7.9 Case Study: Random-Number Generation

- Objects of class **`Random`** can produce random `byte`, `int` and `double` values.
- Method `Next` of class `Random` generates a random `int` value.
- The values returned by `Next` are actually **pseudorandom numbers**—a sequence of values produced by a complex mathematical calculation.
- The calculation uses the current time of day to **seed** the random-number generator.

7.9 Case Study: Random-Number Generation (Cont.)

- If you supply the `Next` method with an argument—called the **scaling factor**—it returns a value from 0 up to, but not including, the argument's value.
- You can also **shift** the range of numbers produced by adding a **shifting value** to the number returned by the `Next` method.
- Finally, if you provide `Next` with two `int` arguments, it returns a value from the first argument's value up to, but not including, the second argument's value.

Outline

Rolling a Six-Sided Die

- Figure 7.7 shows two sample outputs of an application that simulates 20 rolls of a six-sided die and displays each roll's value.

RandomIntegers.cs

(1 of 2)

```
1  // Fig7.7: RandomIntegers.cs
2  // Shifted and scaled random integers.
3  using System;
4
5  public class RandomIntegers
6  {
7    public static void Main( string[] args )
8    {
9      Random randomNumbers = new Random(); // random-number generator
10     int face; // stores each random integer generated
11
12       // loop 20 times
13     for ( int counter = 1; counter <= 20; counter++ )
14     {
15       // pick random integer from 1 to 6
```

Create the `Random` object `randomNumbers` to produce random values.

Fig. 7.7 | Shifted and scaled random integers. (Part 1 of 2.)

Outline

```
16       face = randomNumbers.Next( 1, 7 );
17
18       Console.Write( "{0} ", face ); // display generated value
19
20          // if counter is divisible by 5, start a new line of output
21       if ( counter % 5 == 0 )
22          Console.WriteLine();
23    } // end for
24  } // end Main
25 } // end class RandomIntegers
```

```
3  3  3  1  1
2  1  2  4  2
2  3  6  2  5
3  4  6  6  1
```

```
6  2  5  1  3
5  2  1  6  5
4  1  6  1  3
3  1  4  3  4
```

RandomIntegers.cs

(2 of 2)

Call `Next` with two arguments.

Fig. 7.7 | Shifted and scaled random integers. (Part 2 of 2.)

7.9 Case Study: Random-Number Generation (Cont.)

7.9.1 Scaling and Shifting Random Numbers

- Given two arguments, the next method allows scaling and shifting as follows:

`number` = `randomNumbers.Next(` *shiftingValue*, *shiftingValue* + *scalingFactor*);

  - *shiftingValue* specifies the first number in the desired range of consecutive integers.
  - *scalingFactor* specifies how many numbers are in the range.

7.9 Case Study: Random-Number Generation (Cont.)

- To choose integers at random from sets of values other than ranges of consecutive integers, it is simpler to use the version of the `Next` method that takes only one argument:

`number` = *shiftingValue* +
differenceBetweenValues * `randomNumbers.Next(` *scalingFactor*);

  - *shiftingValue* specifies the first number in the desired range of values.
  - *differenceBetweenValues* represents the difference between consecutive numbers in the sequence.
  - *scalingFactor* specifies how many numbers are in the range.

7.9 Case Study: Random-Number Generation (Cont.)

7.9.2 Random-Number Repeatability for Testing and Debugging

- The calculation that produces the pseudorandom numbers uses the time of day as a **seed value** to change the sequence's starting point.
- You can pass a seed value to the `Random` object's constructor.
- Given the same seed value, the `Random` object will produce the same sequence of random numbers.

9

Introduction to LINQ and Generic Collections

9.1 Introduction

- Although commonly used, arrays have limited capabilities.
- `List`s are similar to arrays but provide additional functionality, such as **dynamic resizing**.
- Traditionally, programs used **SQL queries** to access a database.
- C#'s new **LINQ** (**Language-Integrated Query**) capabilities allow you to write **query expressions** that retrieve information from many 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.

