Introduction to LINQ and Generic Collections
To write it, it took three months; to conceive it three minutes; to collect the data in it—all my life.

— F. Scott Fitzgerald

Science is feasible when the variables are few and can be enumerated; when their combinations are distinct and clear.

— Paul Valéry

You shall listen to all sides and filter them from your self.

— Walt Whitman
The portraitist can select one tiny aspect of everything shown at a moment to incorporate into the final painting.

– Robert Nozick

*List, list, O, list!*

– William Shakespeare

*Be wise to-day; ’t is madness to defer.*

– Edward Young
OBJECTIVES

In this chapter you will learn:

- Basic LINQ concepts.
- How to query an array using LINQ.
- Basic .NET collections concepts.
- How to create and use a generic `List` collection.
- How to write a generic method.
- How to query a generic `List` collection using LINQ.
9.1 Introduction

- Collections offer greater capabilities than arrays.
- Lists automatically change their size.
- Large amounts of data are often stored in a database.
- A database management system (DBMS) is used to control data.
9.1 Introduction (Cont.)

- SQL is the international standard used with relational databases.
- **LINQ (Language-Integrated Query)** allows you to write query expressions to retrieve information.
- **LINQ to Objects** can query arrays and **Lists**, selecting elements that satisfy a set of conditions.
## 9.1 Introduction (Cont.)

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Used to</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chapter 9, Introduction to LINQ and Generic Collections</td>
<td>Query arrays and Lists.</td>
</tr>
<tr>
<td>Chapter 17, Graphics and Multimedia</td>
<td>Select GUI controls in a Windows Forms application.</td>
</tr>
<tr>
<td>Chapter 18, Files and Streams</td>
<td>Search a directory and manipulate text files.</td>
</tr>
<tr>
<td>Chapter 19, XML and LINQ to XML</td>
<td>Query an XML document.</td>
</tr>
<tr>
<td>Chapter 20, Databases and LINQ to SQL</td>
<td>Retrieve information from a database; insert data in a database.</td>
</tr>
<tr>
<td>Chapter 21, ASP.NET and ASP.NET Ajax</td>
<td>Retrieve information from a database to be used in a web-based application.</td>
</tr>
<tr>
<td>Chapter 23, Silverlight, Rich Internet Applications and Multimedia</td>
<td>Process XML returned by web services to a Silverlight application.</td>
</tr>
<tr>
<td>Chapter 24, Data Structures and Generic Collections</td>
<td>Query .NET collections.</td>
</tr>
</tbody>
</table>

**Fig. 9.1** | LINQ usage throughout the book.
Figure 9.2 demonstrates querying an array using LINQ.

```vbnet
Module LINQWithSimpleTypeArray
    Sub Main()
        ' create an integer array
        Dim values() As Integer = {2, 9, 5, 0, 3, 7, 1, 4, 8, 5}
        Display(values, "Original array:")
        ' LINQ query that obtains values greater than 4 from the array
        Dim filtered = From value In values Where value > 4 Select value
        Returns all integers in the array greater than 4.
    End Sub
End Module
```

Fig. 9.2 | LINQ to Objects using an Integer array. (Part 1 of 4.)
' display filtered results
Display(filtered, "Array values greater than 4:"

' use Order By clause to sort original array in ascending order
Dim sorted = _
  From value In values _
  Order By value _
  Select value

Display(sorted, "Original array, sorted:"

' sort the filtered results into descending order
Dim sortFilteredResults = _
  From value In filtered _
  Order By value Descending _
  Select value

' display the sorted results
Display(sortFilteredResults, _
  "Values greater than 4, descending order (separately):")

Fig. 9.2 | LINQ to Objects using an Integer array. (Part 2 of 4.)
Dim sortAndFilter = From value In values Where value > 4 Order By value Descending Select value

' display the filtered and sorted results
Display(sortAndFilter, "Values greater than 4, descending order (one query):")

End Sub ' Main

' display a sequence of integers with the specified header
Sub Display(ByVal results As IEnumerable(Of Integer), ByVal header As String)
Fig. 9.2 | LINQ to Objects using an Integer array. (Part 4 of 4.)

- **OrderBy** sorts results in ascending order.
- The **Descending** query operator sorts results in descending order.
9.2 Querying an Array Using LINQ

• The **From clause** specifies a **range variable** and the data source to query.

• If the condition in the **Where clause** evaluates to **True**, the element is subject to the **Select** clause.

• The **Select clause** specifies what value appears in the results. If omitted, the range variable is selected.
9.2 Querying an Array Using LINQ (Cont.)

