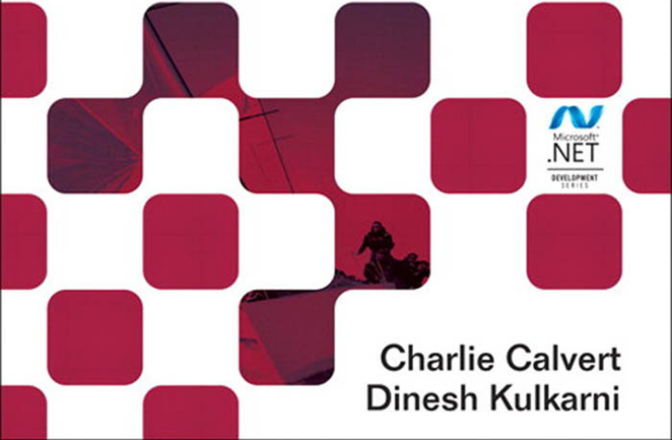
Foreword by Anders Hejlsberg

Essential LINQ



Microsoft .NET Development Series

Charlie Calvert
Dinesh Kulkarni

Praise for *Essential LINQ*

"*Essential LINQ* is the most comprehensive book I have read so far on LINQ technology. Both Charlie and Dinesh have done an excellent job bringing their internal expertise to developers through this book. The book starts with the basics of LINQ and delves deep into the LINQ Ocean. If you would like to learn the internals of LINQ technology and master it, this book is for you."

—Mahesh Chand, MVP, MCP, author and founder of C# Corner

"LINQ is one of the most transformational technologies of .NET and will have a profound impact on how developers architect and code applications going forward. *Essential LINQ* is an excellent book that will help you learn and understand LINQ, and enable you to immediately start applying it with your projects."

—Scott Guthrie, Corporate Vice President, .NET Developer Platform, Microsoft

"Essential LINQ provides an excellent and cohesive overview of LINQ with emphasis on LINQ to Objects, LINQ to SQL, and LINQ to XML."

—Pat Helland, Partner Architect, Microsoft Corporation

"Self-effacing, Charlie Calvert will tell you he is just lucky to hang around smart guys that let him explain things to others, but his straightforward, clear, and precise explanations will make LINQ one of your new best friends."

—Paul Kimmel, Microsoft MVP and author of LINQ Unleashed for C#

"Something wonderful is happening. Developers are discovering the newly integrated ability to query in-memory collections such as arrays and lists, Datasets, XML, and relational databases directly from their .Net code. These collective capabilities, typically referred to as a paradigm shift, have already begun to shake foundations and open developers' eyes to new possibilities. As the initial waves of this shift take place, there exists a parallel need for understanding essential fundamentals and principles. "Essential LINQ" is the best written, most concise source from which to build your fundamental understanding of LINQ. Read this book!"

—Ron Landers, Senior Technical Consultant, Right-Click Consulting, LLC

Listing 5.4 Continued

```
        var query = from p in people
                    join o in query0
                        on p.MusicianId equals o.MusicianId into m
                    from x in m.DefaultIfEmpty()
                    select new { p, x };

        foreach (var items in query)
        {
            Console.WriteLine("{0} {1}", items.p.Name, items.x);
        }
    }
```

In this listing I use composition to link two queries. Nevertheless, I want you to focus on the second query. The first query simply joins the `Order` and `Instrument` classes to create a new anonymous class that includes the name of each instrument:

```
1 1 Tenor Saxophone
2 2 Trumpet
3 3 Soprano Saxophone
4 3 Tenor Saxophone
5 4 Tenor Saxophone
6 2 Keyboard
```

The second query in this series joins the `Musicians` in the `people` collection with the anonymous class returned by the first query in this series. The `into` operator is used just as in a group join. The key of this join is a musician, and the associated data is the anonymous class returned by the first query.

