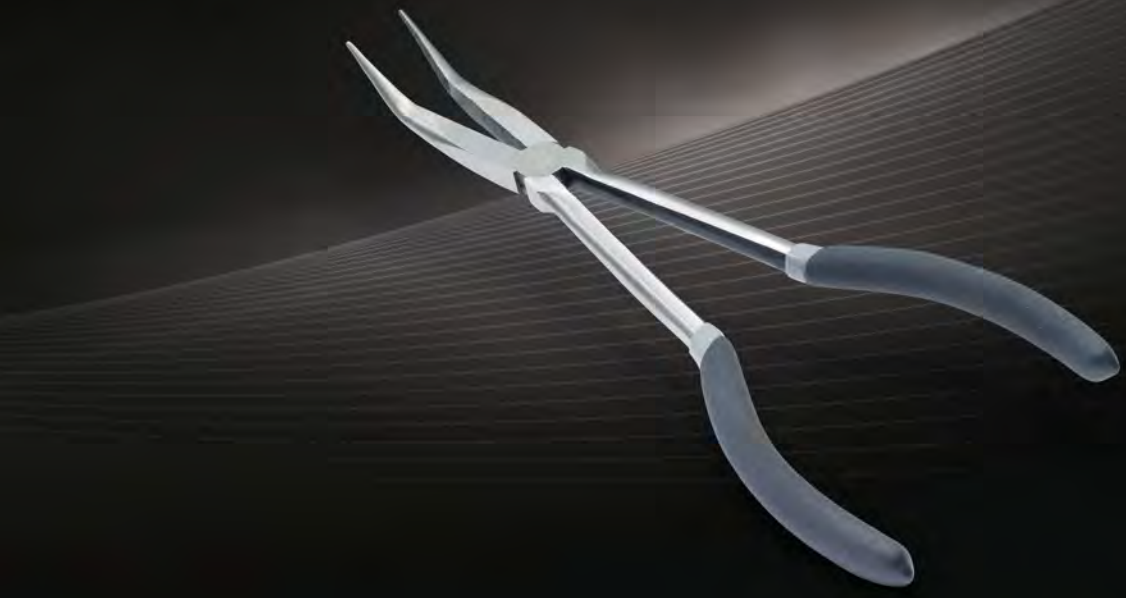
Microsoft

Programming Microsoft® LINQ in Microsoft .NET Framework 4



Paolo Pialorsi
Marco Russo

Microsoft®

Programming Microsoft® LINQ in Microsoft .NET Framework 4

Paolo Pialorsi

Marco Russo

Printed and bound in the United States of America.

1 2 3 4 5 6 7 8 9 M 5 4 3 2 1 0

Microsoft Press titles may be purchased for educational, business or sales promotional use. Online editions are also available for most titles (*http://my.safaribooksonline.com*). For more information, contact our corporate/institutional sales department: (800) 998-9938 or send comments to *mspinput@microsoft.com*.

Acquisitions and Development Editor: Russell Jones
Production Editor: Adam Zaremba
Editorial Production: OTSI, Inc.
Technical Reviewer: Debbie Timmins
Indexing: Ron Strauss
Cover: Karen Montgomery
Compositor: Octal Publishing, Inc.
Illustrator: Robert Romano

978-0-735-64057-3

To Andrea and Paola: thanks for your everyday support!

—Paolo

Contents at a Glance

Table of Contents

Part IV Advanced LINQ

Preface

We saw Language Integrated Query (LINQ) for the first time in September 2005, when the LINQ Project was announced during the Professional Developers Conference (PDC 2005). We immediately realized the importance and the implications of LINQ for the long term. At the same time, we felt it would be a huge error to look to LINQ only for its capability to wrap access to relational data. This would be an error because the important concept introduced by LINQ is the growth in code abstraction that comes from using a consistent pattern that makes code more readable, without having to pay in terms of loss of control. We liked LINQ, we could foresee widespread use for it, but we were worried about the possible misperception of its key points. For these reasons, we started to think about writing a book about LINQ.

Our opportunity to write such a book began when our proposal was accepted by Microsoft Press. We wrote an initial short version of this book, *Introducing Microsoft LINQ* (Microsoft Press), which was based on beta 1 code. A second book, *Programming Microsoft LINQ* (Microsoft Press), comprehensively discussed LINQ in .NET 3.5. Readers provided a lot of feedback about both these books. We took both the positive and more importantly, the negative comments as opportunities to improve the book. Today, we are writing the preface to the third book about LINQ, *Programming Microsoft LINQ in Microsoft .NET Framework 4*, which we believe is a more mature book, full of useful content to help people develop real-world .NET solutions that leverage LINQ and new .NET 4.0 features!

After spending almost five years working with LINQ, this book represents a tremendous goal for us, but it is just the beginning for you. LINQ introduces a more declarative style of programming; it's not a temporary trend. Anders Hejlsberg, the chief designer of C#, said that LINQ tries to solve the impedance mismatch between code and data. We think that LINQ is probably already one step ahead of other methods of resolving that dilemma because it can also be used to write parallel algorithms, such as when using the Parallel LINQ (PLINQ) implementation.

LINQ can be pervasive in software architectures because you can use it in any tier of an application; however, just like any other tool, it can be used effectively or not. We tried to address the most beneficial ways to use LINQ throughout the book. We suspect that at the beginning, you—as we did five years ago—will find it natural to use LINQ in place of relational database queries, but you'll soon find that the ideas begin to pervade your approach to programming. This turning point happens when you begin writing algorithms that operate on in-memory data using LINQ to Objects queries. That should be easy. In fact, after only three chapters of this book, you will already have the knowledge required to do that. But in reality, that is the hardest part, because you need to change the way you think about your code. You need to start thinking in LINQ. We have not found a magic formula to teach this. Probably, like any big change, you will need time and practice to metabolize it.

Enjoy the reading!

Acknowledgments

A book is the result of the work of many people. Unfortunately, only the authors have their names on the cover. This section is only partial compensation for other individuals who helped out.

First, we want to thank Luca Bolognese for his efforts in giving us resources and contacts that helped us to write this book and the two previous editions.

We also want to thank all the people from Microsoft who answered our questions along the way—in particular, Mads Torgersen, Amanda Silver, Erick Thompson, Joe Duffy, Ed Essey, Yuan Yu, Dinesh Kulkarni, and Luke Hoban. Moreover, Charlie Calvert deserves special mention for his great and precious help.

We would like to thank Microsoft Press, O'Reilly, and all the publishing people who contributed to this book project: Ben Ryan, Russell Jones, Jaime Odell, Adam Witwer, and Debbie Timmins. Russell has followed this book from the beginning; he helped us to stay on track, answered all our questions, remained tolerant of our delays, and improved a lot of our drafts. Jaime and Adam have been so accurate and patient in their editing work that we really want to thank them for their great job. Debbie has been the main technical reviewer.

We also want to thank the many people who had the patience to read our drafts and suggest improvements and corrections. Big thanks to Guido Zambarda, Luca Regnicoli, and Roberto Brunetti for their reviews. Guido deserves special thanks for his great job in reviewing all the chapters and the code samples during the upgrade of this book from .NET 3.5 to .NET 4.0.

Finally, we would like to thank Giovanni Librando, who supported us—one more time in our life—when we were in doubt about starting this new adventure. Now the book is here, thanks Giovanni!

Introduction

This book covers Language Integrated Query (LINQ) both deeply and widely. The main goal is to give you a complete understanding of how LINQ works, as well as what to do—and what not to do—with LINQ.

To work with the examples in this book, you need to install both Microsoft .NET Framework 4.0 and Microsoft Visual Studio 2010 on your development machine.

This book has been written against the released-to-market (RTM) edition of LINQ and Microsoft .NET 4.0. The authors have created a website (*http://www.programminglinq.com/*) where they will maintain a change list, a revision history, corrections, and a blog about what is going on with the LINQ project and this book.

Who Is This Book For?

The target audience for this book is .NET developers with a good knowledge of Microsoft .NET 2.0 or 3.x who are wondering whether to upgrade their expertise to Microsoft .NET 4.0.

Organization of This Book

This book is divided into five parts that contain 19 chapters.

The authors use C# as the principal language in their examples, but almost all the LINQ features shown are available in Visual Basic as well. Where appropriate, the authors use Visual Basic because it has some features that are not available in C#.

The first part of this book, "LINQ Foundations," introduces LINQ, explains its syntax, and supplies all the information you need to start using LINQ with in-memory objects (LINQ to Objects). It is important to learn LINQ to Objects before any other LINQ implementation because many of its features are used in the other LINQ implementations described in this book. Therefore, the authors strongly suggest that you read the three chapters in Part I first.

The second part of this book, "LINQ to Relational," is dedicated to all the LINQ implementations that provide access to relational stores of data. In Chapter 4 "Choosing Between LINQ to SQL and LINQ to Entities," you will find some useful tips and suggestions that will help you choose between using LINQ to SQL and LINQ to Entities in your software solutions.

The LINQ to SQL implementation is divided into three chapters. In Chapter 5, "LINQ to SQL: Querying Data," you will learn the basics for mapping relational data to LINQ entities and how to build LINQ queries that will be transformed into SQL queries. In Chapter 6, "LINQ to SQL:

Managing Data," you will learn how to handle changes to data extracted from a database using LINQ to SQL entities. Chapter 7, "LINQ to SQL: Modeling Data and Tools," is a guide to the tools available for helping you define data models for LINQ to SQL. If you are interested in using LINQ to SQL in your applications, you should read all the LINQ to SQL chapters.

The LINQ to Entities implementation is also divided into three chapters. In Chapter 8, "LINQ to Entities: Modeling Data with Entity Framework," you will learn how to create an Entity Data Model and how to leverage the new modeling features of Entity Framework 4.0. Chapter 9, "LINQ to Entities: Querying Data," focuses on querying and retrieving entities using LINQ to Entities, while Chapter 10, "LINQ to Entities: Managing Data," shows how to handle changes to those entities using LINQ to Entities, how to manage data concurrency, and how to share entities across multiple software layers. If you are interested in leveraging LINQ to Entities in your software solutions, you should read all the LINQ to Entities chapters.

Chapter 11, "LINQ to DataSet," covers the implementation of LINQ that targets ADO.NET DataSets. If you have an application that makes use of DataSets, this chapter will teach you how to integrate LINQ, or at least how to progressively migrate from DataSets to the domain models handled with LINQ to SQL or LINQ to Entities.

The third part, "LINQ to XML," includes two chapters about LINQ to XML: Chapter 12, "LINQ to XML: Managing the XML Infoset," and Chapter 13, "LINQ to XML: Querying Nodes." The authors suggest that you read these chapters before you start any development that reads or manipulates data in XML.

The fourth part, "Advanced LINQ," includes the most complex topics of the book. In Chapter 14, "Inside Expression Trees," you will learn how to handle, produce, or simply read an expression tree. Chapter 15, "Extending LINQ," provides information about extending LINQ using custom data structures by wrapping an existing service, and finally by creating a custom LINQ provider. Chapter 16, "Parallelism and Asynchronous Processing," describes a LINQ interface to the Parallel Framework for .NET. Finally, Chapter 17, "Other LINQ Implementations," offers an overview of the most significant LINQ components available from Microsoft and third-party vendors. For the most part, the chapters in this part are independent, although Chapter 15 makes some references to Chapter 14.

The fifth part, "Applied LINQ," describes the use of LINQ in several different scenarios of a distributed application. Chapter 18, "LINQ in a Multitier Solution," is likely to be interesting for everyone because it is an architecturally focused chapter that can help you make the right design decisions for your applications. Chapter 19, "LINQ Data Binding," presents relevant information about the use of LINQ for binding data to user interface controls using existing libraries such as ASP.NET, Windows Presentation Foundation, Silverlight, and Windows Forms. The authors suggest that you read Chapter 18 before delving into the details of specific libraries.

Conventions and Features in This Book

This book presents information using conventions designed to make the information readable and easy to follow:

- Boxed elements with labels such as "Note" provide additional information or alternative methods for completing a step successfully.
- Text that you type (apart from code blocks) appears in bold.
- A plus sign (+) between two key names means that you must press those keys at the same time. For example, "Press Alt+Tab" means that you hold down the Alt key while you press the Tab key.
- A vertical bar between two or more menu items (e.g., File | Close), means that you should select the first menu or menu item, then the next, and so on.

System Requirements

Here are the system requirements you will need to work with LINQ and to work with and execute the sample code that accompanies this book:

- Supported operating systems: Microsoft Windows Server 2003, Windows Server 2008, Windows Server 2008 R2, Windows XP with Service Pack 2, Windows Vista, Windows 7
- Microsoft Visual Studio 2010

The Companion Website

This book features a companion website where you can download all the code used in the book. The code is organized by topic; you can download it from the companion site here: *http://aka.ms/640573/files*.

Find Additional Content Online

As new or updated material becomes available that complements this book, it will be posted online on the Microsoft Press Online Developer Tools website. The type of material you might find includes updates to book content, articles, links to companion content, errata, sample chapters, and more. This website will be available soon at *www.microsoft.com/learning/books/online/developer*, and will be updated periodically.

Errata & Book Support

We've made every effort to ensure the accuracy of this book and its companion content. If you do find an error, please report it on our Microsoft Press site:

1. Go to *www.microsoftpressstore.com*.
2. In the Search box, enter the book's ISBN or title.
3. Select your book from the search results.
4. On your book's catalog page, find the Errata & Updates tab
5. Click View/Submit Errata.

You'll find additional information and services for your book on its catalog page. If you need additional support, please e-mail Microsoft Press Book Support at *mspinput@microsoft.com*.

Please note that product support for Microsoft software is not offered through the addresses above.

We Want to Hear from You

At Microsoft Press, your satisfaction is our top priority, and your feedback our most valuable asset. Please tell us what you think of this book at:

http://www.microsoft.com/learning/booksurvey

The survey is short, and we read every one of your comments and ideas. Thanks in advance for your input!

Stay in Touch

Let's keep the conversation going! We're on Twitter: *http://twitter.com/MicrosoftPress*

Part I

LINQ Foundations

In this part:

Chapter 1

LINQ Introduction

By surfing the web, you can find several descriptions of Microsoft Language Integrated Query (LINQ), including these:

- LINQ provides a uniform programming model for any kind of data. With it, you can query and manipulate data by using a consistent model that is independent of data sources.
- LINQ is another tool for embedding SQL queries into code.
- LINQ is another data abstraction layer.

All these descriptions are correct to a degree, but each focuses on only a single aspect of LINQ. LINQ is much easier to use than a "uniform programming mode"; it can do much more than embed SQL queries; and it is far from being just another data abstraction layer.

What Is LINQ?

LINQ is a programming model that introduces queries as a first-class concept into any Microsoft .NET Framework language. Complete support for LINQ, however, requires some extensions to whatever .NET Framework language you are using. These language extensions boost developer productivity, thereby providing a shorter, more meaningful, and expressive syntax with which to manipulate data.

More Info Details about language extensions can be found on the Microsoft Developer Network (MSDN), located at *msdn.microsoft.com*.