• Interfaces define and standardize classes.
• `IEnumerable` is an interface describing a collection of objects (such as an array or a LINQ result).
• A `For Each...Next` statement can iterate through any `IEnumerable` object.
Using LINQ to Query an Array of Employee Objects

- Figure 9.3 presents the Employee class.

```
' Fig. 9.3: Employee.vb
' Employee class with FirstName, LastName and MonthlySalary properties.
Public Class Employee
    Private firstNameValue As String ' first name of employee
    Private lastNameValue As String ' last name of employee
    Private monthlySalaryValue As Decimal ' monthly salary of employee

    ' constructor initializes first name, last name and monthly salary
    Public Sub New(ByVal first As String, ByVal last As String, ByVal salary As Decimal)
        FirstName = first
        LastName = last
        MonthlySalary = salary
    End Sub ' New
```

Fig. 9.3 | Employee class with FirstName, LastName and MonthlySalary properties. (Part 1 of 3.)
Public Property FirstName() As String
  Get
    Return firstNameValue
  End Get
  Set(ByVal value As String)
    firstNameValue = value
  End Set
End Property ' FirstName

Public Property LastName() As String
  Get
    Return lastNameValue
  End Get
  Set(ByVal value As String)
    lastNameValue = value
  End Set
End Property ' LastName

Fig. 9.3 | Employee class with FirstName, LastName, and MonthlySalary properties. (Part 2 of 3.)
' property that gets and sets the employee's monthly salary
Public Property MonthlySalary() As Decimal
    Get
        Return monthlySalaryValue
    End Get

    Set(ByVal value As Decimal)
        If value >= 0 Then ' if salary is non-negative
            monthlySalaryValue = value
        End If
    End Set
End Property ' MonthlySalary

' return a String containing the employee's information
Public Overrides Function ToString() As String
    Return String.Format("{0,-10} {1,-10} {2,10:C}", _
                        FirstName, LastName, MonthlySalary)
End Function ' ToString
End Class ' Employee

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

Sub Main()
    ' Initialize array of employees
    Dim employees As Employee() = { 
        New Employee("Jason", "Red", 5000D), 
        New Employee("Ashley", "Green", 7600D), 
        New Employee("Matthew", "Indigo", 3587.5D), 
        New Employee("James", "Indigo", 4700.77D), 
        New Employee("Luke", "Indigo", 6200D), 
        New Employee("Jason", "Blue", 3200D), 
        New Employee("Wendy", "Brown", 4236.4D) 
    } ' end initializer list

    Display(employees, "Original array") ' display all employees

    ' Filter a range of salaries using AndAlso in a LINQ query
    Dim between4K6K = 
        From e In employees 
        Where e.MonthlySalary >= 4000D AndAlso e.MonthlySalary <= 6000D 
        Select e

The compiler infers that the range variable is of type Employee.
Display employees making between 4000 and 6000 per month:

```
Display(between4K6K, String.Format(_
    "Employees earning in the range {0:C} - {1:C} per month", _
    4000, 6000))
```

Order the employees by last name, then first name with LINQ:

```
Dim nameSorted = _
    From e In employees _
    Order By e.LastName, e.FirstName _
    Select e
```

Console.WriteLine("First employee when sorted by name:") ' header

Attempt to display the first result of the above LINQ query:

```
If nameSorted.Count() = 0 Then
    Console.WriteLine("not found" & vbNewLine)
Else
    Console.WriteLine(nameSorted.First().ToString() & vbNewLine)
End If
```

**Fig. 9.4** | LINQ to Objects using an array of Employee objects. (Part 2 of 5.)
' use LINQ's Distinct clause to select unique last names
Dim lastNames = _
    From e In employees _
    Select e.LastName _
    Distinct

' display unique last names
Display(lastNames, "Unique employee last names")

' use LINQ to select first and last names
Dim names = _
    From e In employees _
    Select e.FirstName, Last = e.LastName

Display(names, "Names only") ' display full names
End Sub ' Main

**Fig. 9.4** | LINQ to Objects using an array of Employee objects. (Part 3 of 5.)
Sub Display(Of T)(ByVal results As IEnumerable(Of T), ByVal header As String)

    Console.WriteLine("{0}:", header) ' display header

    For Each element As T In results
        Console.WriteLine(element)
    Next

    Console.WriteLine() ' add end of line

End Sub ' Display
End Module ' LINQWithArrayOfObjects

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

(continued on next page...)
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 }

Fig. 9.4 | LINQ to Objects using an array of Employee objects. (Part 5 of 5.)
9.2 Querying an Array Using LINQ (Cont.)

- **Count** returns the number of elements in the result.
- The **First** method returns the first element.
- The **Distinct clause** prevents duplicates in results.
9.2 Querying an Array Using LINQ (Cont.)

• In a LINQ Select clause, list an object’s properties in a comma-separated list.
  – The compiler creates a new class with select properties called an anonymous class.
  – Local type inference allows you to use anonymous types without using names.
9.2 Querying an Array Using LINQ (Cont.)

• Overloaded methods can be more compactly coded using a **generic method**.

• Specify a **type parameter list**—placed in parentheses following the method name, begins with keyword `of` and contains one or more type parameters.

• A **type parameter** is a placeholder for an actual type.
  – When you call a generic method, the compiler infers the type.

