Introduction to **LINQ** & the **List Collections**

Source : [BOOK] How to programming C# 2012

In this chapter you’ll learn

- Learn basic LINQ concepts.
- Query an array using LINQ.
- Learn basic .NET collections concepts.
- Create and use a generic List collection.
- Query a generic List collection using LINQ.

Introduction

C#’s **LINQ (Language Integrated Query)** capabilities.

LINQ allows you to **write query expressions**, similar to **SQL queries**, that **retrieve information** from a variety of **data sources**, not just databases.

We use **LINQ to Objects** in this chapter to query arrays and Lists, selecting elements that satisfy a set of conditions.

Introduction

Arrays **have limited capabilities** such as **array’s size**

.NET Framework’s collection classes: A set of data structures offer **greater capabilities than traditional arrays**.

Collections are **reusable**, **powerful** and **efficient** and **have been carefully designed** and **tested to ensure correctness** and **good performance**.

List collection: Similar to an array but provides additional functionality, such as **dynamic resizing**

LINQ Providers

LINQ queries may be **used in many different contexts** because of **libraries known as providers**.

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.

LinQ to Wikipedia, LinQ to Twitter, …

Querying an Array of int Values Using **LINQ**

Known as **declarative programming** — as opposed to **imperative programming** in which you **specify the actual steps to perform a task**.

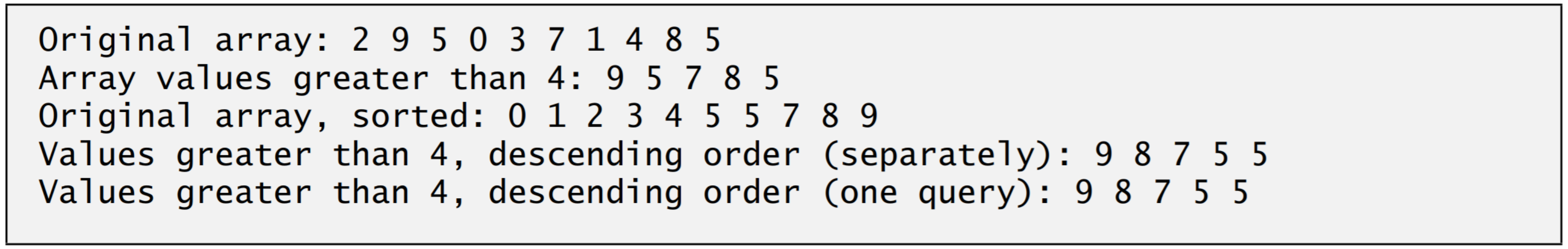
Not require specify how those results are obtained—the C# compiler generates all the necessary code, which is one of the great strengths of LINQ.

To use LINQ capabilities, you must import the **System.Linq** namespace.

// Fig. 9.2: LINQWithSimpleTypeArray.cs

```
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;

      // 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;

      // 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
```

Result



```
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
```

The *from* Clause

```
var filtered =
        from value in values
```

A **LINQ query** begins with a ***from*** clause: 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's **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

implicitly typed local variables: Enables the compiler to infer a local variable's type based on the context in which it's used.

The *var* Keyword and Implicitly Typed Local Variables - Continue

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 to **let the compiler determine the type of each variable that stores the results of a LINQ query**.

```
var filtered =
        from value in values
```

The *where* Clause

```
var filtered =
        from value in values
        where value > 4
```

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

```
var filtered =
        from value in values
        where value > 4
        select value;
```

For each item in the data source, the ***select*** clause **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**

```
foreach ( var element in filtered )
        Console.Write( " {0}", element );
```

Foreach statement can iterate through the contents of an array

We use a ***foreach*** statement to **display the query results**.

Actually, the ***foreach*** statement can iterate through the contents of **arrays**, **collections** and the results of **LINQ queries**.

The ***foreach*** statement over the query result filtered, displaying each of its items.

The *orderby* Clause

```
var sorted =
        from value in values
        orderby value
        select value;
```

Sorts the query results in **ascending order**.

Use the **descending** modifier in the ***orderby*** clause: `orderby value descending`

An ascending modifier is 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.