LINQ provides a methodology that simplifies and unifies the implementation of any kind of data access. LINQ does not force you to use a specific architecture; it facilitates the implementation of several existing architectures for accessing data, such as:

- RAD/prototype
- Client/server
- N-tier
- Smart client

LINQ made its first appearance in September 2005 as a technical preview. Since then, it has evolved from an extension of Microsoft Visual Studio 2005 to an integrated part of .NET Framework 3.5 and Visual Studio 2008, both released in November 2007. The first released version of LINQ directly supported several data sources. Now with .NET Framework 4 and Visual Studio 2010, LINQ also includes LINQ to Entities, which is part of the Microsoft ADO.NET Entity Framework, and Parallel LINQ (PLINQ). This book describes current LINQ implementations from Microsoft for accessing several different data sources, such as the following:

- LINQ to Objects
- LINQ to ADO.NET
- LINQ to Entities
- LINQ to SQL
- LINQ to DataSet
- LINQ to XML

Extending LINQ

In addition to the built-in data source types, you can extend LINQ to support additional data sources. Possible extensions might be LINQ to Exchange or LINQ to LDAP, to name just a couple of examples. Some implementations are already available using LINQ to Objects. We describe a possible LINQ to Reflection query in the "LINQ to Objects" section of this chapter. Chapter 15, "Extending LINQ," discusses more advanced extensions of LINQ, and Chapter 17, "Other LINQ Implementations," covers some of the existing LINQ implementations.

LINQ is likely to have an impact on the way applications are coded, but it would be incorrect to think that LINQ will change application architectures; its goal is to provide a set of tools that improve code implementation by adapting to several *different* architectures. However, we expect that LINQ will affect some critical parts of the layers of an n-tier solution. For example, we envision the use of LINQ in a SQLCLR stored procedure, with a direct transfer of the query expression to the SQL engine instead of using a SQL statement.

Many possible evolutionary tracks could originate from LINQ, but we should not forget that SQL is a widely adopted standard that cannot be easily replaced by another, just for performance reasons. Nevertheless, LINQ is an interesting step in the evolution of current mainstream programming languages. The declarative nature of its syntax might be interesting for uses other than data access, such as the parallel programming that is offered by

PLINQ. Many other services can be offered by an execution framework to a program written using a higher level of abstraction, such as the one offered by LINQ. A good understanding of this technology is important because LINQ has become a "standard" way to describe data manipulation operations inside a program written in the .NET Framework.

More Info PLINQ is covered in Chapter 16, "Parallelism and Asynchronous Processing."

Why Do We Need LINQ?

Today, data managed by a program can originate from various data sources: an array, an object graph, an XML document, a database, a text file, a registry key, an email message, Simple Object Access Protocol (SOAP) message content, a Microsoft Excel file…. The list is long.

Each data source has its own specific data access model. When you have to query a database, you typically use SQL. You navigate XML data by using the Document Object Model (DOM) or XPath/XQuery. You iterate an array and build algorithms to navigate an object graph. You use specific application programming interfaces (APIs) to access other data sources, such as an Excel file, an email message, or the Windows registry. In the end, you use different programming models to access different data sources.

The unification of data access techniques into a single comprehensive model has been attempted in many ways. For example, by using Open Database Connectivity (ODBC) providers, you can query an Excel file as you would a Windows Management Instrumentation (WMI) repository. With ODBC, you use a SQL-like language to access data represented through a relational model.

Sometimes, however, data is represented more effectively in a hierarchical or network model instead of a relational one. Moreover, if a data model is not tied to a specific language, you probably need to manage several type systems. All these differences create an "impedance mismatch" between data and code.

LINQ addresses these issues by offering a uniform way to access and manage data without forcing the adoption of a "one size fits all" model. LINQ makes use of common capabilities in the *operations* in different data models instead of flattening the different *structures* between them. In other words, by using LINQ, you keep existing heterogeneous data structures, such as classes or tables, but you get a uniform syntax to query all these data types—regardless of their physical representation. Think about the differences between a graph of in-memory objects and relational tables with proper relationships. With LINQ, you can use the same query syntax over both models.

Here is a simple LINQ query for a typical software solution that returns the names of customers in Italy:

```
var query =
    from   c in Customers
    where  c.Country == "Italy"
    select c.CompanyName;
```

The result of this query is a list of strings. You can enumerate these values with a *foreach* loop in Microsoft Visual C#:

```
foreach ( string name in query ) {
    Console.WriteLine( name );
}
```

Both the *query* definition and the *foreach* loop are regular C# 3.0 statements, but what is *Customers*? At this point, you might be wondering what it is we are querying. Is this query a new form of Embedded SQL? Not at all. You can apply the same query (and the *foreach* loop) to a SQL database, to a *DataSet* object, to an array of objects in memory, to a remote service, or to many other kinds of data.

For example, *Customers* could be a collection of objects:

```
Customer[] Customers;
```

Customer data could reside in a DataTable in a DataSet:

```
DataSet ds = GetDataSet();
DataTable Customers = ds.Tables["Customers"];
```

Customers could be an entity class that describes a physical table in a relational database:

```
DataContext db = new DataContext( ConnectionString );
Table<Customer> Customers = db.GetTable<Customer>();
```

Or *Customers* could be an entity class that describes a conceptual model and is mapped to a relational database:

```
NorthwindModel dataModel = new NorthwindModel();
ObjectSet<Customer> Customers = dataModel.Customers;
```

How LINQ Works

As you will learn in Chapter 2, "LINQ Syntax Fundamentals," the SQL-like syntax used in LINQ is called a *query expression*. A SQL-like query mixed with the syntax of a program written in a language that is not SQL is typically called *Embedded SQL*, but languages that implement it do so using a simplified syntax. In Embedded SQL, these statements are not integrated into the language's native syntax and type system because they have a different syntax and several

restrictions related to their interaction. Moreover, Embedded SQL is limited to querying databases, whereas LINQ is not. LINQ provides much more than Embedded SQL does; it provides a query syntax that is integrated into a language. But how does LINQ work?

Let's say you write the following code using LINQ:

```
Customer[] Customers = GetCustomers();
var query =
    from   c in Customers
    where  c.Country == "Italy"
    select c;
```

The compiler generates this code:

```
Customer[] Customers = GetCustomers();
IEnumerable<Customer> query =
        Customers
        .Where( c => c.Country == "Italy" );
```

The following query is a more complex example (without the *Customers* declaration, for the sake of brevity):

```
var query =
    from    c in Customers
    where   c.Country == "Italy"
    orderby c.Name
    select  new { c.Name, c.City };
```

As you can see, the generated code is more complex too:

```
var query =
        Customers
        .Where( c => c.Country == "Italy" );
        .OrderBy( c => c.Name )
        .Select( c => new { c.Name, c.City } );
```

As you can see, the generated code apparently calls instance members on the object returned from the previous call: *Where* is called on *Customers*, *OrderBy* is called on the object returned by *Where*, and finally *Select* is called on the object returned by *OrderBy*. You will see that this behavior is regulated by what are known as *extension methods* in the host language (C# in this case). The implementation of the *Where*, *OrderBy*, and *Select* methods—called by the sample query—depends on the type of *Customers* and on namespaces specified in relevant *using* statements. Extension methods are a fundamental syntax feature that is used by LINQ to operate with different data sources by using the same syntax.

More Info An extension method appears to extend a class (the *Customers* class in our examples), but in reality a method of an external type receives the instance of the class that seems to be extended as the first argument. The *var* keyword used to declare *query* infers the variable type declaration from the initial assignment, which in this case will return an *IEnumerable<T>* type.

Another important concept is the timing of operations over data. In general, a LINQ query is not executed until the result of the query is required. Each query describes a set of operations that will be performed only when the result is actually accessed by the program. In the following example, this access is performed only when the *foreach* loop executes:

```
var query = from c in Customers ...
foreach ( string name in query ) ...
```

There are also methods that iterate a LINQ query result, producing a persistent copy of data in memory. For example, the *ToList* method produces a typed *List*<*T*> collection:

```
var query = from c in Customers ...
List<Customer> customers = query.ToList();
```

When the LINQ query operates on data that is in a relational database (such as a Microsoft SQL Server database), it generates an equivalent SQL statement instead of operating with in-memory copies of data tables. The query's execution on the database is delayed until the query results are first accessed. Therefore, if in the last two examples *Customers* was a *Table*<*Customer*> type (a physical table in a relational database) or an *ObjectSet*<*Customer*> type (a conceptual entity mapped to a relational database), the equivalent SQL query would not be sent to the database until the *foreach* loop was executed or the *ToList* method was called. The LINQ query can be manipulated and composed in different ways until those events occur.

More Info A LINQ query can be represented as an expression tree. Chapter 14, "Inside Expression Trees," describes how to visit and dynamically build an expression tree, and thereby build a LINQ query.

Relational Model vs. Hierarchical/Network Model

At first, LINQ might appear to be just another SQL dialect. This similarity has its roots in the way a LINQ query can describe a relationship between entities, as shown in the following code:

```
var query =
    from   c in Customers
    join   o in Orders
           on c.CustomerID equals o.CustomerID
    select new { c.CustomerID, c.CompanyName, o.OrderID };
```

This syntax is similar to the regular way of querying data in a relational model by using a SQL *join* clause. However, LINQ is not limited to a single data representation model such as the relational one, where relationships between entities are expressed inside a query but not in the data model. (Foreign keys keep referential integrity but do not participate in a query.) In a hierarchical or network model, parent/child relationships are part of the data structure. For example, suppose that each customer has its own set of orders, and each order has its own list of products. In LINQ, you can get the list of products ordered by each customer in this way:

```
var query =
    from   c in Customers
    from   o in c.Orders
    select new { c.Name, o.Quantity, o.Product.ProductName };
```

This query contains no joins. The relationship between *Customers* and *Orders* is expressed by the second *from* clause, which uses *c.Orders* to say "get all *Orders* for the *c Customer*." The relationship between *Orders* and *Products* is expressed by the *Product* member of the *Order* instance. The result projects the product name for each order row by using *o.Product*. *ProductName*.

Hierarchical and network relationships are expressed in type definitions through references to other objects. (Throughout, we will use the phrase "graph of objects" to generically refer to hierarchical or network models.) To support the previous query, we would have classes similar to those in Listing 1-1.

LISTING 1-1 Type declarations with simple relationships

```
public class Customer {
    public string Name;
    public string City;
    public Order[] Orders;
}
public struct Order {
    public int Quantity;
    public Product Product;
}
public class Product {
    public int IdProduct;
    public decimal Price;
    public string ProductName;
}
```

However, chances are that we want to use the same *Product* instance for many different *Orders* of the same product. We probably also want to filter *Orders* or *Products* without accessing them through *Customer*. A common scenario is the one shown in Listing 1-2.

LISTING 1-2 Type declarations with two-way relationships

```
public class Customer {
    public string Name;
    public string City;
    public Order[] Orders;
}
public struct Order {
    public int Quantity;
    public Product Product;
    public Customer Customer;
}
public class Product {
    public int IdProduct;
    public decimal Price;
    public string ProductName;
    public Order[] Orders;
}
```

Let's say we have an array of all products declared as follows:

```
Product[] products;
```

We can query the graph of objects, asking for the list of orders for the single product with an ID equal to 3:

```
var query =
    from   p in products
    where  p.IdProduct == 3
    from   o in p.Orders
    select o;
```

With the same query language, we are querying different data models. When you do not have a relationship defined between the entities used in a LINQ query, you can always rely on subqueries and joins that are available in LINQ syntax just as you can in a SQL language. However, when your data model already defines entity relationships, you can use them, avoiding replication of (and possible mistakes in) the same information.

If you have entity relationships in your data model, you can still use explicit relationships in a LINQ query—for example, when you want to force some condition, or when you simply want to relate entities that do not have native relationships. For example, imagine that you want to find customers and suppliers who live in the same city. Your data model might not provide an explicit relationship between these attributes, but with LINQ you can write the following:

```
var query =
    from   c in Customers
    join   s in Suppliers
           on c.City equals s.City
    select new { c.City, c.Name, SupplierName = s.Name };
```

Data like the following will be returned:

```
City=Torino     Name=Marco      SupplierName=Trucker
City=Dallas     Name=James      SupplierName=FastDelivery
City=Dallas     Name=James      SupplierName=Horizon
City=Seattle    Name=Frank      SupplierName=WayFaster
```

If you have experience using SQL queries, you probably assume that a query result is always a "rectangular" table, one that repeats the data of some columns many times in a join like the previous one. However, often a query contains several entities with one or more one-to-many relationships. With LINQ, you can write queries like the following one to return a graph of objects:

```
var query =
    from   c in Customers
    join   s in Suppliers
           on c.City equals s.City
           into customerSuppliers
    select new { c.City, c.Name, customerSuppliers };
```

This query returns a row for each customer, each containing a list of suppliers available in the same city as the customer. This result can be queried again, just as any other object graph with LINQ. Here is how the *hierarchized* results might appear:

```
City=Torino     Name=Marco      customerSuppliers=...
  customerSuppliers: Name=Trucker         City=Torino
City=Dallas     Name=James      customerSuppliers=...
  customerSuppliers: Name=FastDelivery    City=Dallas
  customerSuppliers: Name=Horizon         City=Dallas
City=Seattle    Name=Frank      customerSuppliers=...
  customerSuppliers: Name=WayFaster       City=Seattle
```

If you want to get a list of customers and provide each customer with the list of products he ordered at least one time and the list of suppliers in the same city, you can write a query like this:

```
var query =
    from   c in Customers
    select new {
        c.City,
        c.Name,
        Products = (from   o in c.Orders
                    select new { o.Product.IdProduct,
                                 o.Product.Price }).Distinct(),
        CustomerSuppliers = from   s in Suppliers
                            where  s.City == c.City
                            select s };
```

You can take a look at the results for a couple of customers to understand how data is returned from the previous single LINQ query:

```
City=Torino     Name=Marco      Products=...    CustomerSuppliers=...
  Products: IdProduct=1   Price=10
  Products: IdProduct=3   Price=30
  CustomerSuppliers: Name=Trucker         City=Torino
City=Dallas     Name=James      Products=...    CustomerSuppliers=...
  Products: IdProduct=3   Price=30
  CustomerSuppliers: Name=FastDelivery    City=Dallas
  CustomerSuppliers: Name=Horizon         City=Dallas
```

This type of result would be hard to obtain with one or more SQL queries because it would require an analysis of query results to build the desired graph of objects. LINQ offers an easy way to move data from one model to another and different ways to get the same results.

LINQ requires you to describe your data in terms of entities that are also types in the language. When you build a LINQ query, it is always a set of operations on instances of some classes. These objects might be the real containers of data, or they might be simple descriptions (in terms of metadata) of the external entity you are going to manipulate. A query can be sent to a database through a SQL command only if it is applied to a set of types that maps tables and relationships contained in the database. After you have defined entity classes, you can use *both* approaches we described (joins and entity relationships navigation). The conversion of all these operations into SQL commands is the responsibility of the LINQ engine.

Note When using LINQ to SQL, you can create entity classes by using code-generation tools such as SQLMetal or the Object Relational Designer in Visual Studio. These tools are described in Chapter 7, "LINQ to SQL: Modeling Data and Tools."

