Chapter 9
Introduction to LINQ and the `List` Collection

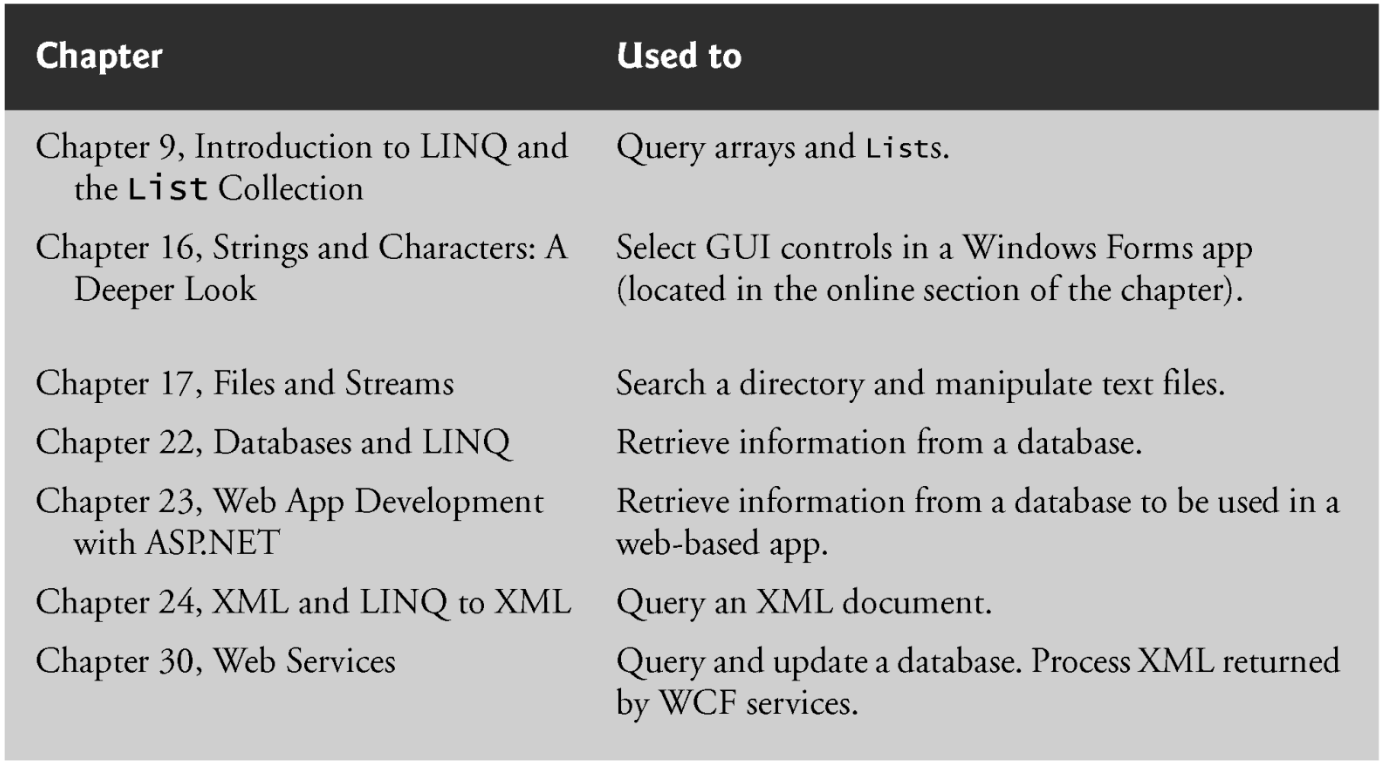
Visual C# 2012 How to Program

9.1 Introduction

- Although commonly used, arrays have limited capabilities.
- A `List` is similar to an array but provides additional functionality, such as **dynamic resizing**.
- A language called SQL is the international standard used to perform **queries** (i.e., to request information that satisfies given criteria) and to manipulate data.
- C#'s new **LINQ** (**Language-Integrated Query**) capabilities allow you to write **query expressions** that retrieve information from a *variety* of data sources, not just databases.
- **LINQ to Objects** can be used to **filter** arrays and `List`s, selecting elements that satisfy a set of conditions

9.1 Introduction (Cont.)

- Figure 9.1 shows where and how we use LINQ throughout the book to retrieve information from many data sources.
- 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 projecting, *sorting*, *grouping* and *filtering* elements.