The distinguishing trait of a left outer join in LINQ is the `from` clause that uses `DefaultIfEmpty` as a data source. We have an artist, Bela Fleck, with no associated orders. If we asked for the first order associated with Bela Fleck, we would get a range error on the empty collection of orders associated with this artist. `DefaultIfEmpty` resolves this error by returning the default value for this anonymous reference type, which is null. (Recall that the default value for any reference type is null.) In our `foreach` loop, when it comes time to print the orders associated with Bela Fleck, C# handles our null value smoothly, and prints nothing to the screen.

Most importantly, it returns a flat relational table rather than hierarchical data:

```
Sonny Rollings { OrderId = 1, MusicianId = 1, Name = Tenor Saxophone }
Miles Davis { OrderId = 2, MusicianId = 2, Name = Trumpet }
Miles Davis { OrderId = 6, MusicianId = 2, Name = Keyboard }
John Coltrane { OrderId = 3, MusicianId = 3, Name = Soprano Saxophone }
John Coltrane { OrderId = 4, MusicianId = 3, Name = Tenor Saxophone }
Charlie Parker { OrderId = 5, MusicianId = 4, Name = Tenor Saxophone }
Bela Fleck
```

The point here is not that flat datasets are better or worse than the hierarchical data seen in the group join from the previous section. The point is merely that you should use `DefaultIfEmpty` if you want to return a traditional, SQL-like flat dataset.

If we wanted to, we could use an override of `DefaultIfEmpty` to send back custom data:

```
var query03 = from p in people
              join o in query0
                 on p.MusicianId equals o.MusicianId into m
              from x in m.DefaultIfEmpty(
                 new { OrderId = 0, MusicianId = 0, Name = "Nothing" })
              select new { p, x };
```

This call to `DefaultIfEmpty` is fairly interesting. The method is expecting an instance of our anonymous type. How can we create an instance of a type if we don't know its name? It would seem impossible, but there is a solution. LINQ knows the fields of our anonymous type, and if we create another anonymous object with the same fields, in the same order, the compiler is smart enough to match it up with our anonymous type and create the proper instance!

Here is the output "flat" result set returned by running this computation through a `foreach` loop:

```
Sonny Rollings { OrderId = 1, MusicianId = 1, Name = Tenor Saxophone }
Miles Davis { OrderId = 2, MusicianId = 2, Name = Trumpet }
Miles Davis { OrderId = 6, MusicianId = 2, Name = Keyboard }
John Coltrane { OrderId = 3, MusicianId = 3, Name = Soprano Saxophone }
John Coltrane { OrderId = 4, MusicianId = 3, Name = Tenor Saxophone }
Charlie Parker { OrderId = 5, MusicianId = 4, Name = Tenor Saxophone }
Bela Fleck { OrderId = 0, MusicianId = 0, Name = Nothing }
```

Using the Object Model to "Join" Classes

I've spent quite a bit of time showing you how to write `join` clauses. There is no doubt that `join` clauses play an important role in LINQ, but they do not take center stage as often as they do in SQL. That is because object-oriented developers have a better way of showing the relationship between classes: They simply establish an association. Consider the code shown in Listing 5.5.

LISTING 5.5 Working with the Simple Association Between the Instrument and Musician Classes

```
class Instrument
{
    public int InstrumentId { get; set; }
    public string Name { get; set; }
}

class Musician
{
    public int MusicianId { get; set; }
    public string Name { get; set; }
    public Instrument instrument;
}

public void RunTest03()
{
    List<Musician> people = new List<Musician>
    {
        new Musician { MusicianId = 1, Name = "Charlie Parker",
            instrument = new Instrument {
               InstrumentId = 1, Name = "Saxophone" } },
        new Musician { MusicianId = 1, Name = "Sonny Rollings",
            instrument = new Instrument {
               InstrumentId = 1, Name = "Saxophone"} },
        new Musician { MusicianId = 2, Name = "Miles Davis",
            instrument = new Instrument {
               InstrumentId = 2, Name = "Trumpet" } }
    };

    var query = from p in people
                where p.instrument.InstrumentId == 1
                select new { Musician=p.Name,
                             Intrument=p.instrument.Name };

}
```

In this example, the `Musician` and `Instrument` tables are associated by a field in the `Musician` table called `Instrument`. In a real-world program, this would probably be declared as an array of `Instruments`, but I have simplified matters here to keep the code short and easy to read.