Chapter	Used to
Chapter 9, Introduction to LINQ and Generic Collections	Query arrays and `List`s.
Chapter 18, Strings, Characters and Regular Expressions	Select GUI controls in a Windows Forms application.
Chapter 19, Files and Streams	Search a directory and manipulate text files.
Chapter 20, XML and LINQ to XML	Query an XML document.

Fig. 9.1 | LINQ usage throughout the book. (Part 1 of 2.)

9.1 Introduction (Cont.)

Chapter	Used to
Chapter 21, Databases and LINQ to SQL	Retrieve information from a database; insert data into a database.
Chapter 22, ASP.NET 3.5 and ASP.NET AJAX	Retrieve information from a database to be used in a web-based application.
Chapter 23, Windows Communication Foundation (WCF) Web Services	Query and update a database. Process XML returned by WCF services.
Chapter 24, Silverlight, Rich Internet Applications and Multimedia	Process XML returned by web services to a Silverlight application.

Fig. 9.1 | LINQ usage throughout the book. (Part 2 of 2.)

- 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.

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.)

- 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.
- 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.

Outline

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

LINQWithSimpleTypeArray.cs

(1 of 5)

```
1  // Fig. 9.2: LINQWithSimpleTypeArray.cs
2  // LINQ to Objects using an Integer array.
3  using System;
4  using System.Linq;
5  using System.Collections.Generic;
6
7  class LINQWithSimpleTypeArray
8  {
9    public static void Main( string[] args )
10   {
11       // create an integer array
12     int[] values = { 2, 9, 5, 0, 3, 7, 1, 4, 8, 5 };
13
14     Display( values, "Original array:" ); // display original values
15
```

Fig. 9.2 | LINQ to Objects using an `int` array. (Part 1 of 5.)

Outline

LINQWithSimpleTypeArray.cs

(2 of 5)

```
16      // LINQ query that obtains values greater than 4 from the array
17      var filtered =
18         from value in values
19         where value > 4
20         select value;
21
22      // display filtered results
23      Display( filtered, "Array values greater than 4:" );
24
25      // use orderby clause to sort original array in ascending order
26      var sorted =
27         from value in values
28         orderby value
29         select value;
30
31      // display sorted results
32      Display( sorted, "Original array, sorted:" );
33
```

A LINQ query begins with a **`from` clause**, which specifies a **range variable** (`value`) and the data source to query (`values`).

If the condition in the **`where` clause** evaluates to `true`, the element is selected.

The **`select` clause** determines what value appears in the results.

The **`orderby` clause** sorts the query results in ascending order.

Fig. 9.2 | LINQ to Objects using an `int` array. (Part 2 of 5.)

Outline

LINQWithSimpleTypeArray.cs

(3 of 5)

```
34       // sort the filtered results into descending order
35    var sortFilteredResults =
36       from value in filtered
37       orderby value descending
38       select value;
39
40       // display the sorted results
41    Display( sortFilteredResults,
42       "Values greater than 4, descending order (separately):" );
43
44       // filter original array and sort in descending order
45    var sortAndFilter =
46       from value in values
47       where value > 4
48       orderby value descending
49       select value;
50
```

The `descending` modifier in the `orderby` clause sorts the results in descending order.

Fig. 9.2 | LINQ to Objects using an `int` array. (Part 3 of 5.)

Outline

LINQWithSimpleTypeArray.cs

(4 of 5)

```
51        // display the filtered and sorted results
52        Display( sortAndFilter,
53           "Values greater than 4, descending order (one query):" );
54     } // end Main
55
56     // display a sequence of integers with the specified header
57     public static void Display(
58        IEnumerable< int > results, string header )
59     {
60        Console.Write( "{0}", header ); // display header
61
62        // display each element, separated by spaces
63        foreach ( var element in results )
64           Console.Write( "{0}", element );
```

Fig. 9.2 | LINQ to Objects using an `int` array. (Part 4 of 5.)