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 };
```

Note that 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.

An Aside: Interface **IEnumerable <T>**

As we mentioned, the ***foreach*** statement can iterate through the contents of arrays, collections and LINQ query results. Actually, ***foreach*** iterates over any so-called **IEnumerable** object, **which just happens to be what LINQ queries return**.

IEnumerable <T>

Continue

Software **objects communicate via interfaces**. A C# interface **describes a set of members** that can be called on an object—to tell the object, for example, to **perform some task or return some piece of information**.

The **IEnumerable<T>** interface **describes the functionality of any object that can be iterated** over and thus offers members to access each element.

A class that **implements an interface must implement each member in the interface with a signature identical to the one in the interface definition**.

IEnumerable <T>

Continue

Each **LINQ query returns an IEnumerable object** - can use a *foreach* statement to iterate over the results of any LINQ query.

The notation **<T>** indicates that the interface is a **generic interface** that can be used with any type of data.

Querying an Array of *Employee* Objects Using LINQ

LINQ can be used with most data types, including **strings and user-defined classes**.

Cannot be used when a query does not have a defined meaning—for example, you cannot use orderby on objects that are not *comparable*.

Comparable types in .NET: string, int, char, float, double, and etc.

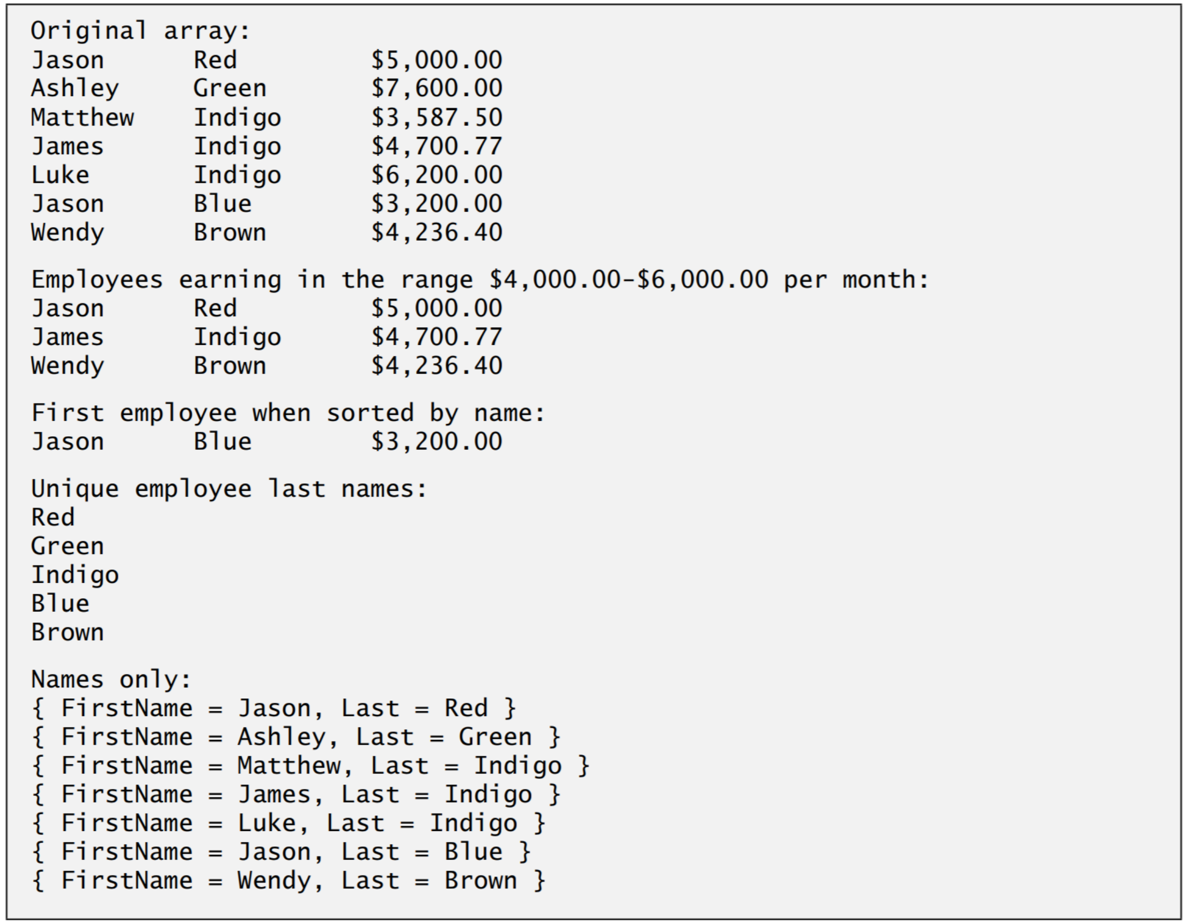
```
// Fig. 9.3: Employee.cs
// Employee class with FirstName, LastName and MonthlySalary properties.
public class Employee
{
   private decimal monthlySalaryValue; // monthly salary of employee

