Lists just as arrays. In Fig. 6.5, a List of strings is converted to uppercase and searched for those that begin with "R".

```
1  // Fig. 6.5: LINQWithListCollection.cs
2  // LINQ to Objects using a List< string >.
3  using System;
4  using System.Linq;
5  using System.Collections.Generic;
6
7  public class LINQWithListCollection
8  {
9     public static void Main( string[] args )
10    {
11       // populate a List of strings
12       List< string > items = new List< string >();
13       items.Add( "aQua" ); // add "aQua" to the end of the List
14       items.Add( "RusT" ); // add "RusT" to the end of the List
15       items.Add( "yElLow" ); // add "yElLow" to the end of the List
16       items.Add( "rEd" ); // add "rEd" to the end of the List
```

```
17
18      // convert all strings to uppercase; select those starting with "R"
19      var startsWithR =
20         from item in items
21         let uppercaseString = item.ToUpper()
22         where uppercaseString.StartsWith( "R" )
23         orderby uppercaseString
24         select uppercaseString;
25
26      // display query results
27      foreach ( var item in startsWithR )
28            Console.Write( "{0} ", item );
29
30      Console.WriteLine(); // output end of line
31
32      items.Add( "rUbY" ); // add "rUbY" to the end of the List
33      items.Add( "SaFfRon" ); // add "SaFfRon" to the end of the List
34
35      // display updated query results
36      foreach ( var item in startsWithR )
37            Console.Write( "{0} ", item );
38
39      Console.WriteLine(); // output end of line
40   } // end Main
41 } // end class LINQWithListCollection
```

```
RED RUST
RED RUBY RUST
```

Fig. 6.5 | LINQ to Objects using a `List<string>`.

Line 21 uses LINQ's let clause to create a new range variable. This is useful if you need to store a temporary result for use later in the LINQ query. Typically, let declares a new range variable to which you assign the result of an expression that operates on the query's original range variable. In this case, we use string method ToUpper to convert each item to uppercase, then store the result in the new range variable uppercaseString. We then use the new range variable uppercaseString in the where, orderby and select clauses. The where clause (line 22) uses string method StartsWith to determine whether uppercaseString starts with the character "R". Method StartsWith performs a case- sensitive comparison to determine whether a string starts with the string received as an argument. If uppercaseString starts with "R", method StartsWith returns true, and the element is included in the query results. More powerful string matching can be done using the regular-expression capabilities introduced in Chapter 16, Strings and Charac- ters.

The query is created only once (lines 20–24), yet iterating over the results (lines 27–28 and 36–37) gives two different lists of colors. This demonstrates LINQ's deferred execution—the query executes only when you access the results—such as iterating over them or using the Count method—not when you define the query. This allows you to create a query once and execute it many times. Any changes to the data source are reflected in the results each time the query executes.

There may be times when you do not want this behavior, and want to retrieve a col- lection of the results immediately. LINQ provides extension methods ToArray and ToList for this purpose. These methods execute the query on which they're called and give you the results as an array or List<T>, respectively. These methods can also improve efficiency if you'll be iterating over the results multiple times, as you execute the query only once.

C# has a feature called collection initializers, which provide a convenient syntax (similar to array initializers) for initializing a collection. For example, lines 12–16 of Fig. 6.5 could be replaced with the following statement:

```
List< string > items =
   new List< string > { "aQua", "RusT",  "yEllow", "rEd" };
```

6.7 Computer Files

When data items are stored in a computer system, they can be stored for varying periods of time—temporarily or permanently. Temporary storage is usually called computer memory or random access memory (RAM). When you write a C# program that stores a value in a variable, you are using temporary storage; the value you store is lost when the program ends or the computer loses power. This type of storage is volatile.

Permanent storage, on the other hand, is not lost when a computer loses power; it is nonvolatile. When you write a program and save it to a disk, you are using permanent storage. A computer file is a collection of data stored on a nonvolatile device in a computer system. Files exist on permanent storage devices, such as hard disks, USB drives, reels of magnetic tape, and optical discs, which include CDs and DVDs.

6.7.1 Files Categories

You can categorize files by the way they store data:

- ***Text files** contain data that can be read in a text editor because the data has been encoded using a scheme such as ASCII or Unicode. Text files might include facts and figures used by business programs; when they do, they are also called data files. The C# source programs you have written are stored in text files.*
- ***Binary files** contain data that has not been encoded as text. Their contents are in binary format, which means that you cannot understand them by viewing them in a text editor. Examples include images, music, and the compiled program files with an .exe extension that you have created using this book.*

Although their contents vary, files have many common characteristics, as follows:

- *Each has a name. The name often includes a dot and a file extension that describes the type of the file. For example, .txt isa plain text file, .dat is a data file, and .jpg is an image file in Joint Pictures Expert Group format. Each file has a specific time of creation and a time it was last modified.*
- *Each file occupies space on a section of a storage device; that is, each file has a size. Sizes are measured in bytes. A byte is a small unit of storage; for example, in a simple text file, a byte holds only one character. Because a byte is so small, file sizes usually are expressed in kilobytes (thousands of bytes), megabytes (millions of bytes), or gigabytes (billions of bytes).*