Outline

LINQWithSimpleTypeArray.cs

(5 of 5)

```
65
66          Console.WriteLine(); // add end of line
67       } // end method Display
68    } // 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. 9.2 | LINQ to Objects using an `int` array. (Part 5 of 5.)

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 `Display` method takes an `IEnumerable<int>` object as an argument.
  - The type `int` enclosed in angle brackets after the type name indicates that this `IEnumerable` may only hold integers.
  - Any type may be used as a **type argument** in this manner—types can be passed as arguments to **generic types** just as objects are passed as arguments to methods.

9.2 Querying an Array Using LINQ (Cont.)

- **Interfaces** define and standardize the ways in which people and systems can interact with one another.
- A C# interface describes a set of methods that can be called on an object.
- A class that implements an interface must define each method in the interface with a signature identical to the one in the interface definition.

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 methods 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.

9.2 Querying an Array Using LINQ (Cont.)

- LINQ is not limited to querying arrays of primitive 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>`.

Outline

- Figure 9.3 presents the `Employee` class.

`Employee.cs`

(1 of 3)

```
1  // Fig. 9.3: Employee.cs
2  // Employee class with FirstName, LastName and MonthlySalary properties.
3  public class Employee
4  {
5     private decimal monthlySalaryValue; // monthly salary of employee
6
7     // auto-implemented property FirstName
8     public string FirstName { get; set; }
9
10    // auto-implemented property LastName
11    public string LastName { get; set; }
12
13    // constructor initializes first name, last name and monthly salary
14    public Employee( string first, string last, decimal salary )
15    {
```

Fig. 9.3 | `Employee` class with `FirstName`, `LastName` and `MonthlySalary` properties. (Part 1 of 3.)

Outline

Employee.cs

(2 of 3)

```
16          FirstName = first;
17          LastName = last;
18          MonthlySalary = salary;
19       } //end constructor
20
21    // property that gets and sets the employee's monthly salary
22    public decimal MonthlySalary
23    {
24       get
25       {
26          return monthlySalaryValue;
27       } // end get
28       set
29       {
30          if ( value >= 0M ) // if salary is nonnegative
31          {
32             monthlySalaryValue = value;
33          } // end if
34       } // end set
```

Fig. 9.3 | Employee class with FirstName, LastName and MonthlySalary properties. (Part 2 of 3.)

Outline

Employee.cs

(3 of 3)

```
35    } // end property MonthlySalary
36
37    // return a String containing the employee's information
38    public override string ToString()
39    {
40       return string.Format( "{0,-10} {1,-10} {2,10:C}"
41          FirstName, LastName, MonthlySalary );
42    } // end method ToString
43 } //  end class Employee
```

Fig. 9.3 | Employee class with FirstName, LastName and MonthlySalary properties. (Part 3 of 3.)

Outline

- Figure 9.4 uses LINQ to query an array of `Employee` objects.

LINQWithArrayOfObjects.cs

(1 of 5)

```
1  // Fig. 9.4: LINQWithArrayOfObjects.cs
2  // LINQ to Objects using an array of Employee objects.
3  using System;
4  using System.Linq;
5  using System.Collections.Generic;
6
7  public class LINQWithArrayOfObjects
8  {
9     public static void Main( string[] args )
10    {
```

Fig. 9.4 | LINQ to Objects using an array of Employee objects. (Part 1 of 5.)

Outline

```
11       // initialize array of employees
12       Employee[] employees = {
13          new Employee( "Jason", "Red", 5000M ),
14          new Employee( "Ashley", "Green", 7600M ),
15          new Employee( "Matthew", "Indigo", 3587.5M ),
16          new Employee( "James", "Indigo", 4700.77M ),
17          new Employee( "Luke", "Indigo", 6200M ),
18          new Employee( "Jason", "Blue", 3200M ),
19          new Employee( "Wendy", "Brown", 4236.4M ) }; // end init list
20
21       Display( employees, "Original array" ); // display all employees
22
23       // filter a range of salaries using && in a LINQ query
24       var between4K6K =
25          from e in employees
26          where e.MonthlySalary >= 4000M && e.MonthlySalary <= 6000M
27          select e;
28
29       // display employees making between 4000 and 6000 per month
30       Display( between4K6K, string.Format(
31          "Employees earning in the range {0:C}-{1:C} per month",
32          4000, 6000 ) );
```

LINQWithArrayOfObjects.cs

(2 of 5)

A `where` clause can access the properties of the range variable.

Fig. 9.4 | LINQ to Objects using an array of Employee objects. (Part 2 of 5.)

Outline