   // 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
   // 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 non-negative
         {
            monthlySalaryValue = value;
         } // end if
      } // end set
   } // end property MonthlySalary

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

```csharp
// 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 );
```

```
// 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:" );

// 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 };

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

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

Result



```
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 }
```

Accessing the Properties of a LINQ Query's Range Variable

```
var between4K6K =
        from e in employees
        where e.MonthlySalary >= 4000M && e.MonthlySalary <= 6000M
        select e;
```

In ***where*** clause, we can **access the properties** of the range variable:

```
where e.MonthlySalary >= 4000M && e.MonthlySalary <= 6000M
```

Any bool expression can be used in a ***where*** clause (**conditional AND (&&))**

`from e in employees ->` The compiler infers that the range variable is of type Employee based on its knowledge that employees was defined as an array of Employee objects:

```
Employee[] employees = {
        new Employee( "Jason", "Red", 5000M ), …
```

Sorting a LINQ Query's Results By Multiple Properties

```
var nameSorted =
        from e in employees
        orderby e.LastName, e.FirstName
        select e;
```

Uses an ***orderby*** clause to sort the results according to multiple properties—specified in a comma-separated list (e.LastName, e.FirstName).

Any, **First** and **Count** Extension Methods

```
if ( nameSorted.Any() )
        Console.WriteLine( nameSorted.First() );
     else
        Console.WriteLine( "not found" );
```

nameSorted.Any() Returns true if there’s at least one element, and false if there are no elements in nameSorted.

The query result’s First method returns the first element in the result. You should check that the query result is not empty **before calling First**.

LINQ defines many more extension methods, such as Count, which returns the number of elements in the results.

Selecting a Property of an Object

```
var lastNames =
        from e in employees
        select e.LastName;
```

Select the range variable’s LastName property rather than the range variable itself

The result consists only the last names (as strings), instead of complete Employee objects

```
foreach ( var element in lastNames.Distinct() )
        Console.WriteLine( element );
```

The Distinct method removes duplicate elements - Display the unique last names.

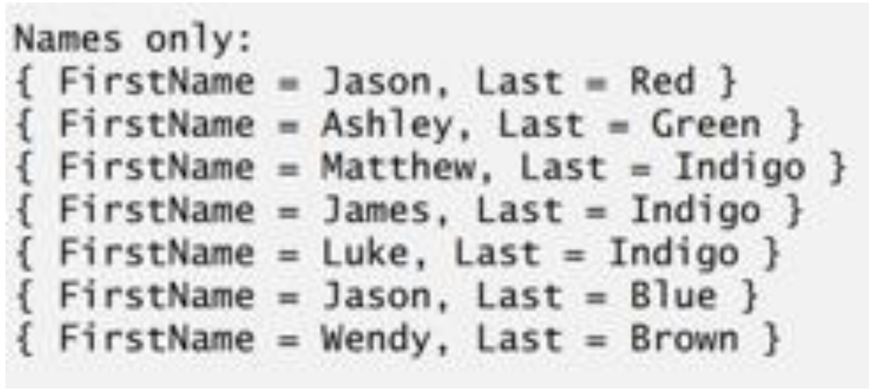
Creating New Types in the *select* Clause of a LINQ Query

```
var names =
        from e in employees
        select new { e.FirstName, Last = e.LastName };
```

Selects the properties FirstName and LastName

creates a new object of an anonymous class, which the compiler generates on the fly based on the properties listed in the curly braces

an example of a projection—it performs a transformation on the data