Listing 1-3 shows an excerpt of a *Product* class that maps a relational table named *Products*, with five columns that correspond to public properties, using LINQ to SQL.

LISTING 1-3 Class declaration mapped on a database table with LINQ to SQL

```
[Table("Products")]
public class Product {
    [Column(IsPrimaryKey=true)] public int IdProduct;
    [Column(Name="UnitPrice")] public decimal Price;
    [Column()] public string ProductName;
    [Column()] public bool Taxable;
    [Column()] public decimal Tax;
}
```

When you work on entities that describe external data (such as database tables), you can create instances of these kinds of classes and manipulate in-memory objects just as if the data

from all tables were loaded in memory. You submit these changes to the database through SQL commands when you call the *SubmitChanges* method, as shown in Listing 1-4.

LISTING 1-4 Database update calling the *SubmitChanges* method of LINQ to SQL

```
var taxableProducts =
    from   p in db.Products
    where  p.Taxable == true
    select p;
foreach( Product product in taxableProducts ) {
    RecalculateTaxes( product );
}
db.SubmitChanges();
```

The *Product* class in the preceding example represents a row in the Products table of an external database. When you call *SubmitChanges*, all changed objects generate a SQL command to synchronize the corresponding data tables in the database—in this case, updating the corresponding rows in the Products table.

More Info You can find more detailed information about class entities that match tables and relationships in Chapter 5, "LINQ to SQL: Querying Data," in Chapter 6, "LINQ to SQL: Managing Data," and in Chapter 9, "LINQ to Entities: Querying Data."

Listing 1-5 shows the same *Product* entity, generated using LINQ to Entities and the Entity Framework that ships with .NET Framework 4 and Visual Studio 2010.

LISTING 1-5 The *Product* entity class declaration using the Entity Framework

```
[EdmEntityType(Name = "Product")]
public class Product {
    [EdmScalarProperty(EntityKeyProperty = true)] public int IdProduct { get; set; }
    [EdmScalarProperty()] public decimal Price { get; set; }
    [EdmScalarProperty()] public string ProductName { get; set; }
    [EdmScalarProperty()] public bool Taxable { get; set; }
    [EdmScalarProperty()] public decimal Tax { get; set; }
}
```

In Chapter 4, "Choosing Between LINQ to SQL and LINQ to Entities," we will compare the main features of LINQ to SQL and LINQ to Entities. However, you can already see that there are different attributes applied to the code, even if the basic idea is almost the same.

Listing 1-6 shows the same data manipulation you have already seen in LINQ to SQL, but this time applied to the *Product* entity generated using the Entity Framework.

LISTING 1-6 Database update calling the *SaveChanges* method of the Entity Framework

```
var taxableProducts =
    from p in db.Products
    where p.Taxable == true
    select p;

foreach (Product product in taxableProducts) {
    RecalculateTaxes(product);
}
db.SaveChanges();
```

Once again, the main concepts are the same, even though the method invoked (*SaveChanges*), which synchronizes the database tables with the in-memory data, is different.

XML Manipulation

LINQ has a different set of classes and extensions to support manipulating XML data. Imagine that your customers are able to send orders using XML files such as the ORDERS.XML file shown in Listing 1-7.

LISTING 1-7 A fragment of an XML file of orders

```
<?xml version="1.0" encoding="utf-8" ?>
<orders xmlns="http://schemas.devleap.com/Orders">
    <order idCustomer="ALFKI" idProduct="1" quantity="10" price="20.59"/>
    <order idCustomer="ANATR" idProduct="5" quantity="20" price="12.99"/>
    <order idCustomer="KOENE" idProduct="7" quantity="15" price="35.50"/>
</orders>
```

Using standard .NET Framework 2.0 *System.Xml* classes, you can load the file by using a DOM approach or you can parse its contents by using an implementation of *XmlReader*, as shown in Listing 1-8.

LISTING 1-8 Reading the XML file of orders by using an *XmlReader*

```
String nsUri = "http://schemas.devleap.com/Orders";
XmlReader xmlOrders = XmlReader.Create( "Orders.xml" );

List<Order> orders = new List<Order>();
Order order = null;
while (xmlOrders.Read()) {
    switch (xmlOrders.NodeType) {
        case XmlNodeType.Element:
            if ((xmlOrders.Name == "order") &&
            (xmlOrders.NamespaceURI == nsUri)) {
                order = new Order();
                order.CustomerID = xmlOrders.GetAttribute( "idCustomer" );
```

```
                order.Product = new Product();
                order.Product.IdProduct =
                    Int32.Parse( xmlOrders.GetAttribute( "idProduct" ) );
                order.Product.Price =
                    Decimal.Parse( xmlOrders.GetAttribute( "price" ) );
                order.Quantity =
                    Int32.Parse( xmlOrders.GetAttribute( "quantity" ) );
                orders.Add( order );
            }
            break;
    }
}
```

You can also use an XQuery to select nodes:

```
for $order in document("Orders.xml")/orders/order
return $order
```

However, using XQuery requires learning yet another language and syntax. Moreover, the result of the previous XQuery example would need to be converted into a set of *Order* instances to be used within the code.

Regardless of the solution you choose, you must always consider nodes, node types, XML namespaces, and whatever else is related to the XML world. Many developers do not like working with XML because it requires knowledge of another domain of data structures and uses its own syntax. For them, it is not very intuitive. As we have already said, LINQ provides a query engine suitable for any kind of source, even an XML document. By using LINQ queries, you can achieve the same result with less effort and with unified programming language syntax. Listing 1-9 shows a LINQ to XML query made over the orders file.

LISTING 1-9 Reading the XML file by using LINQ to XML

```
XDocument xmlOrders = XDocument.Load( "Orders.xml" );

XNamespace ns = "http://schemas.devleap.com/Orders";
var orders = from o in xmlOrders.Root.Elements( ns + "order" )
             select new Order {
                        CustomerID = (String)o.Attribute( "idCustomer" ),
                        Product = new Product {
                            IdProduct = (Int32)o.Attribute("idProduct"),
                            Price = (Decimal)o.Attribute("price") },
                        Quantity = (Int32)o.Attribute("quantity")
                    };
```

Using LINQ to XML in Microsoft Visual Basic syntax (available since Visual Basic 2008) is even easier; you can reference XML nodes in your code by using an XPath-like syntax, as shown in Listing 1-10.

LISTING 1-10 Reading the XML file by using LINQ to XML and Visual Basic syntax

```
Imports <xmlns:o="http://schemas.devleap.com/Orders">
' ...

Dim xmlOrders As XDocument = XDocument.Load("Orders.xml")
Dim orders =
    From o In xmlOrders.<o:orders>.<o:order>
    Select New Order With {
        .CustomerID = o.@idCustomer,
        .Product = New Product With {
            .IdProduct = o.@idProduct,
            .Price = o.@price},
        .Quantity = o.@quantity}
```

The result of these LINQ to XML queries could be used to transparently load a list of *Order* entities into a customer *Orders* property, using LINQ to SQL to submit the changes into the physical database layer:

```
customer.Orders.AddRange(
    From o In xmlOrders.<o:orders>.<o:order>
    Where o.@idCustomer = customer.CustomerID
    Select New Order With {
        .CustomerID = o.@idCustomer,
        .Product = New Product With {
            .IdProduct = o.@idProduct,
            .Price = o.@price},
        .Quantity = o.@quantity})
```

And if you need to generate an ORDERS.XML file starting from your customer's orders, you can at least use Visual Basic XML literals to define the output's XML structure. Listing 1-11 shows an example.

LISTING 1-11 Creating the XML for orders using Visual Basic XML literals

```
Dim xmlOrders = <o:orders>
    <%= From o In orders
        Select <o:order idCustomer=<%= o.CustomerID %>
                    idProduct=<%= o.Product.IdProduct %>
                    quantity=<%= o.Quantity %>
                    price=<%= o.Product.Price %>/> %>
    </o:orders>
```

Note This syntax is an exclusive feature of Visual Basic. There is no equivalent syntax in C#.

You can appreciate the power of this solution, which keeps the XML syntax without losing the stability of typed code and transforms a set of entities selected via LINQ to SQL into an XML *Infoset*.

More Info You will find more information about LINQ to XML syntax and its potential in Chapter 12, "LINQ to XML: Managing the XML Infoset" and in Chapter 13, "LINQ to XML: Querying Nodes."

Language Integration

Language integration is a fundamental aspect of LINQ. The most visible part is the query expression feature, which has been present since C# 3.0 and Visual Basic 2008. With it, you can write code such as you've seen earlier. For example, you can write the following code:

```
var query =
    from    c in Customers
    where   c.Country == "Italy"
    orderby c.Name
    select  new { c.Name, c.City };
```

The previous example is a simplified version of this code:

```
var query =
        Customers
        .Where( c => c.Country == "Italy" );
        .OrderBy( c => c.Name )
        .Select( c => new { c.Name, c.City } );
```

Many people call this simplification *syntax sugaring* because it is just a simpler way to write code that defines a query over data. However, there is more to it than that. Many language constructs and syntaxes are necessary to support what seems to be just a few lines of code that query data. Under the cover of this simple query expression are local type inference, extension methods, lambda expressions, object initialization expressions, and anonymous types. All these features are useful by themselves, but if you look at the overall picture, you can see important steps in two directions: one moving to a more declarative style of coding, and one lowering the impedance mismatch between data and code.

Declarative Programming

What are the differences between a SQL query and an equivalent C# 2.0 or Visual Basic 2005 program that filters data contained in native storage (such as a table for SQL or an array for C# or Visual Basic)?

In SQL, you can write the following:

```
SELECT * FROM Customers WHERE Country = 'Italy'
```

In C#, you would probably write this:

```
public List<Customer> ItalianCustomers( Customer customers[] )
{
    List<Customer> result = new List<Customer>();
    foreach( Customer c in customers ) {
        if (c.Country == "Italy") result.Add( c );
    }
    return result;
}
```

Note This specific example could have been written in C# 2.0 using a *Find* predicate, but we are using it just as an example of the different programming patterns.

The C# code takes longer to write and read. But the most important consideration is expressivity. In SQL, you describe *what* you want. In C#, you describe *how* to obtain the expected result. In SQL, selecting the best algorithm to implement to get the result (which is more explicitly dealt with in C#) is the responsibility of the query engine. The SQL query engine has more freedom to apply optimizations than a C# compiler, which has many more constraints on how operations are performed.

LINQ enables a more declarative style of coding for C# and Visual Basic. A LINQ query describes operations on data through a declarative construct instead of an iterative one. With LINQ, programmers' intentions can be made more explicit—and this knowledge of programmer intent is fundamental to obtaining a higher level of services from the underlying framework. For example, consider parallelization. A SQL query can be split into several concurrent operations simply because it does not place any constraint on the kind of table scan algorithm applied. A C# *foreach* loop is harder to split into several loops over different parts of an array that could be executed in parallel by different processors.

More Info You will find more information about using LINQ to achieve parallelism in code execution in Chapter 16.

Declarative programming can take advantage of services offered by compilers and frameworks, and in general, it is easier to read and maintain. This single feature of LINQ might be the most important because it boosts programmers' productivity. For example, suppose that you want to get a list of all static methods available in the current application domain that return an *IEnumerable<T>* interface. You can use LINQ to write a query over Reflection:

```
var query =
    from    assembly in AppDomain.CurrentDomain.GetAssemblies()
    from    type in assembly.GetTypes()
    from    method in type.GetMethods()
    where   method.IsStatic
            && method.ReturnType.GetInterface( "IEnumerable'1" ) != null
    orderby method.DeclaringType.Name, method.Name
    group   method by new { Class = method.DeclaringType.Name,
                            Method = method.Name };
```

The equivalent C# code that handles data takes more time to write, is harder to read, and is probably more error prone. You can see a version that is not particularly optimized in Listing 1-12.

LISTING 1-12 C# code equivalent to a LINQ query over Reflection

```
List<String> results = new List<string>();
foreach( var assembly in AppDomain.CurrentDomain.GetAssemblies()) {
    foreach( var type in assembly.GetTypes() ) {
        foreach( var method in type.GetMethods()) {
            if (method.IsStatic &&
                method.ReturnType.GetInterface("IEnumerable'1") != null) {
                string fullName = String.Format( "{0}.{1}",
                                        method.DeclaringType.Name,
                                        method.Name );
                if (results.IndexOf( fullName ) < 0) {
                    results.Add( fullName );
                }
            }
        }
    }
}
results.Sort();
```

Type Checking

Another important aspect of language integration is type checking. Whenever data is manipulated by LINQ, no unsafe cast is necessary. The short syntax of a query expression makes no compromises with type checking: data is always strongly typed, including both the queried collections and the single entities that are read and returned.

The type checking of the languages that support LINQ (starting from C# 3.0 and Visual Basic 2008) is preserved even when LINQ-specific features are used. This enables the use of Visual Studio features such as IntelliSense and Refactoring, even with LINQ queries. These Visual Studio features are other important factors in programmers' productivity.

Transparency Across Different Type Systems

If you think about the type system of the .NET Framework and the type system of SQL Server, you will realize they are different. Using LINQ gives precedence to the .NET Framework type system, because it is the one supported by any language that hosts a LINQ query. However, most of your data will be saved in a relational database, so it is necessary to convert many types of data between these two worlds. LINQ handles this conversion for you automatically, making the differences in type systems almost completely transparent to the programmer.

More Info There are some limitations in the capability to perform conversions between different type systems and LINQ. You will find some information about this topic throughout the book, and you can find a more detailed type system compatibilities table in the product documentation.

LINQ Implementations

LINQ is a technology that covers many data sources. Some of these sources are included in LINQ implementations that Microsoft has provided—starting with .NET Framework 3.5—as shown in Figure 1-1, which also includes LINQ to Entities.

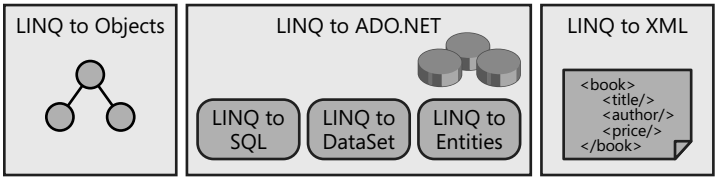


FIGURE 1-1 LINQ implementations provided by Microsoft starting with .NET Framework 3.5.

Each implementation is defined through a set of extension methods that implement the operators needed by LINQ to work with a particular data source. Access to these features is controlled by the imported namespaces.

LINQ to Objects

LINQ to Objects is designed to manipulate collections of objects, which can be related to each other to form a graph. From a certain point of view, LINQ to Objects is the default implementation used by a LINQ query. You enable LINQ to Objects by including the *System.Linq* namespace.

More Info The base concepts of LINQ are explained in Chapter 2, using LINQ to Objects as a reference implementation.

However, it would be a mistake to think that LINQ to Objects queries are limited to collections of user-generated data. You can see why this is not true by analyzing Listing 1-13, which shows a LINQ query that extracts information from the file system. The code reads the list of all files in a given directory into memory and then filters that list with the LINQ query.