Chapter	Used to
Chapter 9, Introduction to LINQ and the `List` Collection	Query arrays and `Lists`.
Chapter 16, Strings and Characters: A Deeper Look	Select GUI controls in a Windows Forms app (located in the online section of the chapter).
Chapter 17, Files and Streams	Search a directory and manipulate text files.
Chapter 22, Databases and LINQ	Retrieve information from a database.
Chapter 23, Web App Development with ASP.NET	Retrieve information from a database to be used in a web-based app.
Chapter 24, XML and LINQ to XML	Query an XML document.
Chapter 30, Web Services	Query and update a database. Process XML returned by WCF services.

Fig. 9.1 | LINQ usage throughout the book.

9.2 Querying an Array of int Values Using LINQ

- Figure 9.2 demonstrates querying an array of integers using LINQ.

```
// Fig. 9.2: LINQWithSimpleTypeArray.cs
// LINQ to Objects using an int array.
using System;
using System.Linq;

class LINQWithSimpleTypeArray
{
   public static void Main( string[] args )
   {
      // create an integer array
      int[] values = { 2, 9, 5, 0, 3, 7, 1, 4, 8, 5 };

      // display original values
      Console.Write( "Original array:" );
      foreach ( var element in values )
         Console.Write( " {0}", element );

      // LINQ query that obtains values greater than 4 from the array
      var filtered =
         from value in values
         where value > 4
         select value;

```

Fig. 9.2 | LINQ to Objects using an int array. (Part 1 of 4.)

```
      // display filtered results
      Console.Write( "\nArray values greater than 4:" );
      foreach ( var element in filtered )
         Console.Write( " {0}", element );

      // use orderby clause to original values in ascending order
      var sorted =
         from value in values
         orderby value
         select value;

      // display sorted results
      Console.Write( "\nOriginal array, sorted:" );
      foreach ( var element in sorted )
         Console.Write( " {0}", element );

      // sort the filtered results into descending order
      var sortFilteredResults =
         from value in filtered
         orderby value descending
         select value;

```

Fig. 9.2 | LINQ to Objects using an `int` array. (Part 2 of 4.)

```
         // display the sorted results
         Console.Write(
            "\nValues greater than 4, descending order (separately):" );
         foreach ( var element in sortFilteredResults )
            Console.Write( " {0}", element );

         // filter original array and sort results in descending order
         var sortAndFilter =
            from value in values
            where value > 4
            orderby value descending
            select value;

         // display the filtered and sorted results
         Console.Write(
            "\nValues greater than 4, descending order (one query):" );
         foreach ( var element in sortAndFilter )
            Console.Write( " {0}", element );

         Console.WriteLine();
      } // end Main
   } // end class LINQWithSimpleTypeArray
```

Fig. 9.2 | LINQ to Objects using an `int` array. (Part 3 of 4.)

```
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. 9.2 | LINQ to Objects using an `int` array. (Part 4 of 4.)

9.2 Querying an Array Using LINQ (Cont.)

- Repetition statements that filter arrays focus on the steps required to get the results. This is called **imperative programming**.
- LINQ queries, however, specify the conditions that selected elements must satisfy. This is known as **declarative programming**.
- The `System.Linq` namespace contains the LINQ to Objects provider.

9.2 Querying an Array Using LINQ (Cont.)

- You can declare a local variable and let the compiler infer the variable's type based on the variable's initializer. The **`var` keyword** is used in place of the variable's type when declaring the variable.
- A LINQ query begins with a **`from` clause**, which specifies a **range variable** (`value`) and the data source to query (`values`).
  - The range variable represents each item in the data source, much like the control variable in a `foreach` statement.
- If the condition in the **`where` clause** evaluates to `true`, the element is selected.

9.2 Querying an Array Using LINQ (Cont.)

- A **predicate** is an expression that takes an element of a collection and returns `true` or `false` by testing a condition on that element.
- The **select clause** determines what value appears in the results.

9.2 Querying an Array Using LINQ (Cont.)

- The **orderby clause** sorts the query results in ascending order.
- The **descending** modifier in the `orderby` clause sorts the results in descending order.
- Any value that can be compared with other values of the same type may be used with the `orderby` clause.

9.2 Querying an Array Using LINQ (Cont.)

- The `IEnumerable<T>` interface describes the functionality of any object that can be iterated over and thus offers members to access each element.
- Arrays and collections already implement the `IEnumerable<T>` interface.
- A LINQ query returns an object that implements the `IEnumerable<T>` interface.
- With LINQ, the code that selects elements and the code that displays them are kept separate, making the code easier to understand and maintain.

9.3 Querying an Array of Employee Objects Using LINQ

- LINQ is not limited to querying arrays of simple types such as `integers`.
- Comparable types in .NET are those that implement the `IComparable<T>`.
- All built-in types, such as `string`, `int` and double implement `IComparable<T>`.
- Figure 9.3 presents the Employee class. Figure 9.4 uses LINQ to query an array of Employee objects.