```
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 }
```

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 app development time.

Types:

- Queue
- List
- LinkedList
- Stack
- …

System.Collections.Generic.List<T>

Continue

For example :

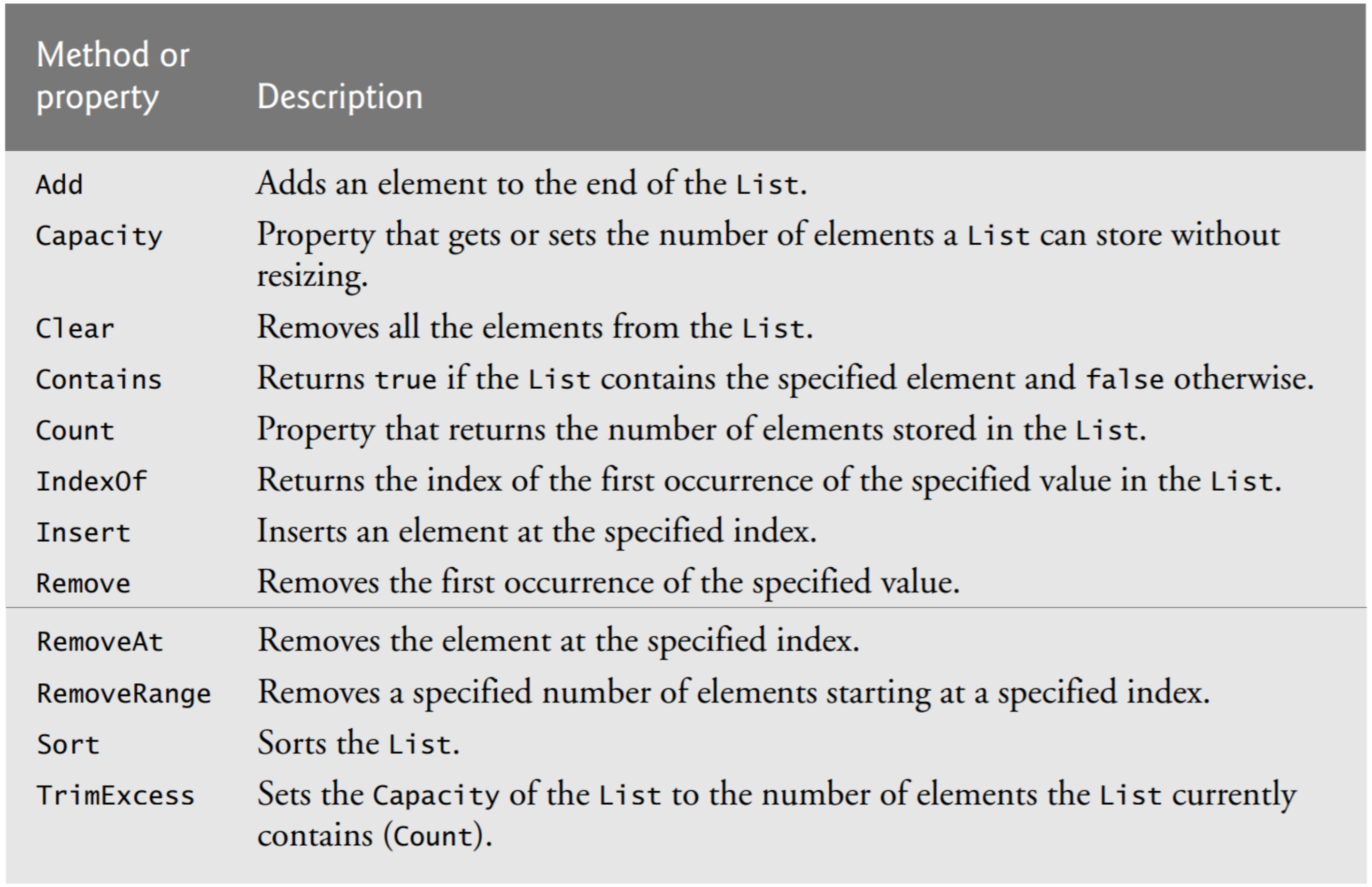
```
List< int > list1;
```

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.

Some 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 and `false` otherwise.
`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`).

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

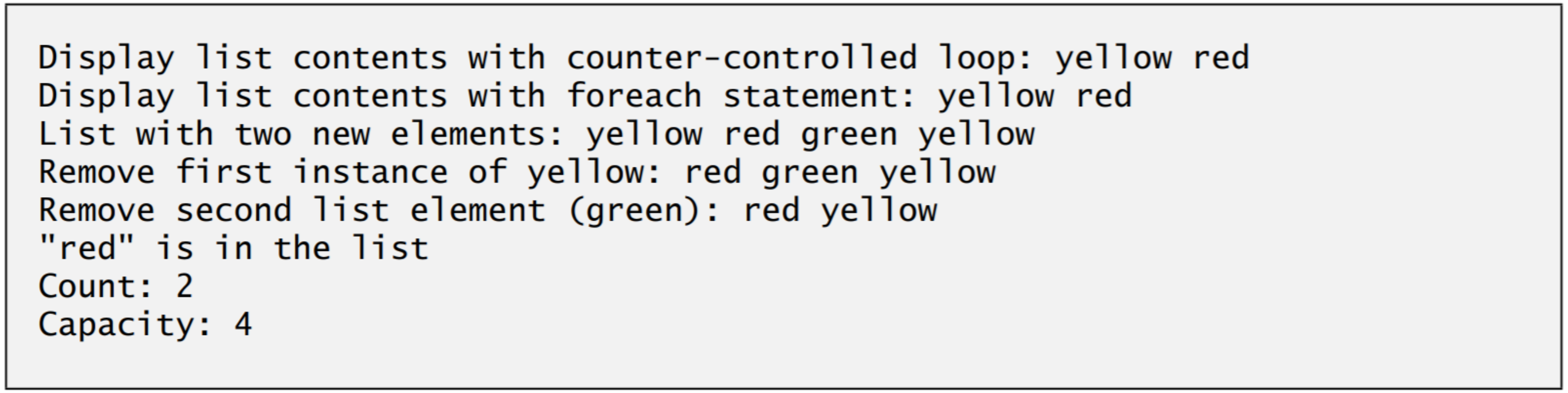
The ***Insert*** method inserts a new element at the specified position. The first argument is an index—as with arrays, collection indices start at zero. The second argument is the value that’s to be inserted at the specified index. The indices of elements at the specified index and above increase by one.

Insert is usually slower than adding an element to the end of the List.