LISTING 1-13 LINQ query that retrieves a list of temporary files larger than 10,000 bytes, ordered by size

```
string tempPath = Path.GetTempPath();
DirectoryInfo dirInfo = new DirectoryInfo( tempPath );
var query =
    from    f in dirInfo.GetFiles()
    where   f.Length > 10000
    orderby f.Length descending
    select  f;
```

LINQ to ADO.NET

LINQ to ADO.NET includes different LINQ implementations that share the need to manipulate relational data. It also includes other technologies that are specific to each particular persistence layer:

- **LINQ to SQL** Handles the mapping between custom types in the .NET Framework and the physical table schema in SQL Server.
- **LINQ to Entities** An Object Relational Mapping (ORM) that—instead of using the physical database as a persistence layer—uses a conceptual Entity Data Model (EDM). The result is an abstraction layer that is independent from the physical data layer.
- **LINQ to DataSet** Enables querying a DataSet by using LINQ.

LINQ to SQL and LINQ to Entities have similarities because they both access information stored in a relational database and operate on object entities that represent external data in memory. The main difference is that they operate at a different level of abstraction. Whereas LINQ to SQL is tied to the physical database structure, LINQ to Entities operates over a conceptual model (business entities) that might be far from the physical structure (database tables).

The reason for these different options for accessing relational data through LINQ is that different models for database access are in use today. Some organizations implement all access through stored procedures, including any kind of database query, without using dynamic queries. Many others use stored procedures to insert, update, or delete data and dynamically build SELECT statements to query data. Some see the database as a simple object persistence layer, whereas others put some business logic into the database by using triggers, stored procedures, or both. LINQ tries to offer help and improvement in database access without forcing everyone to adopt a single comprehensive model.

More Info The use of any LINQ to ADO.NET implementation depends on the inclusion of particular namespaces in the scope. Part II, "LINQ to Relational," investigates LINQ to ADO.NET implementations and similar details.

LINQ to XML

You've already seen that LINQ to XML offers a slightly different syntax that operates on XML data, allowing query and data manipulation. A particular type of support for LINQ to XML is offered by Visual Basic, which includes XML literals in the language. This enhanced support simplifies the code needed to manipulate XML data. In fact, you can write a query such as the following in Visual Basic:

```
Dim book =
    <Book Title="Programming  LINQ">
        <%= From person In team
            Where person.Role = "Author"
            Select <Author><%= person.Name %></Author> %>
    </Book>
```

This query corresponds to the following C# syntax:

```
dim book =
    new XElement( "Book",
        new XAttribute( "Title", "Programming LINQ" ),
        from   person in team
        where  person.Role == "Author"
        select new XElement( "Author", person.Name ) );
```

More Info You can find more information about LINQ to XML in Chapters 12 and 13.

Summary

In this chapter, we introduced LINQ and discussed how it works. We also examined how different data sources can be queried and manipulated by using a uniform syntax that is integrated into current mainstream programming languages such as C# and Visual Basic. We took a look at the benefits offered by language integration, including declarative programming, type checking, and transparency across different type systems. We briefly presented the LINQ implementations available since .NET Framework 3.5—LINQ to Objects, LINQ to ADO.NET, and LINQ to XML—which we will cover in more detail in the remaining parts of the book.

Chapter 5

LINQ to SQL: Querying Data

The first and most obvious application of Microsoft Language Integrated Query (LINQ) is in querying an external relational database. LINQ to SQL is a LINQ component that provides the capability to query a relational Microsoft SQL Server database, offering you an object model based on available entities. In other words, you can define a set of objects that represents a thin abstraction layer over the relational data, and you can query this object model by using LINQ queries that are automatically converted into corresponding SQL queries by the LINQ to SQL engine. LINQ to SQL supports Microsoft SQL Server 2008 through SQL Server 2000 and Microsoft SQL Server Compact 3.5.

Using LINQ to SQL, you can write a simple query such as the following:

```
var query =
    from    c in Customers
    where   c.Country == "USA"
            && c.State == "WA"
    select  new {c.CustomerID, c.CompanyName, c.City };
```

This query is converted into a SQL query that is sent to the relational database:

```
SELECT CustomerID, CompanyName, City
FROM   Customers
WHERE  Country = 'USA'
  AND  Region = 'WA'
```

Important The SQL queries generated by LINQ that we show in this chapter are illustrative only. Microsoft reserves the right to independently define the SQL query that is generated by LINQ, and we sometimes use simplified queries in the text. Thus, you should not rely on the SQL query that is shown.

At this point, you might have a few questions, such as:

- How can you write a LINQ query using object names that are validated by the compiler?
- When is the SQL query generated from the LINQ query?
- When is the SQL query executed?

To understand the answers to these questions, you need to understand the entity model in LINQ to SQL, and then delve into deferred query evaluation.

Entities in LINQ to SQL

Any external data must be described with appropriate metadata bound to class definitions. Each table must have a corresponding class decorated with particular attributes. That class corresponds to a row of data and describes all columns in terms of data members of the defined type. The type can be a complete or partial description of an existing physical table, view, or stored procedure result. Only the described fields can be used inside a LINQ query for both projection and filtering. Listing 5-1 shows a simple entity definition.

Important You need to include the *System.Data.Linq* assembly in your projects to use LINQ to SQL classes and attributes. The attributes used in Listing 5-1 are defined in the *System.Data.Linq.Mapping* namespace.

LISTING 5-1 Entity definition for LINQ to SQL

```
using System.Data.Linq.Mapping;

[Table(Name="Customers")]
public class Customer {
    [Column] public string CustomerID;
    [Column] public string CompanyName;
    [Column] public string City;
    [Column(Name="Region")] public string State;
    [Column] public string Country;
}
```

The *Customer* type defines the content of a row, and each field or property decorated with *Column* corresponds to a column in the relational table. The *Name* parameter can specify a column name that is different from the data member name. (In this example, the *State* member corresponds to the Region table column.) The *Table* attribute specifies that the class is an entity representing data from a database table; its *Name* property specifies a table name that could be different from the entity name. It is common to use the singular form for the class name (which represents a single row) and the plural form for the name of the table (a set of rows).

You need a *Customers* table to build a LINQ to SQL query over *Customers* data. The *Table<T>* generic class is the right way to create such a type:

```
Table<Customer> Customers = ...;
// ...
var query =
    from    c in Customers
    // ...
```

Note To build a LINQ query over *Customers*, you need a class that implements *IEnumerable<T>*, using the *Customer* type as *T*. However, LINQ to SQL needs to implement extension methods in a different way than the LINQ to Objects implementation used in Chapter 3, "LINQ to Objects." You must use an object that implements *IQueryable<T>* to build LINQ to SQL queries. The *Table<T>* class implements *IQueryable<T>*. To include the LINQ to SQL extension, the statement *using System.Data.Linq;* must be part of the source code.

The *Customers* table object has to be instantiated. To do that, you need an instance of the *DataContext* class, which defines the bridge between the LINQ world and the external relational database. The nearest concept to *DataContext* that comes to mind is a database connection—in fact, the database connection string or the *Connection* object is a mandatory parameter for creating a *DataContext* instance. *DataContext* exposes a *GetTable<T>* method that returns a corresponding *Table<T>* for the specified type:

```
DataContext db = new DataContext("Database=Northwind");
Table<Customer> Customers = db.GetTable<Customer>();
```

Note Internally, the *DataContext* class uses the *SqlConnection* class from Microsoft ADO.NET. You can pass an existing *SqlConnection* to the *DataContext* constructor, and you can also read the connection used by a *DataContext* instance through its *Connection* property. All services related to the database connection, such as connection pooling (which is turned on by default), are accessible at the *SqlConnection* class level and are not directly implemented in the *DataContext* class.

Listing 5-2 shows the resulting code when you put all the pieces together.

LISTING 5-2 Simple LINQ to SQL query

```
DataContext db = new DataContext( ConnectionString );
Table<Customer> Customers = db.GetTable<Customer>();

var query =
    from    c in Customers
    where   c.Country == "USA"
            && c.State == "WA"
    select  new {c.CustomerID, c.CompanyName, c.City };

foreach( var row in query ) {
    Console.WriteLine( row );
}
```

The *query* variable is initialized with a query expression that forms an expression tree. An expression tree maintains a representation of the expression in memory rather than pointing to a method through a delegate. When the *foreach* loop enumerates data selected by the query, the expression tree is used to generate the corresponding SQL query, using the metadata and information from the entity classes and the referenced *DataContext* instance.

Note The *deferred execution* method used by LINQ to SQL converts the expression tree into a SQL query that is valid in the underlying relational database. The LINQ query is functionally equivalent to a string containing a SQL command, but with at least two important differences:

- The LINQ query is tied to the object model and not to the database structure.
- Its representation is semantically meaningful without requiring a SQL parser, and without being tied to a specific SQL dialect.

The expression tree can be dynamically built in memory before its use, as you will learn in Chapter 14, "Inside Expression Trees."

The data returned from the SQL query accessing *row* and placed into the *foreach* loop is then used to fill the projected anonymous type following the *select* keyword. In this example, the *Customer* class is never instantiated, and LINQ uses it only to analyze its metadata.

To explore the generated SQL command, you can use the *GetCommand* method of the *DataContext* class by accessing the *CommandText* property of the returned *DbCommand*, which contains the generated SQL query; for example:

```
Console.WriteLine( db.GetCommand( query ).CommandText );
```

A simpler way to examine the generated SQL is to call *ToString* on a LINQ to SQL query. The overridden *ToString* method produces the same result as the *GetCommand(query).CommandText* statement:

```
Console.WriteLine( query );
```

The simple LINQ to SQL query in Listing 5-2 generates the following SQL query:

```
SELECT [t0].[CustomerID], [t0].[CompanyName], [t0].[City]
FROM   [Customers] AS [t0]
WHERE  ([t0].[Country] = @p0) AND ([t0].[Region] = @p1)
```

To get a trace of all SQL statements that are sent to the database, you can assign a value to the *DataContext.Log* property, as shown here:

```
db.Log = Console.Out;
```

The next section provides more detail on how to generate entity classes for LINQ to SQL.

External Mapping

The mapping between LINQ to SQL entities and database structures has to be described through metadata information. In Listing 5-1, you saw attributes on an entity definition that

fulfills this rule. However, you can also use an external XML mapping file to decorate entity classes instead of using attributes. An XML mapping file looks like this:

```
<Database Name="Northwind">
   <Table Name="Products">
       <Type Name="Product">
           <Column Name="ProductID" Member="ProductID"
                   Storage="_ProductID" DbType="Int NOT NULL IDENTITY"
                   IsPrimaryKey="True" IsDbGenerated="True" />
```

The *Type* tag defines the relationship with an entity class, and the *Member* attribute of the *Column* tag defines the corresponding member name of the class entity (in case it differs from the column name of the table). By default, *Member* is not required and if not present, is assumed to be the same as the *Name* attribute of *Column*. This XML file usually has a *.dbml* file name extension.

More Info You can produce a Database Markup Language (DBML) file automatically with some of the tools described in Chapter 7, "LINQ to SQL: Modeling Data and Tools."

To load the DBML file, you can use an *XmlMappingSource* instance, generated by calling its *FromXml* static method, and then pass that instance to the *DataContext* derived class constructor. The following example shows how to use such syntax:

```
string path = "Northwind.dbml";
XmlMappingSource prodMapping =
        XmlMappingSource.FromXml(File.ReadAllText(path));
Northwind db = new Northwind(
        "Database=Test_Northwind;Trusted_Connection=yes",
        prodMapping
    );
```

One use of this technique is in a scenario in which different databases must be mapped to a specific data model. Differences in databases might include table and field names (for example, localized versions of the database). In general, consider this option when you need to realize a light decoupling of mapping between entity classes and the physical data structure of the database.

More Info It is beyond the scope of this book to describe the details of the XML grammar for a DBML file, but you can find that syntax described in the LinqToSqlMapping.xsd and DbmlSchema.xsd files that reside in your Program Files\Microsoft Visual Studio 10.0\Xml\Schemas directory if you have installed Microsoft Visual Studio 2010. If you do not have either of these files, you can copy the code from the following product documentation pages: "External Mapping" at *http://msdn.microsoft.com/en-us/library/bb386907.aspx* and "Code Generation in LINQ to SQL" at *http://msdn.microsoft.com/en-us/library/bb399400.aspx*.

Data Modeling

The set of entity classes that LINQ to SQL requires is a thin abstraction layer over the relational model. Each entity class defines an accessible table of data, which can be queried and modified. Modified entity instances can apply their changes to the data contained in the relational database. In this section, you will learn how to build a data model for LINQ to SQL.

More Info The options for data updates are described in Chapter 6, "LINQ to SQL: Managing Data."

DataContext

The *DataContext* class handles the communication between LINQ and external relational data sources. Each instance has a single *Connection* property that refers to a relational database. Its type is *IDbConnection*; therefore, it should not be specific to a particular database product. However, the LINQ to SQL implementation supports only SQL Server databases. Choosing between specific versions of SQL Server depends only on the connection string passed to the *DataContext* constructor.

Important The architecture of LINQ to SQL supports many data providers so that it can map to different underlying relational databases. A provider is a class that implements the *System.Data.Linq.Provider.IProvider* interface. However, that interface is declared as internal and is not documented. Microsoft supports only a SQL Server provider. The Microsoft .NET Framework supports SQL Server since version 2000 for both 32-bit and 64-bit executables, as well as SQL Server Compact 3.5 SP2.

DataContext uses metadata to map the physical structure of the relational data so that LINQ to SQL can generate the appropriate SQL code. You also use *DataContext* to call a stored procedure and persist data changes in entity class instances in the relational database.

Classes that specialize access for a particular database can be derived from *DataContext*. Such classes offer an easier way to access relational data, including members that represent available tables. You can define fields that reference existing tables in the database simply by declaring them, without a specific initialization, as in the following code:

```
public class SampleDb : DataContext {
    public SampleDb(IDbConnection connection)
            : base( connection ) {}
    public SampleDb(string fileOrServerOrConnection)
            : base( fileOrServerOrConnection ) {}
    public SampleDb(IDbConnection connection, MappingSource mapping)
            : base( connection, mapping ) {}

    public Table<Customer> Customers;
}
```

Note Table members are initialized automatically by the *DataContext* base constructor, which examines the type at execution time through Reflection, finds those members, and initializes them based on the mapping metadata.

Entity Classes

An entity class has two roles. The first role is to provide metadata to the LINQ query engine; for this, the class itself suffices—it does not require instantiation of an entity instance. The second role is to provide storage for data read from the relational data source, as well as to track possible updates and support their submission back to the relational data source.

An entity class is any reference type definition decorated with the *Table* attribute. You cannot use a *struct* (which is a value type) for this. The *Table* attribute can have a *Name* parameter that defines the name of the corresponding table in the database. If *Name* is omitted, the name of the class is used as the default:

```
[Table(Name="Products")] public class Product { ... }
```

Note Although the term commonly used is *table*, nothing prevents you from using an updatable view in place of a table name in the *Name* parameter. Using a non-updatable view will also work—at least until you try to update data without using that entity class.