```
// Fig. 9.3: Employee.cs
// Employee class with FirstName, LastName and MonthlySalary properties.
public class Employee
{
   private decimal monthlySalaryValue; // monthly salary of employee

   // auto-implemented property FirstName
   public string FirstName { get; set; }

   // auto-implemented property LastName
   public string LastName { get; set; }

   // constructor initializes first name, last name and monthly salary
   public Employee( string first, string last, decimal salary )
   {
      FirstName = first;
      LastName = last;
      MonthlySalary = salary;
   } // end constructor

```

Fig. 9.3 | Employee class. (Part 1 of 2.)

```
   // property that gets and sets the employee's monthly salary
   public decimal MonthlySalary
   {
      get
      {
         return monthlySalaryValue;
      } // end get
      set
      {
         if ( value >= 0M ) // if salary is nonnegative
         {
            monthlySalaryValue = value;
         } // end if
      } // end set
   } // end property MonthlySalary

   // return a string containing the employee's information
   public override string ToString()
   {
      return string.Format( "{0,-10} {1,-10} {2,10:C}",
         FirstName, LastName, MonthlySalary );
   } // end method ToString
} // end class Employee
```

Fig. 9.3 | Employee class. (Part 2 of 2.)

```
// Fig. 9.4: LINQWithArrayOfObjects.cs
// LINQ to Objects using an array of Employee objects.
using System;
using System.Linq;

public class LINQWithArrayOfObjects
{
   public static void Main( string[] args )
   {
      // initialize array of employees
      Employee[] employees = {
         new Employee( "Jason", "Red", 5000M ),
         new Employee( "Ashley", "Green", 7600M ),
         new Employee( "Matthew", "Indigo", 3587.5M ),
         new Employee( "James", "Indigo", 4700.77M ),
         new Employee( "Luke", "Indigo", 6200M ),
         new Employee( "Jason", "Blue", 3200M ),
         new Employee( "Wendy", "Brown", 4236.4M ) }; // end init list

      // display all employees
      Console.WriteLine( "Original array:" );
      foreach ( var element in employees )
         Console.WriteLine( element );

```

Fig. 9.4 | LINQ to Objects using an array of Employee objects. (Part 1 of 6.)

```
        // filter a range of salaries using && in a LINQ query
        var between4K6K =
           from e in employees
           where e.MonthlySalary >= 4000M && e.MonthlySalary <= 6000M
           select e;

        // display employees making between 4000 and 6000 per month
        Console.WriteLine( string.Format(
           "\nEmployees earning in the range {0:C}-{1:C} per month:",
           4000, 6000 ) );
        foreach ( var element in between4K6K )
           Console.WriteLine( element );

        // order the employees by last name, then first name with LINQ
        var nameSorted =
           from e in employees
           orderby e.LastName, e.FirstName
           select e;

        // header
        Console.WriteLine( "\nFirst employee when sorted by name:" );

```

Fig. 9.4 | LINQ to Objects using an array of `Employee` objects. (Part 2 of 6.)

```
         // attempt to display the first result of the above LINQ query
         if ( nameSorted.Any() )
            Console.WriteLine( nameSorted.First() );
         else
            Console.WriteLine( "not found" );

         // use LINQ to select employee last names
         var lastNames =
            from e in employees
            select e.LastName;

         // use method Distinct to select unique last names
         Console.WriteLine( "\nUnique employee last names:" );
         foreach ( var element in lastNames.Distinct() )
            Console.WriteLine( element );

         // use LINQ to select first and last names
         var names =
            from e in employees
            select new { e.FirstName, Last = e.LastName };

```

Fig. 9.4 | LINQ to Objects using an array of Employee objects. (Part 3 of 6.)

```
         // display full names
         Console.WriteLine( "\nNames only:" );
         foreach ( var element in names )
            Console.WriteLine( element );

         Console.WriteLine();
      } // end Main
   } // end class LINQWithArrayOfObjects
```

Fig. 9.4 | LINQ to Objects using an array of `Employee` objects. (Part 4 of 6.)