**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 collection `List(Of T)` can hold a list of whatever type of elements that you want:

```csharp
Dim list1 As List(Of Integer)
Dim list2 As List(Of String)
```

• Classes with this kind of placeholder that can be used with any type are called **generic classes**.
### Fig. 9.5 | Some methods and properties of class `List(Of T)` (Part 1 of 2.)

<table>
<thead>
<tr>
<th>Method or property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td>Adds an element to the end of the <code>List</code>.</td>
</tr>
<tr>
<td>Capacity</td>
<td>Property that gets or sets the number of elements a <code>List</code> can store.</td>
</tr>
<tr>
<td>Clear</td>
<td>Removes all the elements from the <code>List</code>.</td>
</tr>
<tr>
<td>Contains</td>
<td>Returns <code>True</code> if the <code>List</code> contains the specified element; otherwise, returns <code>False</code>.</td>
</tr>
</tbody>
</table>
### 9.3 Introduction to Collections (Cont.)

<table>
<thead>
<tr>
<th>Method or property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Count</td>
<td>Property that returns the number of elements stored in the List.</td>
</tr>
<tr>
<td>IndexOf</td>
<td>Returns the index of the first occurrence of the specified value in the List.</td>
</tr>
<tr>
<td>Insert</td>
<td>Inserts an element at the specified index.</td>
</tr>
<tr>
<td>Remove</td>
<td>Removes the first occurrence of the specified value.</td>
</tr>
<tr>
<td>RemoveAt</td>
<td>Removes the element at the specified index.</td>
</tr>
<tr>
<td>RemoveRange</td>
<td>Removes a specified number of elements starting at a specified index.</td>
</tr>
<tr>
<td>Sort</td>
<td>Sorts the List.</td>
</tr>
<tr>
<td>TrimToSize</td>
<td>Sets the Capacity of the List to the number of elements the List currently contains (Count).</td>
</tr>
</tbody>
</table>

**Fig. 9.5** | Some methods and properties of class `List(Of T)`. (Part 2 of 2.)
Figure 9.6 demonstrates dynamically resizing a **List** object.

```vbnet
Module ListCollection
    Sub Main()
        Dim items As New List(Of String) ' create a new List of Strings
        items.Add("red") ' append an item to the List
        items.Insert(0, "yellow") ' insert the value at index 0
        Console.WriteLine("Display list contents with counter-controlled loop:") ' header
        ' display the colors in the list
        For i = 0 To items.Count - 1
            Console.Write(" {0}", items(i))
        Next
    End Sub
End Module
```

**Fig. 9.6** | Generic **List** collection demonstration. (Part 1 of 3.)

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

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

Displaying the items in the **List**.
' display colors using For Each...Next in the Display method
Display(items, vbNewLine & _
   "Display list contents with For Each...Next statement:")

items.Add("green") ' add "green" to the List
items.Add("yellow") ' add "yellow" to the List
Display(items, "List with two new elements:") ' display the List

items.Remove("yellow") ' remove the first "yellow"
Display(items, "Remove first instance of yellow") ' display List
items.RemoveAt(1) ' remove item at index 1
Display(items, "Remove second list element (green):") ' display List

' check if a value is in the List
Console.WriteLine(vbNewLine & """red"" is {0}in the list", _
   If(items.Contains("red"), String.Empty, "not "))

' display number of elements in the List
Console.WriteLine(vbNewLine & "Count: {0}", Items.Count)
' display the capacity of the List
Console.WriteLine("Capacity: \{0\}", items.Capacity)
End Sub ' Main

' display the List's elements on the console
Sub Display(ByVal items As List(Of String), ByVal header As String)
    Console.Write(header) ' print header

    ' display each element in items
    For Each item In items
        Console.Write(" {0}", item)
    Next

    Console.WriteLine() ' print end of line
End Sub ' Display
End Module ' ListCollection

Display list contents with counter-controlled loop: yellow red
display list contents with For Each...Next 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

`Capacity` property indicates how many items the List can hold without being resized.
9.3 Introduction to Collections (Cont.)

- The `Remove` method removes the first element with a given value.
- `RemoveAt` removes the element at the specified index.
9.3 Introduction to Collections (Cont.)

- **Contains** returns True if the String is already in the List.
- **Count** returns the length of the List.
- The **Capacity** property indicates how many items the List can hold without being resized.
• Fig. 9.7, a **List of Strings** is queried.

• LINQ’s **deferred execution** means that the query is executed only when the results are retrieved, so the same query can be re-used.

```vbnet
Module LinqWithListCollection

Sub Main()
    ' populate a List of Strings
    Dim items As New List(Of 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

    ' select Strings starting with "r" and convert them to uppercase
    Dim startsWithR = _
    From item In items _
    Where item.StartsWith("r") _
    Order By item _
    Select item.ToUpper()
End Sub
```

Selecting **Strings** that start with "r".

An all-uppercase **String** is returned by **ToUpper**.
Fig. 9.7 | LINQ to Objects using a List(Of String). (Part 2 of 2.)