An entity class can have any number and type of members. Just remember that only those data members or properties decorated with the *Column* attribute are significant in defining the mapping between the entity class and the corresponding table in the database:

```
[Column] public int ProductID;
```

An entity class should have a unique key. This key is necessary to support unique identity (more on this later), to identify corresponding rows in database tables, and to generate SQL statements that update data. If you do not have a primary key, entity class instances can be created but are not modifiable. The Boolean *IsPrimaryKey* property of the *Column* attribute, when set to *true*, states that the column belongs to the primary key of the table. If the primary key used is a composite key, all the columns that form the primary key will have *IsPrimaryKey=true* in their parameters:

```
[Column(IsPrimaryKey=true)] public int ProductID;
```

By default, a column mapping uses the same name as the member to which the *Column* attribute is applied. You can specify a different name in the *Name* parameter. For example, the following *Price* member corresponds to the *UnitPrice* field in the database table:

```
[Column(Name="UnitPrice")] public decimal Price;
```

If you want to filter data access through member property accessors, you have to specify the underlying storage member using the *Storage* parameter. If you specify a *Storage* parameter, LINQ to SQL bypasses the public property accessor and interacts directly with the underlying value. Understanding this is particularly important if you want to track only the modifications made by your code and not the read/write operations made by the LINQ framework. In the following code, the *ProductName* property is accessed for each read/write operation made by your code:

```
[Column(Storage="_ProductName")]
public string ProductName {
    get { return this._ProductName; }
    set { this.OnPropertyChanging("ProductName");
          this._ProductName = value;
          this.OnPropertyChanged("ProductName");
    }
}
```

In contrast, LINQ to SQL performs a direct read/write operation on the *_ProductName* data member when it executes a LINQ operation.

The correspondence between relational type and .NET Framework type assumes a default relational type that corresponds to the .NET Framework type used. Whenever you need to define a different type, you can use the *DBType* parameter, specifying a valid type by using valid SQL syntax for your relational data source. You need to use this parameter only when you want to create a database schema starting from entity class definitions (a process described in Chapter 6). Here's an example of the *DBType* parameter in use:

```
[Column(DBType="NVARCHAR(20)")] public string QuantityPerUnit;
```

When the database automatically generates a column value (such as with the IDENTITY keyword in SQL Server), you might want to synchronize the entity class member with the generated value whenever you insert an entity instance into the database. To do that, you need to set the *IsDBGenerated* parameter for that member to *true*, and you also need to adapt the *DBType* accordingly—for example, by adding the *IDENTITY* modifier for SQL Server tables:

```
[Column(DBType="INT NOT NULL IDENTITY",
        IsPrimaryKey=true, IsDBGenerated=true)]
public int ProductID;
```

It is worth mentioning that a specific *CanBeNull* parameter exists. This parameter is used to specify that the value can contain the null value; however, it is important to note that the *NOT NULL* clause in *DBType* is still necessary if you want to create such a condition in a database created by LINQ to SQL:

```
[Column(DBType="INT NOT NULL IDENTITY", CanBeNull=false,
        IsPrimaryKey=true, IsDBGenerated=true)]
public int ProductID;
```

Other parameters that are relevant in updating data are *AutoSync*, *Expression*, *IsVersion*, and *UpdateCheck*.

More Info Chapter 6 provides a more detailed explanation of the parameters *IsVersion*, *Expression*, *UpdateCheck*, *AutoSync*, and *IsDBGenerated*.

Entity Inheritance

Sometimes a single table contains many types of entities. For example, imagine a list of contacts—some might be customers, others might be suppliers, and still others might be company employees. From a data point of view, each entity can have specific fields. (For example, a customer can have a discount field, which is not relevant for employees and suppliers.) From a business logic point of view, each entity can implement different business rules. The best way to model this kind of data in an object-oriented environment is by using inheritance to create a hierarchy of specialized classes. LINQ to SQL allows a set of classes derived from the same base class to map to the same relational table.

The *InheritanceMapping* attribute decorates the base class of a hierarchy, indicating the corresponding derived classes that are based on the value of a special *discriminator column*. The *Code* parameter defines a possible value, and the *Type* parameter defines the corresponding derived type. The discriminator column is defined by setting the *IsDiscriminator* argument to *true* in the *Column* attribute specification.

Listing 5-3 provides an example of a hierarchy based on the *Contacts* table of the *Northwind* sample database.

LISTING 5-3 Hierarchy of classes based on contacts

```
[Table(Name="Contacts")]
[InheritanceMapping(Code = "Customer", Type = typeof(CustomerContact))]
[InheritanceMapping(Code = "Supplier", Type = typeof(SupplierContact))]
[InheritanceMapping(Code = "Shipper", Type = typeof(ShipperContact))]
[InheritanceMapping(Code = "Employee", Type = typeof(Contact), IsDefault = true)]
public class Contact {
    [Column(IsPrimaryKey=true)] public int ContactID;
    [Column(Name="ContactName")] public string Name;
    [Column] public string Phone;
    [Column(IsDiscriminator = true)] public string ContactType;
}

public class CompanyContact : Contact {
    [Column(Name="CompanyName")] public string Company;
}
```

```
public class CustomerContact : CompanyContact {
}

public class SupplierContact : CompanyContact {
}

public class ShipperContact : CompanyContact {
    public string Shipper {
        get { return Company; }
        set { Company = value; }
    }
}
```

Contact is the base class of the hierarchy. If the contact is a *Customer*, *Supplier*, or *Shipper*, the corresponding classes derive from an intermediate *CompanyContact* type, which defines the *Company* field corresponding to the CompanyName column in the source table. The *CompanyContact* intermediate class is necessary because you cannot reference the same column (CompanyName) in more than one field, even if this happens in different classes in the same hierarchy. The *ShipperContact* class defines a *Shipper* property that exposes the same value of *Company* but with a different semantic meaning.

Important This approach requires that you flatten the union of all possible data columns for the whole hierarchy into a single table. If you have a normalized database, you might have data for different entities separated in different tables. You can define a view to use LINQ to SQL to support entity hierarchy, but to update data you must make the view updatable.

The level of abstraction offered by having different entity classes in the same hierarchy is well described by the sample queries shown in Listing 5-4. The *queryTyped* query uses the *OfType* operator, whereas *queryFiltered* query relies on a standard *where* condition to filter out contacts that are not customers.

LISTING 5-4 Queries using a hierarchy of entity classes

```
var queryTyped =
    from    c in contacts.OfType<CustomerContact>()
    select  c;

var queryFiltered =
    from    c in contacts
    where   c is CustomerContact
    select  c;
```

```
foreach( var row in queryTyped ) {
    Console.WriteLine( row.Company );
}

// We need an explicit cast to access the CustumerContact members
foreach( CustomerContact row in queryFiltered ) {
    Console.WriteLine( row.Company );
}
```

The SQL queries produced by these LINQ queries are functionally identical to the following (although the actual query is different because of generalization coding):

```
SELECT [t0].[ContactType], [t0].[CompanyName] AS [Company],
       [t0].[ContactID], [t0].[ContactName] AS [Name],
       [t0].[Phone]
FROM   [Contacts] AS [t0]
WHERE  [t0].[ContactType] = 'Customer'
```

The difference between *queryTyped* and *queryFiltered* queries lies in the returned type. A *queryTyped* query returns a sequence of *CustomerContact* instances, whereas *queryFiltered* returns a sequence of the base class *Contact*. With *queryFiltered*, you need to explicitly cast the result into a *CustomerContact* type if you want to access the *Company* property.

Unique Object Identity

An instance of an entity class stores an in-memory representation of table row data. If you instantiate two different entities containing the same row from the same *DataContext*, both will reference the same in-memory object. In other words, object identity (same references) maintains data identity (same table row) using the entity unique key. The LINQ to SQL engine ensures that the same object reference is used when an entity instantiated from a query result coming from the same *DataContext* is already in memory. This check does not happen if you create an instance of an entity by yourself or in a different *DataContext* (regardless of the real data source). In Listing 5-5, you can see that *c1* and *c2* reference the same *Contact* instance, even if they originate from two different queries, whereas *c3* is a different object, even if its content is equivalent to the others.

Note If you want to force data from the database to reload using the same *DataContext*, you must use the *Refresh* method of the *DataContext* class. Chapter 6 discusses this in more detail.

LISTING 5-5 Object identity

```
var queryTyped =
    from    c in contacts.OfType<CustomerContact>()
    orderby c.ContactID
    select  c;

var queryFiltered =
    from    c in contacts
    where   c is CustomerContact
    orderby c.ContactID
    select  c;

Contact c1 = null;
Contact c2 = null;
foreach( var row in queryTyped.Take(1) ) {
    c1 = row;
}
foreach( var row in queryFiltered.Take(1) ) {
    c2 = row;
}
Contact c3 = new Contact();
c3.ContactID = c1.ContactID;
c3.ContactType = c1.ContactType;
c3.Name = c1.Name;
c3.Phone = c1.Phone;
Debug.Assert( c1 == c2 ); // same instance
Debug.Assert( c1 != c3 ); // different objects
```

Entity Constraints

Entity classes support the maintenance of valid relationships between entities, just like the support offered by foreign keys in a standard relational environment. However, the entity classes cannot represent all possible check constraints of a relational table. No attributes are available to specify the same alternate keys (unique constraint), triggers, and check expressions that can be defined in a relational database. This fact is relevant when you start to manipulate data using entity classes because you cannot guarantee that an updated value will be accepted by the underlying database. (For example, it could have a duplicate unique key.) However, because you can load into entity instances only parts (rows) of the whole table, these kinds of checks are not possible without accessing the relational database anyway.

Associations Between Entities

Relationships between entities in a relational database are modeled on the concept of foreign keys in one table referring to primary keys of another table. Class entities can use the same concept through the *Association* attribute, which can describe both sides of a one-to-many relationship described by a foreign key.

EntityRef

Let's start with the concept of *lookup*, which is the typical operation used to get the customer related to one order. *Lookup* can be seen as the direct translation into the entity model of the foreign key relationship existing between the *CustomerID* column of the Orders table and the primary key of the Customers table. In the example entity model, the *Order* entity class will have a *Customer* property (of type *Customer*) that shows the customer data. This property is decorated with the *Association* attribute and stores its information in an *EntityRef<Customer>* member (named *_Customer*), which enables deferred loading of references (as you will see shortly). Listing 5-6 shows the definition of this association.

LISTING 5-6 Association *EntityRef*

```
[Table(Name="Orders")]
public class Order {
    [Column(IsPrimaryKey=true)] public int OrderID;
    [Column] private string CustomerID;
    [Column] public DateTime? OrderDate;

    [Association(Storage="_Customer", ThisKey="CustomerID", IsForeignKey=true)]
    public Customer Customer {
        get { return this._Customer.Entity; }
        set { this._Customer.Entity = value; }
    }

    private EntityRef<Customer> _Customer;
}
```

As you can see, the CustomerID column must be defined in *Order*; otherwise, it would not be possible to obtain the related *Customer*. The *IsForeignKey* argument specifies that *Order* is the child side of a parent-child relationship. The *ThisKey* argument of the *Association* attribute indicates the "foreign key" column (which would be a comma-separated list if more columns were involved for a composite key) that defines the relationship between entities. If you want to hide this detail in the entity properties, you can declare that column as private, just as in the *Order* class shown earlier.

Note There are two other arguments for the *Association* attribute. One is *IsUnique*, which must be *true* whenever the foreign key also has a uniqueness constraint. In that case, the relationship with the parent table is one-to-one instead of many-to-one. The other argument is *Name*, which is used only to define the name of the constraint for a database generated from the metadata by using the *DataContext.CreateDatabase* method, which will be described in Chapter 6.

Using the *Order* class in a LINQ query, you can specify a *Customer* property in a filter without writing a join between *Customer* and *Order* entities. In the following query, the *Country* member of the related *Customer* is used to filter orders that come from customers of a particular *Country*:

```
Table<Order> Orders = db.GetTable<Order>();
var query =
    from   o in Orders
    where  o.Customer.Country == "USA"
    select o.OrderID;
```

The previous query is translated into a SQL JOIN like the following one:

```
SELECT    [t0].[OrderID]
FROM      [Orders] AS [t0]
LEFT JOIN [Customers] AS [t1]
       ON [t1].[CustomerID] = [t0].[CustomerID]
WHERE     [t1].[Country] = "USA"
```

Until now, we have used entity relationships only for their metadata in building LINQ queries. When an instance of an entity class is created, a reference to another entity (such as the previous *Customer* property) works with a technique called *deferred loading*. The related *Customer* entity is not instantiated and loaded into memory from the database until it is accessed either in read or write mode.

Note *EntityRef<T>* is a wrapper class that is instantiated with the container object (a class derived from *DataContext*) to give a valid reference for any access to the referenced entity. Each read/write operation is filtered by a property *getter* and *setter*, which execute a query to load data from the database the first time this entity is accessed if it is not already in memory.

In other words, to generate a SQL query to populate the *Customer*-related entity when the *Country* property is accessed, you use the following code:

```
var query =
    from   o in Orders
    where  o.OrderID == 10528
    select o;

foreach( var row in query ) {
    Console.WriteLine( row.Customer.Country );
}
```

The process of accessing the *Customer* property involves determining whether the related *Customer* entity is already in memory for the current *DataContext*. If it is, that entity is

accessed; otherwise, the following SQL query is executed and the corresponding *Customer* entity is loaded in memory and then accessed:

```
SELECT [t0].[Country], [t0].[CustomerID], [t0].[CompanyName]
FROM   [Customers] AS [t0]
WHERE  [t0].[CustomerID] = "GREAL"
```

The GREAL string is the *CustomerID* value for order 10528. As you can see, the SELECT statement queries all columns declared in the *Customer* entity, even if they are not used in the expression that accessed the *Customer* entity. (In this case, the executed code never referenced the *CompanyName* member.)

EntitySet

The other side of an association is a table that is referenced from another table through its primary key. Although this is an implicit consequence of the foreign key constraint in a relational model, you need to explicitly define this association in the entity model. If the Customers table is referenced from the Orders table, you can define an *Orders* property in the *Customer* class that represents the set of *Order* entities related to a given *Customer*. The relationship is implemented by an instance of *EntitySet*<*Order*>, which is a wrapper class over the sequence of related orders. You might want to directly expose this *EntitySet*<*T*> type, as in the code shown in Listing 5-7. In that code, the *OtherKey* argument of the *Association* attribute specifies the name of the member on the related type (*Order*) that defines the association between *Customer* and the set of *Order* entities.

LISTING 5-7 *Association EntitySet* (visible)

```
[Table(Name="Customers")]
public class Customer {
    [Column(IsPrimaryKey=true)] public string CustomerID;
    [Column] public string CompanyName;
    [Column] public string Country;

    [Association(OtherKey="CustomerID")]
    public EntitySet<Order> Orders;
}
```

You might also decide to expose *Orders* as a property, as in the declaration shown in Listing 5-8. In this case, the *Storage* argument of the *Association* attribute specifies the *EntitySet*<*T*> for physical storage. You could make only an *ICollection*<*Order*> visible outside the *Customer* class, instead of an *EntitySet*<*Order*>, but this is not a common practice.