```
33
34        // order the employees by last name, then first name with LINQ
35     var nameSorted =
36        from e in employees
37        orderby e.LastName, e.FirstName
38        select e;
39
40        // header
41     Console.WriteLine( "First employee when sorted by name:" );
42
43        // attempt to display the first result of the above LINQ query
44     if ( nameSorted.Any() )
45        Console.WriteLine( nameSorted.First().ToString() + "\n" );
46     else
47        Console.WriteLine( "not found\n" );
48
49        // use LINQ to select employee last names
50     var lastNames =
51        from e in employees
52        select e.LastName;
53
```

LINQWithArrayOfObjects.cs

(3 of 5)

An `orderby` clause can sort the results according to multiple properties, specified in a comma-separated list.

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 (line 45) returns the first element in the result

The `select` clause can be used to select a member of the range variable rather than the range variable itself.

Fig. 9.4 | LINQ to Objects using an array of Employee objects. (Part 3 of 5.)

Outline

```
54         // use method Distinct to select unique last names
55         Display( lastNames.Distinct(), "Unique employee last names" );
56
57         // use LINQ to select first and last names
58         var names =
59            from e in employees
60            select new { e.FirstName, Last = e.LastName };
61
62         Display( names, "Names only" ); // display full names
63      } // end Main
64
65      // display a sequence of any type, each on a separate line
66      public static void Display< T >(
67         IEnumerable< T > results, string header )
68      {
69         Console.WriteLine( "{0}:", header ); // display header
70
71         // display each element, separated by spaces
72         foreach ( T element in results )
73            Console.WriteLine( element );
74
75         Console.WriteLine(); // add a blank line
76      } // end method Display
77   } // end class LINQWithArrayOfObjects
```

LINQWithArrayOfObjects.cs

(4 of 5)

The **Distinct method** removes duplicate elements, causing all elements in the result to be unique.

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 ({}).

To define a generic method, you must specify a **type parameter list** which contains one or more type parameters separated by commas.

Fig. 9.4 | LINQ to Objects using an array of Employee objects. (Part 4 of 5.)

```
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 }
```

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

Outline

LINQWithArrayOf Objects.cs

(5 of 5)

9.2 Querying an Array 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.

9.2 Querying an Array 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 (line 45) 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.

9.2 Querying an Array 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.

9.2 Querying an Array 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.

9.2 Querying an Array Using LINQ (Cont.)

- **Generic methods** enable you to create a single method definition that can be called with arguments of many types.
- To define a generic method, you must specify a **type parameter list** which contains one or more type parameters separated by commas.
- A **type parameter** is a placeholder for a type argument. They can be used to declare return types, parameter types and local variable types in generic method declarations.

9.2 Querying an Array Using LINQ (Cont.)

  - Can only appear once in the type-parameter list.
  - Can appear more than once in the method's parameter list and body
  - Can be the method's return type
- Type-parameter names must match throughout a method, but need not be unique among different generic methods.

Common Programming Error 9.1

If you forget to include the type-parameter list when declaring a generic method, the compiler will not recognize the type-parameter names when they're encountered in the method, causing compilation errors.

9.3 Introduction to Collections

- The .NET Framework Class Library provides **collections**, which are used to store groups of related objects.
- Collections provide efficient methods that organize, store and retrieve your data without requiring knowledge of how the data is being stored.
- The collection class `List<T>` (from namespace `System.Collections.Generic`) does not need to be reallocated to change its size.

9.3 Introduction to Collections (Cont.)

- `List<T>` is called a **generic class** because it can be used with any type of object.
- `T` is a placeholder for the type of the objects stored in the list.
- Figure 9.5 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.
`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`.

Fig. 9.5 | Some methods and properties of class `List<T>`. (Part 1 of 2.)

9.3 Introduction to Collections (Cont.)

Method or property	Description
`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 ofelements the List currently contains (`Count`).

Fig. 9.5 | Some methods and properties of class `List<T>`. (Part 2 of 2.)

Outline

- Figure 9.6 demonstrates dynamically resizing a `List` object.

`ListCollection.cs`

(1 of 4)