```
// Fig. 9.6: ListCollection.cs
// Generic List collection demonstration.
using System;
using System.Collections.Generic;

public class ListCollection
{
   public static void Main( string[] args )
   {
      // create a new List of strings
      List< string > items = new List< string >();

      items.Add( "red" ); // append an item to the List
      items.Insert( 0, "yellow" ); // insert the value at index 0

      // display the colors in the list
      Console.Write(
         "Display list contents with counter-controlled loop:" );
      for ( int i = 0; i < items.Count; i++ )
         Console.Write( " {0}", items[ i ] );

      // display colors using foreach
      Console.Write(
         "\nDisplay list contents with foreach statement:" );
      foreach ( var item in items )
         Console.Write( " {0}", item );

      items.Add( "green" ); // add "green" to the end of the List
      items.Add( "yellow" ); // add "yellow" to the end of the List
```

```
         // display the List
         Console.Write( "\nList with two new elements:" );
         foreach ( var item in items )
            Console.Write ($" {item}");

         items.Remove( "yellow" ); // remove the first "yellow"

         // display the List
         Console.Write( "\nRemove first instance of yellow:" );
         foreach ( var item in items )
            Console.Write ($" {item}");

         items.RemoveAt( 1 ); // remove item at index 1

         // display the List
         Console.Write( "\nRemove second list element (green):" );
         foreach ( var item in items )
            Console.Write ($" {item}");

         // check if a value is in the List
         Console.WriteLine( "\n\"red\" is {0}in the list",
            items.Contains( "red" ) ? string.Empty : "not " );

         // display number of elements in the List
         Console.WriteLine( "Count: {0}", items.Count );

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

Result



```
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
```

Querying a Generic Collection Using LINQ

```
// 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;

         // 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
```

```
 var startsWithR =
        from item in items
        let uppercaseString = item.ToUpper()
        where uppercaseString.StartsWith( "R" )
        orderby uppercaseString
        select uppercaseString;
```

LINQ's let clause creates 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.

ToUpper method converts 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 uses string method ***StartsWith*** to determine whether ***uppercaseString*** starts with the character "R".

The query is created only, yet iterating over the results **gives two different lists of colors**.

- **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 collection 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 same results multiple times, as you execute the query only once.

Collection Initializers

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. 9.7 could be replaced with the following statement:

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