LISTING 5-8 *Association EntitySet* (hidden)

```
public class Customer {
    [Column(IsPrimaryKey=true)] public string CustomerID;
    [Column] public string CompanyName;
    [Column] public string Country;

    private EntitySet<Order> _Orders;

    [Association(OtherKey="CustomerID", Storage="_Orders")]
    public EntitySet<Order> Orders {
        get { return this._Orders; }
        set { this._Orders.Assign(value); }
    }
    public Customer() {
        this._Orders = new EntitySet<Order>();
    }
}
```

With both models of association declaration, you can use the *Customer* class in a LINQ query, accessing the related *Order* entities without the need to write a join. You simply specify the *Orders* property. The next query returns the names of customers who placed more than 20 orders:

```
Table<Customer> Customers = db.GetTable<Customer>();
var query =
    from   c in Customers
    where  c.Orders.Count > 20
    select c.CompanyName;
```

The previous LINQ query is translated into a SQL query like the following one:

```
SELECT [t0].[CompanyName]
FROM   [Customers] AS [t0]
WHERE ( SELECT COUNT(*)
        FROM [Orders] AS [t1]
        WHERE [t1].[CustomerID] = [t0].[CustomerID]
       ) > 20
```

This example creates no *Order* entity instances. The *Orders* property serves only as a metadata source to generate the desired SQL query. If you return a *Customer* entity from a LINQ query, you can access the *Orders* of a customer on demand:

```
var query =
    from   c in Customers
    where  c.Orders.Count > 20
    select c;
```

```
foreach( var row in query ) {
    Console.WriteLine( row.CompanyName );
    foreach( var order in row.Orders ) {
        Console.WriteLine( order.OrderID );
    }
}
```

The preceding code uses deferred loading. Each time you access the *Orders* property of a customer for the first time (as indicated by the bold in the preceding code), a query like the following one (which uses the *@p0* parameter to filter *CustomerID*) is sent to the database:

```
SELECT [t0].[OrderID], [t0].[CustomerID]
FROM   [Orders] AS [t0]
WHERE  [t0].[CustomerID] = @p0
```

If you want to load all orders for all customers into memory using only one query to the database, you need to request *immediate loading* instead of deferred loading. To do that, you have two options. The first approach, which is demonstrated in Listing 5-9, is to force the inclusion of an *EntitySet* using a *DataLoadOptions* instance and the call to its *LoadWith<T>* method.

LISTING 5-9 Use of *DataLoadOptions* and *LoadWith<T>*

```
DataContext db = new DataContext( ConnectionString );
Table<Customer> Customers = db.GetTable<Customer>();

DataLoadOptions loadOptions = new DataLoadOptions();
loadOptions.LoadWith<Customer>( c => c.Orders );
db.LoadOptions = loadOptions;
var query =
     from   c in Customers
     where  c.Orders.Count > 20
     select c;
```

The second option is to return a new entity that explicitly includes the *Orders* property for the *Customer*:

```
var query =
    from   c in Customers
    where  c.Orders.Count > 20
    select new { c.CompanyName, c.Orders };
```

These LINQ queries send a SQL query to the database to get all customers who placed more than 20 orders, including the entire order list for each customer. That SQL query might be similar to the one shown in the following code:

```
SELECT [t0].[CompanyName], [t1].[OrderID], [t1].[CustomerID], (
    SELECT COUNT(*)
    FROM [Orders] AS [t3]
    WHERE [t3].[CustomerID] = [t0].[CustomerID]
```

```
    ) AS [value]
FROM [Customers] AS [t0]
LEFT OUTER JOIN [Orders] AS [t1] ON [t1].[CustomerID] = [t0].[CustomerID]
WHERE (
    SELECT COUNT(*)
    FROM [Orders] AS [t2]
    WHERE [t2].[CustomerID] = [t0].[CustomerID]
    ) > 20
ORDER BY [t0].[CustomerID], [t1].[OrderID]
```

Note As you can see, a single SQL statement is here and the LINQ to SQL engine parses the result, extracting different entities (*Customers* and *Orders*). Keeping the result ordered by *CustomerID*, the engine can build in-memory entities and relationships in a faster way.

You can filter the subquery produced by relationship navigation. Suppose you want to see only customers who placed at least five orders in 1997, and you want to load only these orders. You can use the *AssociateWith<T>* method of the *DataLoadOptions* class to do that, as demonstrated in Listing 5-10.

LISTING 5-10 Use of *DataLoadOptions* and *AssociateWith<T>*

```
DataLoadOptions loadOptions = new DataLoadOptions();
loadOptions.AssociateWith<Customer>(
    c => from   o in c.Orders
         where  o.OrderDate.Value.Year == 1997
         select o);
db.LoadOptions = loadOptions;
var query =
     from   c in Customers
     where  c.Orders.Count > 5
     select c;
```

The Microsoft Visual C# filter condition (*o.OrderDate.Value.Year == 1997*) is translated into the following SQL expression:

```
(DATEPART(Year, [t2].[OrderDate]) = 1997)
```

AssociateWith<T> can also control the initial ordering of the collection. To do that, you can simply add an order condition to the query passed as an argument to *AssociateWith<T>*. For example, if you want to get the orders for each customer starting from the newest one, add the *orderby* line shown in bold in the following code:

```
loadOptions.AssociateWith<Customer>(
    c => from    o in c.Orders
         where   o.OrderDate.Value.Year == 1997
         orderby o.OrderDate descending
         select  o);
```

Using *AssociateWith<T>* alone does not apply the immediate loading behavior. If you want both immediate loading and filtering through a relationship, you have to call both the *LoadWith<T>* and *AssociateWith<T>* methods. The order of these calls is not relevant. For example, you can write the following code:

```
DataLoadOptions loadOptions = new DataLoadOptions();
loadOptions.AssociateWith<Customer>(
    c => from   o in c.Orders
         where  o.OrderDate.Value.Year == 1997
         select o);
loadOptions.LoadWith<Customer>( c => c.Orders );
db.LoadOptions = loadOptions;
```

Loading all data into memory using a single query might be a better approach if you are sure you will access all data that is loaded, because you will spend less time in round-trip latency. However, this technique will consume more memory and bandwidth when the typical access to a graph of entities is random. Think about these details when you decide how to query your data model.

Graph Consistency

Relationships are bidirectional between entities—when an update is made on one side, the other side should be kept synchronized. LINQ to SQL does not automatically manage this kind of synchronization, which has to be done by the class entity implementation. Instead, LINQ to SQL offers an implementation pattern that is also used by code-generation tools such as SQLMetal, a tool that is part of the Windows Software Development Kit (SDK) (and has been part of the .NET Framework SDK since Microsoft .NET Framework 3.5), or the LINQ to SQL class generator included with Visual Studio. Chapter 7 describes both these tools. This pattern is based on the *EntitySet<T>* class on one side and on the complex *setter* accessor on the other side. Take a look at the tools-generated code if you are interested in the implementation details of this pattern.

Change Notification

You will see in Chapter 6 that LINQ to SQL is able to track changes in entities, submitting equivalent changes to the database. This process is implemented by default through an algorithm that compares an object's content with its original values, requiring a copy of each tracked object. The memory consumption can be high, but it can be optimized if entities participate in the change tracking service by announcing when an object has been changed.

The implementation of change notification requires an entity to expose all its data through properties implementing the *System.ComponentModel.INotifyPropertyChanging* interface. Each property *setter* needs to call the *PropertyChanging* method of *DataContext*. Tools-generated code for entities (such as that emitted by SQLMetal and Visual Studio) already implement this pattern.

More Info For more information about change tracking, see the product documentation "Object States and Change-Tracking (LINQ to SQL)" at *http://msdn.microsoft.com/en-us/library/bb386982.aspx*.

Relational Model vs. Hierarchical Model

The entity model used by LINQ to SQL defines a set of objects that maps database tables into objects that can be used and manipulated by LINQ queries. The resulting model represents a paradigm shift that has been revealed in descriptions of associations between entities because it moves from a relational model (tables in a database) to a hierarchical or graph model (objects in memory).

A hierarchical/graph model is the natural way to manipulate objects in a program written in C# or Microsoft Visual Basic. When you try to consider how to translate an existing SQL query into a LINQ query, this is the major conceptual obstacle you encounter. In LINQ, you can write a query using joins between separate entities, just as you do in SQL. However, you can also write a query that uses the existing relationships between entities, as we did with *EntitySet* and *EntityRef* associations.

Important Remember that SQL does not make use of relationships between entities when querying data. Those relationships exist only to define the data integrity conditions. LINQ does not have the concept of *referential integrity*, but it makes use of relationships to define possible navigation paths in the data.

Data Querying

A LINQ to SQL query gets sent to the database only when the program needs to read data. For example, the following *foreach* loop iterates rows returned from a table:

```
var query =
    from    c in Customers
    where   c.Country == "USA"
    select  c.CompanyName;
```

```
 foreach( var company in query ) {
    Console.WriteLine( company );
}
```

The code generated by the *foreach* statement is equivalent to the following code. The exact moment the query is executed corresponds to the *GetEnumerator* call:

```
 // GetEnumerator sends the query to the database
IEnumerator<string> enumerator = query.GetEnumerator();
while (enumerator.MoveNext()) {
    Console.WriteLine( enumerator.Current );
}
```

Writing more *foreach* loops in the same query generates an equal number of calls to *GetEnumerator*, and thus an equal number of repeated executions of the same query. If you want to iterate the same data many times, you might prefer to cache data in memory. Using *ToList* or *ToArray*, you can convert the results of a query into a *List* or an *Array*, respectively. When you call these methods, the SQL query is sent to the database immediately:

```
// ToList() sends the query to the database
var companyNames = query.ToList();
```

You might want to send the query to the database several times when you manipulate the LINQ query between data iterations. For example, you might have an interactive user interface that allows the user to add a new filter condition for each iteration of data. In Listing 5-11, the *DisplayTop* method shows only the first few rows of the result; query manipulation between calls to *DisplayTop* simulates a user interaction that ends in a new filter condition each time.

More Info Listing 5-11 shows a very simple technique for query manipulation, adding more restrictive filter conditions to an existing query represented by an *IQueryable<T>* object. Chapter 14 describes the techniques to dynamically build a query tree in a more flexible way.

LISTING 5-11 Query manipulation

```
static void QueryManipulation() {
    DataContext db = new DataContext( ConnectionString );
    Table<Customer> Customers = db.GetTable<Customer>();
    db.Log = Console.Out;

    // All Customers
    var query =
        from    c in Customers
        select  new {c.CompanyName, c.State, c.Country };
```

```
    DisplayTop( query, 10 );

    // User interaction adds a filter
    // to the previous query
    // Customers from USA
    query =
        from   c in query
        where  c.Country == "USA"
        select c;

    DisplayTop( query, 10 );

    // User interaction adds another
    // filter to the previous query
    // Customers from WA, USA
    query =
        from   c in query
        where  c.State == "WA"
        select c;

    DisplayTop( query, 10 );
}

static void DisplayTop<T>( IQueryable<T> query, int rows ) {
    foreach( var row in query.Take(rows)) {
        Console.WriteLine( row );
    }
}
```

Important The previous example used *IQueryable<T>* as the *DisplayTop* parameter. If you pass *IEnumerable<T>* instead, the results would *appear* identical, but the query sent to the database would not contain the *TOP (rows)* clause to filter data directly on the database. Passing *IEnumerable<T>* uses a different set of extension methods to resolve the *Take* operator, which does not generate a new expression tree. Refer to Chapter 2, "LINQ Syntax Fundamentals," for an introduction to the differences between *IEnumerable<T>* and *IQueryable<T>*.

One common query reads a single row from a table, defining a condition that is guaranteed to be unique, such as a record key, shown in the following code:

```
var query =
    from    c in db.Customers
    where   c.CustomerID == "ANATR"
    select  c;

var enumerator = query.GetEnumerator();
if (enumerator.MoveNext()) {
    var customer = enumerator.Current;
    Console.WriteLine( "{0} {1}", customer.CustomerID, customer.CompanyName );
}
```

When you know a query will return a single row, use the *Single* operator to state your intention. Using this operator, you can write the previous code in a more compact way:

```
var customer = db.Customers.Single( c => c.CustomerID == "ANATR" );
Console.WriteLine( "{0} {1}", customer.CustomerID, customer.CompanyName );
```

However, it is important to note that calling *Single* has a different semantic than the previous equivalent *query*. Calling *Single* generates a query to the database only if the desired entity (in this case, the *Customer* with *ANATR* as *CustomerID*) is not already in memory. If you want to read the data from the database, you need to call the *DataContext.Refresh* method:

```
db.Refresh(RefreshMode.OverwriteCurrentValues, customer);
```

More Info Chapter 6 contains more information about the entity life cycle.

Projections

The transformation from an expression tree to a SQL query requires the complete understanding of the query operations sent to the LINQ to SQL engine. This transformation affects the use of object initializers. You can use projections through the *select* keyword, as in the following example:

```
var query =
   from    c in Customers
   where   c.Country == "USA"
   select  new {c.CustomerID, Name = c.CompanyName.ToUpper()} into r
   orderby r.Name
   select  r;
```

The whole LINQ query is translated into this SQL statement:

```
SELECT [t1].[CustomerID], [t1].[value] AS [Name]
FROM ( SELECT [t0].[CustomerID],
              UPPER([t0].[CompanyName]) AS [value],
              [t0].[Country]
       FROM [Customers] AS [t0]
     ) AS [t1]
WHERE    [t1].[Country] = "USA"
ORDER BY [t1].[value]
```

As you can see, the *ToUpper* method has been translated into an UPPER T-SQL function call. To do that, the LINQ to SQL engine needs a deep knowledge of the meaning of any operation in the expression tree. Consider this query:

```
var queryBad =
    from    c in Customers
    where   c.Country == "USA"
    select  new CustomerData( c.CustomerID, c.CompanyName.ToUpper()) into r
    orderby r.Name
    select  r;
```

The preceding example calls a *CustomerData* constructor that can do anything a piece of Intermediate Language (IL) code can do. In other words, there is no semantic value in calling a constructor other than the initial assignment of the instance created. The consequence is that LINQ to SQL cannot correctly translate this syntax into equivalent SQL code, and it throws an exception if you try to execute the query. However, you can safely use a parameterized constructor in the final projection of a query, as in the following example:

```
var queryParamConstructor =
    from    c in Customers
    where   c.Country == "USA"
    orderby c.CompanyName
    select  new CustomerData( c.CustomerID, c.CompanyName.ToUpper() );
```

If you only need to initialize an object, use object initializers instead of a parameterized constructor call, as in the following query:

```
var queryGood =
    from    c in Customers
    where   c.Country == "USA"
    select  new CustomerData { CustomerID = c.CustomerID,
                               Name = c.CompanyName.ToUpper() } into r
    orderby r.Name
    select  r;
```

Important Always use object initializers to encode projections in LINQ to SQL. Use parameterized constructors only in the final projection of a query.

Stored Procedures and User-Defined Functions

Accessing data through stored procedures and user-defined functions (UDFs) requires the definition of corresponding methods decorated with attributes. With this definition, you can write LINQ queries in a strongly typed form. From the point of view of LINQ, it makes no difference whether a stored procedure or UDF is written in T-SQL or SQLCLR, but there are some details you must know to handle differences between stored procedures and UDFs.