After declaring the types, this code fragment creates a list of musicians and the instruments they use. Given these declarations, we can now move from the `Musician` class to the `Instrument` class using dot notation: `p.instrument.Name`.

It is obviously much easier to use this syntax than it is to create a `join` clause, as we did in the previous listings. As a result, the syntax I've shown here is the preferred way to handle joins in LINQ.

However, many times there is no direct relationship in the object model between classes. The kind of association needed to use this dot notation requires that one object, or a collection of objects, be declared as a field of a second object. If that relationship does not exist, you must use the `join` clauses shown earlier to link two tables.

Associations in LINQ to SQL

Join syntax plays a big role in SQL queries. As a result, you would probably expect that LINQ to SQL would make heavy use of `join` clauses. In practice, however, that is usually not the case. In Chapters 7 through 10, you will see that whenever you have a true relationship between tables based on key, you can use dot notation to perform joins. As a result, join syntax is something that you use infrequently in LINQ.

That is all I'll say about creating joins at this time. However, if you keep reading, you will find sections on the `SelectMany` operator that describe how to use multiple `from` clauses in a single query. Those sections reveal yet another very important way to perform a join between two classes.

Projections

Although some details have been omitted, by this point in the chapter you have had a chance to look at all the major features of query expressions except for the return sequence, or computation, that they produce. This section is designed to give you a close look at that subject. It covers the following topics:

- An overview of projections
- Projections and deferred execution
- Using `Select`, `SelectMany`, and two `from` clauses to project a result from a query

Overview of Projections

A `select` or `group-by` clause usually determines the type returned, or *projected*, by a deferred query expression. In this sense, it plays much the same role in a query expression as the keyword `return` plays in a method. But a query expression is said to project a type, whereas a function is said to return a type. Furthermore, a deferred query expression does not execute until you begin asking for the individual members of the result sequence.

As you've seen in previous chapters, a query expression can use anonymous types to project a new class in a `select` clause. The code shown in Listing 5.6 provides a quick review of this subject.

LISTING 5.6 This Program Uses a List of Customers as a Data Source and Returns an `IEnumerable<T>`, Where T is an Anonymous Class

```
class Customer
{
   public string CustomerID { get; set; }
   public string ContactName { get; set; }
   public string City { get; set; }
}

class Program
{
   private static List<Customer> GetCustomers()
   {
      return new List<Customer>
      {
         new Customer { CustomerID = "ALFKI",
                        ContactName = "Maria Anders", City = "Berlin" },
         new Customer { CustomerID = "ANATR",
                        ContactName = "Ana Trujillo", City = "Mexico D.F." },
         new Customer { CustomerID="ANTON",
                              ContactName="Antonio Moreno", City="Mexico D.F." }
        };
    }

static void Main(string[] args)
{
    var query = from c in GetCustomers()
                where c.City == "Mexico D.F."
                select new { City = c.City, ContactName = c.ContactName };
}
```

The data source for this query is a collection of `Customers`, but it returns a collection of some anonymous class defined in the `select` clause of the query.

Projects in SQL

SQL queries also project a new type. Assume the existence of a table called `Customer` that contains five fields: `Id`, `Name`, `Address`, `Zip`, and `Phone`. If you write `select Name, Address from Customer`, you are projecting a new type that contains two fields called `Name` and `Address`. LINQ does much the same thing, but it uses an anonymous class to encapsulate the data that is returned from the query.

The projection found in a `select` clause can also play a role in determining the transformational properties of a LINQ query. You've already read about transformations, and they will surface frequently in subsequent chapters. So for now I'll simply include Listing 5.7, taken from the Linq-Transform sample that accompanies this book, as a reminder of how a `select` clause can help transform object-oriented data into XML.

Listing 5.7 Using a LINQ Query to Transform Objects into XML

```
class Mountain
{
    public string Name { get; set; }
    public int Height { get; set; }
    public string State { get; set; }
}

static void Main(string[] args)
{
```

continues