```
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
```

Fig. 9.4 | LINQ to Objects using an array of `Employee` objects. (Part 5 of 6.)

```
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 }
```

Fig. 9.4 | LINQ to Objects using an array of `Employee` objects. (Part 6 of 6.)

9.3 Querying an Array of Employee Objects Using LINQ (Cont.)

- A `where` clause can access the properties of the range variable.
- The conditional AND (&&) operator can be used to combine conditions.
- An `orderby` clause can sort the results according to multiple properties, specified in a comma-separated list.

9.3 Querying an Array of Employee Objects Using LINQ (Cont.)

- The query result's `Any` method returns true if there is at least one element, and false if there are no elements.
- The query result's `First` method returns the first element in the result.
- The `Count` method of the query result returns the number of elements in the results.
- The `select` clause can be used to select a member of the range variable rather than the range variable itself.
- The `Distinct method` removes duplicate elements, causing all elements in the result to be unique.

9.3 Querying an Array of Employee Objects Using LINQ (Cont.)

- The select clause can create a new object of **anonymous type** (a type with no name), which the compiler generates for you based on the properties listed in the curly braces (`{}`).
- By default, the name of the property being selected is used as the property's name in the result.
- You can specify a different name for the property inside the anonymous type definition.

9.3 Querying an Array of Employee Objects Using LINQ (Cont.)

- The preceding query is an example of a **projection**—it performs a transformation on the data.
- The transformation creates new objects containing only the `FirstName` and `Last` properties.
- Transformations can also manipulate the data.
  - For example, you could give all employees a 10% raise by multiplying their `MonthlySalary` properties by 1.1.

9.3 Querying an Array of Employee Objects Using LINQ (Cont.)

- Implicitly typed local variables allow you to use anonymous types because you do not have to explicitly state the type when declaring such variables.
- When the compiler creates an anonymous type, it automatically generates a `ToString` method that returns a `string` representation of the object.

9.5 Querying a Generic Collection Using LINQ

- You can use LINQ to Objects to query Lists just as arrays.
- In Fig. 9.7, a List of strings is converted to uppercase and searched for those that begin with "R".

```
// Fig. 9.7: LINQWithListCollection.cs
// LINQ to Objects using a List< string >.
using System;
using System.Linq;
using System.Collections.Generic;

public class LINQWithListCollection
{
   public static void Main( string[] args )
   {
      // populate a List of strings
      List< string > items = new List< string >();
      items.Add( "aQua" ); // add "aQua" to the end of the List
      items.Add( "RusT" ); // add "RusT" to the end of the List
      items.Add( "yElLow" ); // add "yElLow" to the end of the List
      items.Add( "rEd" ); // add "rEd" to the end of the List

      // convert all strings to uppercase; select those starting with "R"
      var startsWithR =
         from item in items
         let uppercaseString = item.ToUpper()
         where uppercaseString.StartsWith( "R" )
         orderby uppercaseString
         select uppercaseString;
```

Fig. 9.7 | LINQ to Objects using a List<string>. (Part 1 of 2.)

```
         // display query results
         foreach ( var item in startsWithR )
            Console.Write( "{0} ", item );

         Console.WriteLine(); // output end of line

         items.Add( "rUbY" ); // add "rUbY" to the end of the List
         items.Add( "SaFfRon" ); // add "SaFfRon" to the end of the List

         // display updated query results
         foreach ( var item in startsWithR )
            Console.Write( "{0} ", item );

         Console.WriteLine(); // output end of line
      } // end Main
   } // end class LINQWithListCollection
```

```
RED RUST
RED RUBY RUST
```

Fig. 9.7 | LINQ to Objects using a `List<string>`. (Part 2 of 2.)

9.5 Querying a Generic Collection Using LINQ (Cont.)

- LINQ's `let` **clause** can be used to create a new range variable to store a temporary result for use later in the LINQ query.
- The `string` method **`ToUpper`** to converts a string to uppercase.
- The `string` method **`StartsWith`** performs a case sensitive comparison to determine whether a `string` starts with the `string` received as an argument.

9.5 Querying a Generic Collection Using LINQ (Cont.)

- LINQ uses **deferred execution**—the query executes *only* when you access the results, *not* when you define the query.
- LINQ extension methods `ToArray` and `ToList` immediately execute the query on which they are called.
  - These methods execute the query only once, improving efficiency.