Note Because many of you will automatically generate specialized *DataContext* derived classes, we will focus attention on the most important concepts that you should know to effectively use these objects. If you want to create these wrappers manually, please refer to the product documentation "DataContext Class" at *http://msdn.microsoft.com/library/system.data.linq.datacontext.aspx* for a detailed list of the attributes and their arguments.

Stored Procedures

Consider this *Customers by City* stored procedure:

```
CREATE PROCEDURE [dbo].[Customers By City]( @param1 NVARCHAR(20) )
AS BEGIN
    SET NOCOUNT ON;
    SELECT CustomerID, ContactName, CompanyName, City
    FROM   Customers AS c
    WHERE  c.City = @param1
END
```

You can define a method decorated with a *Function* attribute that calls the stored procedure through the *DataContext.ExecuteMethodCall* method. Listing 5-12 defines *CustomersByCity* as a member of a class derived from *DataContext*.

LISTING 5-12 Stored procedure declaration

```
class SampleDb : DataContext {
    // ...
    [Function(Name = "Customers by City", IsComposable = false)]
    public ISingleResult<CustomerInfo> CustomersByCity(string param1) {
        IExecuteResult executeResult =
            this.ExecuteMethodCall(
                    this,
                    (MethodInfo) (MethodInfo.GetCurrentMethod()),
                    param1);
        ISingleResult<CustomerInfo> result =
            (ISingleResult<CustomerInfo>) executeResult.ReturnValue;
        return result;
    }
}
```

The *ExecuteMethodCall* is declared in this way:

```
IExecuteResult ExecuteMethodCall( object instance,
                                  MethodInfo methodInfo,
                                  params object[] parameters)
```

The method's first parameter is the instance, which is not required if you call a static method. The second parameter is a metadata description of the method to call, which could be obtained through Reflection, as shown in Listing 5-12. The third parameter is an array containing parameter values to pass to the method that is called.

CustomersByCity returns an instance of *ISingleResult<CustomerInfo>*, which implements *IEnumerable<CustomerInfo>* and can be enumerated in a *foreach* statement like this one:

```
SampleDb db = new SampleDb( ConnectionString );
foreach( var row in db.CustomersByCity( "London" )) {
    Console.WriteLine( "{0} {1}", row.CustomerID, row.CompanyName );
}
```

As you can see in Listing 5-12, you have to access the *IExecuteResult* interface returned by *ExecuteMethodCall* to get the desired result. This requires further explanation. You use the same *Function* attribute to decorate a method wrapping either a stored procedure or a UDF. The discrimination between these constructs is made by the *IsComposable* argument of the *Function* attribute: if it is *false*, the following method wraps a stored procedure; if it is *true*, the method wraps a user-defined function.

Note The name *IsComposable* relates to the composability of user-defined functions in a query expression. You will see an example of this when the mapping of UDFs is described in the next section of this chapter.

The *IExecuteResult* interface has a simple definition:

```
public interface IExecuteResult : IDisposable {
    object GetParameterValue(int parameterIndex);
    object ReturnValue { get; }
}
```

The *GetParameterValue* method allows access to the output parameters of a stored procedure. You need to cast this result to the correct type, also passing the ordinal position of the output parameter in *parameterIndex*.

The *ReturnValue* read-only property is used to access the return value of a stored procedure or UDF. The scalar value returned is accessible with a cast to the correct type: a stored procedure always returns an integer, whereas the type of a UDF function can be different. However, when the results are tabular, you use *ISingleResult<T>* to access a single result set, or *IMultipleResults* to access multiple result sets.

You always need to know the metadata of all possible returned result sets, applying the right types to the generic interfaces used to return data. *ISingleResult<T>* is a simple wrapper of *IEnumerable<T>* that also implements *IFunctionResult*, which has a *ReturnValue* read-only property that acts as the *IExecuteResult.ReturnValue* property you have already seen:

```
public interface IFunctionResult {
    object ReturnValue { get; }
}
public interface ISingleResult<T> :
    IEnumerable<T>, IEnumerable, IFunctionResult, IDisposable { }
```

You saw an example of *ISingleResult<T>* in Listing 5-12. We wrote the *CustomersByCity* wrapper in a verbose way to better illustrate the internal steps necessary to access the returning data.

Whenever you have multiple result sets from a stored procedure, you call the *IMultipleResult.GetResult<T>* method for each result set sequentially and specify the correct *T* type for the expected result. *IMultipleResults* also implements *IFunctionResult*, thereby also offering a *ReturnValue* read-only property:

```
public interface IMultipleResults : IFunctionResult, IDisposable {
      IEnumerable<TElement> GetResult<TElement>();
}
```

Consider the following stored procedure that returns two result sets with different structures:

```
CREATE PROCEDURE TwoCustomerGroups
AS BEGIN
    SELECT  CustomerID, ContactName, CompanyName, City
    FROM   Customers AS c
    WHERE  c.City = 'London'

    SELECT  CustomerID, CompanyName, City
    FROM   Customers AS c
    WHERE  c.City = 'Torino'
END
```

The results returned from this stored procedure can be stored in the following *CustomerInfo* and *CustomerShortInfo* types, which do not require any attributes in their declarations:

```
public class CustomerInfo {
    public string CustomerID;
    public string CompanyName;
    public string City;
    public string ContactName;
}

public class CustomerShortInfo {
    public string CustomerID;
    public string CompanyName;
    public string City;
}
```

The declaration of the LINQ counterpart of the *TwoCustomerGroups* stored procedure should be like the one shown in Listing 5-13.

LISTING 5-13 Stored procedure with multiple results

```
class SampleDb : DataContext {
    // ...
    [Function(Name = "TwoCustomerGroups", IsComposable = false)]
    [ResultType(typeof(CustomerInfo))]
    [ResultType(typeof(CustomerShortInfo))]
    public IMultipleResults TwoCustomerGroups() {
        IExecuteResult executeResult =
                 this.ExecuteMethodCall(
                     this,
                     (MethodInfo) (MethodInfo.GetCurrentMethod()));
        IMultipleResults result =
            (IMultipleResults) executeResult.ReturnValue;
        return result;
    }
}
```

Each result set has a different type. When calling each *GetResult<T>*, you need to specify the correct type, which needs at least a public member with the same name for each returned column. If you specify a type with more public members than available columns, the "missing" members will have a default value. Moreover, each returned type has to be declared by using a *ResultType* attribute that decorates the *TwoCustomerGroups* method, as you can see in Listing 5-13. In the next sample, the first result set must match the *CustomerInfo* type, and the second result set must correspond to the *CustomerShortInfo* type:

```
IMultipleResults results = db.TwoCustomerGroups();
foreach( var row in results.GetResult<CustomerInfo>()) {
    // Access to CustomerInfo instance
}
foreach( var row in results.GetResult<CustomerShortInfo>()) {
    // Access to CustomerShortInfo instance
}
```

Remember that the order of *ResultType* attributes is not relevant, but you have to pay attention to the order of the *GetResult<T>* calls. The first result set will be mapped from the first *GetResult<T>* call, and so on, regardless of the parameter type used. For example, if you invert the previous two calls, asking for *CustomerShortInfo* before *CustomerInfo*, you get no error, but you do get an empty string for the *ContactName* of the second result set mapped to *CustomerInfo*.

Important The order of *GetResult<T>* calls is relevant and must correspond to the order of returned result sets. Conversely, the order of *ResultType* attributes applied to the method representing a stored procedure is not relevant.

Another use of *IMultipleResults* is the case in which a stored procedure can return different types based on parameters. For example, consider the following stored procedure:

```
CREATE PROCEDURE ChooseResultType( @resultType INT )
AS BEGIN
    IF @resultType = 1
        SELECT * FROM [Customers]
    ELSE IF @resultType = 2
        SELECT * FROM [Products]
END
```

Such a stored procedure will always return a single result, but its type might be different on each call. We do not like this use of stored procedures and prefer to avoid this situation. However, if you have to handle this case, by decorating the method with both possible *ResultType* attributes, you can handle both situations:

```
[Function(Name = "ChooseResultType", IsComposable = false)]
[ResultType(typeof(Customer))]
[ResultType(typeof(Product))]
public IMultipleResults ChooseResultType( int resultType ) {
    IExecuteResult executeResult =
            this.ExecuteMethodCall(
                 this,
                 (MethodInfo) (MethodInfo.GetCurrentMethod()),
                 resultType );
    IMultipleResults result =
        (IMultipleResults) executeResult.ReturnValue;
    return result;
}
```

In the single *GetResult<T>* call, you have to specify the type that correctly corresponds to what the stored procedure will return:

```
IMultipleResults results = db.ChooseResultType( 1 );
foreach( var row in results.GetResult<Customer>()) {
    // Access to Customer instance
}
```

If you have a similar scenario, it would be better to encapsulate the stored procedure call (*ChooseResultType* in this case) in several methods, one for each possible returned type. This way, you limit the risk of mismatching the relationship between parameter and result type:

```
public IEnumerable<Customer> ChooseCustomer() {
    IMultipleResults results = db.ChooseResultType( 1 );
    return results.GetResult<Customer>();
}

public IEnumerable<Product> ChooseProduct() {
    IMultipleResults results = db.ChooseResultType( 2 );
    return results.GetResult<Product>();
}
```

Before turning to user-defined functions, it is worth taking a look at what happens when you call a stored procedure in a LINQ query. Consider the following code:

```
var query =
    from   c in db.CustomersByCity("London")
    where  c.CompanyName.Length > 15
    select new { c.CustomerID, c.CompanyName };
```

Apparently, this query can be completely converted into a SQL query. However, all the data returned from *CustomersByCity* is passed from the SQL server to the client, as you can see from the generated SQL statement:

```
EXEC @RETURN_VALUE = [Customers by City] @param1 = 'London'
```

Both the filter (*where*) and projection (*select*) operations are made by LINQ to Objects, filtering data that has been transmitted to the client and enumerating only rows that have a *CompanyName* value longer than 15 characters. Thus, stored procedures are not composable into a single SQL query. To make this kind of composition, you need to use user-defined functions.

User-Defined Functions

To be used in LINQ, a user-defined function needs the same kind of declaration as a stored procedure. When you use a UDF inside a LINQ query, the LINQ to SQL engine must consider it in the construction of the SQL statement, adding a UDF call to the generated SQL. The capability of a UDF to be used in a LINQ query is what we mean by composability—the capability to compose different queries and/or operators into a single query. Because the same *Function* attribute is used for both stored procedures and UDFs, the *IsComposable* argument is set to *true* to map a UDF, and is set to *false* to map a stored procedure. Remember that there is no difference between a UDF written in T-SQL or SQLCLR.

Listing 5-14 provides an example of a LINQ declaration of the scalar-valued UDF *MinUnitPriceByCategory* that is defined in the sample Northwind database.

LISTING 5-14 Scalar-valued UDF

```
class SampleDb : DataContext {
    // ...
    [Function(Name = "dbo.MinUnitPriceByCategory", IsComposable = true)]
    public decimal? MinUnitPriceByCategory( int? categoryID) {
        IExecuteResult executeResult =
            this.ExecuteMethodCall(
                this,
                ((MethodInfo) (MethodInfo.GetCurrentMethod())),
                categoryID);
        decimal? result = (decimal?) executeResult.ReturnValue;
        return result;
    }
}
```

The call to a UDF as an isolated expression generates a single SQL query invocation. You can also use a UDF in a LINQ query such as the following:

```
var query =
    from   c in Categories
    select new { c.CategoryID,
                 c.CategoryName,
                 MinPrice = db.MinUnitPriceByCategory( c.CategoryID )};
```

The generated SQL statement *composes* the LINQ query with the UDF that is called, resulting in a SQL query like this:

```
SELECT [t0].[CategoryID],
       [t0].[CategoryName],
       dbo.MinUnitPriceByCategory([t0].[CategoryID]) AS [value]
FROM   [Categories] AS [t0]
```

There are some differences in table-valued UDF wrappers. Consider the following UDF:

```
CREATE FUNCTION [dbo].[CustomersByCountry] ( @country NVARCHAR(15) )
RETURNS TABLE
AS RETURN
    SELECT  CustomerID,
            ContactName,
            CompanyName,
            City
    FROM    Customers c
    WHERE   c.Country = @country
```

To use this UDF in LINQ, you need to declare a *CustomersByCountry* method, as shown in Listing 5-15. A table-valued UDF always sets *IsComposable* to *true* in *Function* arguments, but it calls the *DataContext.CreateMethodCallQuery* instead of *DataContext.ExecuteMethodCall*.

LISTING 5-15 Table-valued UDF

```
class SampleDb : DataContext {
    // ...
    [Function(Name = "dbo.CustomersByCountry", IsComposable = true)]
    public IQueryable<Customer> CustomersByCountry(string country) {
        return this.CreateMethodCallQuery<Customer>(
            this,
            ((MethodInfo) (MethodInfo.GetCurrentMethod())),
            country);
    }
}
```

A table-valued UDF can be used like any other table in a LINQ query. For example, you can join customers returned by the previous UDF with the orders they placed, as in the following query:

```
Table<Order> Orders = db.GetTable<Order>();
var queryCustomers =
    from   c in db.CustomersByCountry( "USA" )
    join   o in Orders
           on c.CustomerID equals o.CustomerID
           into orders
    select new { c.CustomerID, c.CompanyName, orders };
```

The generated SQL query will be similar to this one:

```
SELECT [t0].[CustomerID], [t0].[CompanyName],
       [t1].[OrderID], [t1].[CustomerID] AS [CustomerID2],
       (SELECT COUNT(*)
        FROM [Orders] AS [t2]
        WHERE [t0].[CustomerID] = [t2].[CustomerID]
        ) AS [value]
FROM dbo.CustomersByCountry('USA') AS [t0]
LEFT OUTER JOIN [Orders] AS [t1] ON [t0].[CustomerID] = [t1].[CustomerID]
ORDER BY [t1].[OrderID]
```

Compiled Queries

If you need to repeat the same query many times, eventually with different argument values, you might be worried about the multiple query construction. Several databases, such as SQL Server, try to parameterize received SQL queries automatically to optimize the compilation of the query execution plan. However, the program that sends a parameterized query to SQL Server will get better performance because SQL Server does not have to spend time analyzing it if the query is similar to one already processed. LINQ already does a fine job of query optimization, but each time that the same query tree is evaluated, the LINQ to SQL engine parses the query tree to build the equivalent SQL code. You can optimize this behavior by using the *CompiledQuery* class.

More Info The built-in SQL Server provider sends parameterized queries to the database. Every time you see a constant value in the SQL code presented in this chapter, keep in mind that the real SQL query sent to the database has a parameter for each constant in the query. That constant can be the result of an expression that is independent of the query execution. This kind of expression is resolved by the host language (C# in this case). When you use the *CompiledQuery* class, it eliminates the need to parse the query tree and create the equivalent SQL code every time LINQ processes the same query. You might ask: What is the break-even point that justifies the use of the *CompiledQuery* class? Rico Mariani did a performance test that is described in a blog post at *http://blogs.msdn.com/b/ricom/archive/2008/01/14/performance-quiz-13-linq-to-sql-compiled-query-cost-solution.aspx*. The response from his benchmark is that, with at least two calls for the query, the use of the *CompiledQuery* class produces a performance advantage.