```
1  // Fig. 9.6: 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
```

The **Add** method appends its argument to the end of the `List`.

The **Insert** method inserts a new element at the specified position.

Fig. 9.6 | Generic `List<T>` collection demonstration. (Part 1 of 4.)

Outline

ListCollection.cs

(2 of 4)

```
16        // header
17        Console.Write(
18           "Display list contents with counter-controlled loop:" );
19
20        // display the colors in the list
21        for ( int i = 0; i < items.Count; i++ )
22           Console.Write( "{0} ", items[ i ] );
23
24        // display colors using foreach in the Display method
25        Display( items,
26           "\nDisplay list contents with foreach statement:" );
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        // display the List
31        Display( items, "List with two new elements:" );
32
33        items.Remove( "yellow" ); // remove the first "yellow"
34        // display List
35        Display( items, "Remove first instance of yellow:" );
36
37        items.RemoveAt( 1 ); // remove item at index 1
38        // display List
39        Display( items, "Remove second list element (green):" );
40
```

`List`s can be indexed like arrays by placing the index in square brackets after the `List` variable's name.

The **`Remove`** method is used to remove the first instance of an element with a specific value.

`RemoveAt` removes the element at the specified index; all elements above that index are shifted down by one.

Fig. 9.6 | Generic `List<T>` collection demonstration. (Part 2 of 4.)

Outline

```
41        // check if a value is in the List
42        Console.WriteLine("\"red\" is {0} in the list",
43           items.Contains("red") ? string.Empty : "not ");
44
45        // display number of elements in the List
46        Console.WriteLine("Count: {0}", items.Count);
47
48        // display the capacity of the List
49        Console.WriteLine("Capacity: {0}", items.Capacity);
50     } // end Main
51
52     // display the List's elements on the console
53     public static void Display(List< string > items, string header)
54     {
55        Console.Write(header); // display header
56
57        // display each element in items
58        foreach (var item in items)
59           Console.Write("{0}", item);
60
61        Console.WriteLine(); // display end of line
62     } // end method Display
63  } // end class ListCollection
```

ListCollection.cs

(3 of 4)

The **Contains** method returns true if the element is found in the List, and false otherwise.

The **Capacity** property indicates how many items the **List** can hold without growing.

Fig. 9.6 | Generic List<T> collection demonstration. (Part 3 of 4.)

Outline

ListCollection.cs

(4 of 4)

```
61         Console.WriteLine(); // display end of line
62      } // end method Display
63   } // 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. 9.6 | Generic `List<T>` collection demonstration. (Part 4 of 4.)

9.3 Introduction to Collections (Cont.)

- 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 is to be inserted at the specified index.
  - All elements at the specified index and above are shifted up by one position.

9.3 Introduction to Collections (Cont.)

- 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 **`Remove`** method is used to remove the first instance of an element with a specific value.
  - If no such element is in the List, `Remove` does nothing.
- **`RemoveAt`** removes the element at the specified index; all elements above that index are shifted down by one.

9.3 Introduction to Collections (Cont.)

- The `Contains` method returns `true` if the element is found in the `List`, and `false` otherwise.
- `Contains` compares its argument to each element of the `List` in order, so using `Contains` on a large `List` is inefficient.
- The `Capacity` property indicates how many items the `List` can hold without growing.
- `List` is implemented using an array behind the scenes. When the `List` grows, it must create a larger internal array and copy each element to the new array.
- A `List` grows only when an element is added and there is no space for the new element.
- The List doubles its `Capacity` each time it grows.

- 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".

Outline

LINQWithList Collection.cs

(1 of 2)

```
1  // Fig. 9.7: 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 with random case
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
```

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

```
21        let uppercasedString = item.ToUpper()
22        where uppercasedString.StartsWith("R")
23        orderby uppercasedString
24        select uppercasedString;
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
```

LINQWithList Collection.cs

(2 of 2)

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

9.4 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.

9.4 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.