To compile a query, you can use one of the *CompiledQuery.Compile* static methods. This approach passes the LINQ query as a parameter in the form of an expression tree, and then obtains a delegate with arguments corresponding to both the *DataContext* on which you want to operate and the parameters of the query. Listing 5-16 illustrates the compiled query declaration and use.

LISTING 5-16 Compiled query in a local scope

```
static void CompiledQueriesLocal() {
    DataContext db = new DataContext( ConnectionString );
    Table<Customer> Customers = db.GetTable<Customer>();

    var query =
        CompiledQuery.Compile(
            ( DataContext context, string filterCountry ) =>
                from   c in Customers
                where  c.Country == filterCountry
                select new { c.CustomerID, c.CompanyName, c.City } );

    foreach (var row in query( db, "USA" )) {
        Console.WriteLine( row );
    }

    foreach (var row in query( db, "Italy" )) {
        Console.WriteLine( row );
    }
}
```

As you can see in Listing 5-16, the *Compile* method requires a lambda expression whose first argument is a *DataContext* instance. That argument defines the connection over which the query will be executed. In this case, we do not use that argument inside our lambda expression. Assigning the *CompiledQuery.Compile* result to a local variable is easy (because you declare that variable with *var*), but you will not encounter this situation very frequently. Chances are that you will need to store the delegate returned from *CompiledQuery.Compile* in an instance or a static member to easily reuse it several times. To do that, you need to know the correct declaration syntax.

A compiled query is stored in a *Func* delegate, where the first argument must be an instance of *DataContext* (or a class derived from *DataContext*) and the last argument must be the type returned from the query. You can define up to three arguments in the middle that will be arguments of the compiled query. You will need to specify these arguments for each compiled query invocation. Listing 5-17 shows the syntax you can use in this scenario to create the compiled query and then use it.

LISTING 5-17 Compiled query assigned to a static member

```
public static Func< SampleDb, string, IQueryable<Customer>>
    CustomerByCountry =
        CompiledQuery.Compile(
             ( nwind.Northwind db, string filterCountry ) =>
                from   c in db.Customers
                where  c.Country == filterCountry
                select c );

static void CompiledQueriesStatic() {
    nwind.Northwind db = new nwind.Northwind( ConnectionString );

    foreach (var row in CustomerByCountry( db, "USA" )) {
        Console.WriteLine( row.CustomerID );
    }

    foreach (var row in CustomerByCountry( db, "Italy" )) {
        Console.WriteLine( row.CustomerID );
    }
}
```

Because the *Func* delegate that holds the compiled query needs the result type in its declaration, you cannot use an anonymous type as the result type of a compiled query. This is possible only when the compiled query is stored in a local variable, as you saw in Listing 5-16.

Different Approaches to Querying Data

When using LINQ to SQL entities, you have two approaches for querying the same data. The classic way to navigate a relational schema is to write associative queries, just as you can do in SQL. The alternative way offered by LINQ to SQL is through graph traversal. Given the same query result, you might obtain different SQL queries and a different level of performance using different LINQ approaches.

Consider this SQL query that calculates the total quantity of orders for a product (in this case, Chocolade, which is a localized name in the Northwind database):

```
SELECT    SUM( od.Quantity ) AS TotalQuantity
FROM      [Products] p
LEFT JOIN [Order Details] od
     ON   od.[ProductID] = p.[ProductID]
WHERE     p.ProductName = 'Chocolade'
```

The natural conversion into a LINQ query is shown in Listing 5-18. The *Single* operator gets the first row and puts it into *quantityJoin*, which is used to display the result.

LISTING 5-18 Query using *Join*

```
var queryJoin =
    from   p in db.Products
    join   o in db.Order_Details
           on p.ProductID equals o.ProductID
           into OrdersProduct
    where  p.ProductName == "Chocolade"
    select OrdersProduct.Sum( o => o.Quantity );
var quantityJoin = queryJoin.Single();
Console.WriteLine( quantityJoin );
```

As you can see, the associative query in LINQ can explicitly require the join between *Products* and *Order_Details* through *ProductID* equivalency. By using entities, you can implicitly use the relationship between *Products* and *Order_Details* defined in the *Product* class, as shown in Listing 5-19.

LISTING 5-19 Query using *Association*

```
var queryAssociation =
    from   p in db.Products
    where  p.ProductName == "Chocolade"
    select p.Order_Details.Sum( o => o.Quantity );
var quantityAssociation = queryAssociation.Single();
Console.WriteLine( quantityAssociation );
```

The single SQL queries produced by both of these LINQ queries are identical. The LINQ query with *join* is more explicit about the access to data, whereas the query that uses the association between *Product* and *Order_Details* is more implicit in this regard. Using implicit associations results in shorter queries that are less error-prone (because you cannot be wrong about the join condition). At first, you might find that a shorter query is harder to read; that might be because you are accustomed to seeing lengthier queries. Your comfort level with shorter ones might change over time.

Note The SQL query produced by the LINQ queries in Listings 5-18 and 5-19 is different between SQL Server 2000 and SQL Server 2005 or later versions. With SQL Server 2005, the OUTER APPLY join is used. This is the result of an internal implementation of the provider, but the final result is the same.

Examining this further, you can observe that reading a single product does not require a query expression. You can apply the *Single* operator directly on the Products table, as shown in Listing 5-20. Although the results are the same, the internal process is much different because this kind of access generates instances of the *Product* and *Order_Details* entities in memory, even if you do not use them in your program.

LISTING 5-20 Access through *Entity*

```
var chocolade = db.Products.Single( p => p.ProductName == "Chocolade" );
var quantityValue = chocolade.Order_Details.Sum( o => o.Quantity );
Console.WriteLine( quantityValue );
```

This is a two-step operation that sends two SQL queries to the database. The first one retrieves the *Product* entity. The second one accesses the Order Details table to get *all* the Order Details rows for the required product and sums up the *Quantity* value in memory for the required product. The operation generates the following SQL statements:

```
SELECT [t0].[ProductID], [t0].[ProductName], [t0].[SupplierID],
       [t0].[CategoryID], [t0].[QuantityPerUnit], [t0].[UnitPrice],
       [t0].[UnitsInStock], [t0].[UnitsOnOrder], [t0].[ReorderLevel],
       [t0].[Discontinued]
FROM   [dbo].[Products] AS [t0]
WHERE  [t0].[ProductName] = "Chocolade"

SELECT [t0].[OrderID], [t0].[ProductID], [t0].[UnitPrice], [t0].[Quantity],
       [t0].[Discount]
FROM   [dbo].[Order Details] AS [t0]
WHERE  [t0].[ProductID] = "Chocolade"
```

Code that uses this kind of access is shorter to write compared to a query, but its performance is worse if you need to get only the total *Quantity* value, without needing to retrieve *Product* and *Order_Detail* entities in memory for further operations.

The queries in Listings 5-18 and 5-19 did not create *Product* or *Order_Details* instances because the output required only the product total. From this point of view, if you already had the required *Product* and *Order_Details* instances for Chocolade in memory, the performance of those queries would be worse because they unnecessarily access the database to get data that is already in memory. On the other hand, a second access to get the sum *Quantity* could be faster if you use the entity approach. Consider this code:

```
var chocolade = db.Products.Single( p => p.ProductName == "Chocolade" );
var quantityValue = chocolade.Order_Details.Sum( o => o.Quantity );
Console.WriteLine( quantityValue );
var repeatCalc = chocolade.Order_Details.Sum( o => o.Quantity );
Console.WriteLine( repeatCalc );
```

The *quantityValue* evaluation requires a database query to create *Order_Details* entities, whereas the *repeatCalc* evaluation is made on the in-memory entities without the need to read other data from SQL Server.

Note A good way to understand how your code behaves is to analyze the SQL queries that are produced. In the previous examples, we wrote a *Sum* in a LINQ query. When the generated SQL query contains a SUM aggregation operation, you are not reading entities in memory; however, when the generated SQL query does not contain the requested aggregation operation, that aggregation will be made in memory on corresponding entities.

A final thought on the number of generated queries: You might think that we generated two queries when accessing data through the *Product* entity because we had two distinct statements—one to assign the *chocolade* variable, and the other to assign a value to *quantityEntity*. This assumption is not completely true. Even if you write a single statement, the use of a *Product* entity (the results from the *Single* operator call) generates a separate query. Listing 5-21 produces the same results (in terms of memory objects and SQL queries) as Listing 5-20.

LISTING 5-21 Access through *Entity* with a single statement

```
var quantityChocolade = db.Products.Single( p => p.ProductName == "Chang" )
                        .Order_Details.Sum( o => o.Quantity );
Console.WriteLine( quantityChocolade );
```

Finding a better way to access data really depends on the entire set of operations performed by a program. If you extensively use entities in your code to store data in memory, access to data through graph traversal based on entity access might offer better performance. On the other hand, if you always transform query results in anonymous types and never manipulate entities in memory, you might prefer an approach based on LINQ queries. As usual, the right answer is, "It depends."

Direct Queries

Sometimes you might need access to database SQL features that are not available with LINQ. For example, imagine that you want to use Common Table Expressions (CTEs) or the PIVOT command with SQL Server. LINQ does not have an explicit constructor to do that, even if its SQL Server provider could use these features to optimize some queries. In such cases, you can use the *ExecuteQuery<T>* method of the *DataContext* class to send a query directly to the database. Listing 5-22 shows an example. (The *T* in *ExecuteQuery<T>* is an entity class that represents a returned row.)

LISTING 5-22 Direct query

```
var query = db.ExecuteQuery<EmployeeInfo>( @"
    WITH EmployeeHierarchy (EmployeeID, LastName, FirstName,
                            ReportsTo, HierarchyLevel) AS
     ( SELECT EmployeeID,LastName, FirstName,
              ReportsTo, 1 as HierarchyLevel
       FROM   Employees
       WHERE  ReportsTo IS NULL

       UNION ALL

       SELECT      e.EmployeeID, e.LastName, e.FirstName,
                   e.ReportsTo, eh.HierarchyLevel + 1 AS HierarchyLevel
       FROM        Employees e
       INNER JOIN  EmployeeHierarchy eh
               ON  e.ReportsTo = eh.EmployeeID
    )
    SELECT   *
    FROM     EmployeeHierarchy
    ORDER BY HierarchyLevel, LastName, FirstName" );
```

As you can see, you need a type to get direct query results. We used the *EmployeeInfo* type in this example, which is declared as follows:

```
public class EmployeeInfo {
    public int EmployeeID;
    public string LastName;
    public string FirstName;
    public int? ReportsTo; // int? Corresponds to Nullable<int>
    public int HierarchyLevel;
}
```

The names and types of *EmployeeInfo* members must match the names and types of the columns returned by the executed query. Please note that if a column can return a NULL value, you need to use a nullable type, as we did for the *ReportsTo* member declared as *int?* above (which corresponds to *Nullable<int>*).

Warning Columns in the resulting rows that do not match entity attributes are ignored. Entity members that do not have corresponding columns are initialized with the default value. If the *EmployeeInfo* class contains a mismatched column name, that member will not be assigned without an error. Be sure to check name correspondence in the result if you find missing column or member values.

The *ExecuteQuery* method can receive parameters using the same parameter placeholders notation (also known as *curly notation*) used by *Console.WriteLine* and *String.Format*, but with a different behavior. Parameters are not replaced in the string sent to the database; they are substituted with automatically generated parameter names such as (*@p0*, *@p1*, *@p2*, ...) and are sent to SQL Server as arguments of the parametric query.

The code in Listing 5-23 shows the call to *ExecuteQuery<T>* using a SQL statement with two parameters. The parameters are used to filter the customers who made their first order within a specified range of dates.

LISTING 5-23 Direct query with parameters

```
var query = db.ExecuteQuery<CompanyOrders>(@"
        SELECT    c.CompanyName,
                  MIN( o.OrderDate ) AS FirstOrderDate,
                  MAX( o.OrderDate ) AS LastOrderDate
        FROM      Customers c
        LEFT JOIN Orders o
               ON o.CustomerID = c.CustomerID
        GROUP BY  c.CustomerID, c.CompanyName
        HAVING    COUNT(o.OrderDate) > 0
           AND    MIN( o.OrderDate ) BETWEEN {0} AND {1}
        ORDER BY  FirstOrderDate ASC",
    new DateTime( 1997, 1, 1 ),
    new DateTime( 1997, 12, 31 ) );
```

The parameters in the preceding query are identified by the *{0}* and *{1}* format items. The generated SQL query simply substitutes them with *@p0* and *@p1*. The results are returned in instances of the *CompanyOrders* class, declared as follows:

```
public class CompanyOrders {
    public string CompanyName;
    public DateTime FirstOrderDate;
    public DateTime LastOrderDate;
}
```

Deferred Loading of Entities

You have seen that using graph traversal to query data is a very comfortable way to proceed. However, sometimes you might want to stop the LINQ to SQL provider from automatically deciding what entities have to be read from the database and when, thereby taking control over that part of the process. You can do this by using the *DeferredLoadingEnabled* and *LoadOptions* properties of the *DataContext* class.

The code in Listing 5-24 makes the same *QueryOrder* call under three different conditions, driven by the code in the *DemoDeferredLoading* method.

LISTING 5-24 Deferred loading of entities

```
public static void DemoDeferredLoading() {
    Console.Write("DeferredLoadingEnabled=true  ");
    DemoDeferredLoading(true);
    Console.Write("DeferredLoadingEnabled=false ");
    DemoDeferredLoading(false);
    Console.Write("Using LoadOptions            ");
    DemoLoadWith();
}

static void DemoDeferredLoading(bool deferredLoadingEnabled) {
    nwDataContext db = new nwDataContext(Connections.ConnectionString);
    db.DeferredLoadingEnabled = deferredLoadingEnabled;

    QueryOrder(db);
}

static void DemoLoadWith() {
    nwDataContext db = new nwDataContext(Connections.ConnectionString);
    db.DeferredLoadingEnabled = false;

    DataLoadOptions loadOptions = new DataLoadOptions();
    loadOptions.LoadWith<Order>(o => o.Order_Details);
    db.LoadOptions = loadOptions;

    QueryOrder(db);
}

static void QueryOrder(nwDataContext db) {
    var order = db.Orders.Single((o) => o.OrderID == 10251);
    var orderValue = order.Order_Details.Sum(od => od.Quantity * od.UnitPrice);
    Console.WriteLine(orderValue);
}
```

The call to *DemoDeferredLoading(true)* sets the *DeferredLoadingEnabled* property to *true*, which is the default condition for a *DataContext* instance. The call to *DemoDeferredLoading(false)* disables the *DeferredLoadingEnabled* property. Any access to the related entities does not automatically load data from the database, and the sum of *Order_Details* entities shows a total of 0. Finally, the call to *DemoLoadWith* also disables *DeferredLoadingEnabled*, but it sets the *LoadOptions* property of the *DataContext*, requesting the loading of *Order_Details* entities related to an *Order* instance. The execution of the *DemoDeferredLoading* method in Listing 5-24 produces the following output:

```
DeferredLoadingEnabled=true  670,8000
DeferredLoadingEnabled=false 0
Using LoadOptions            670,8000
```

Remember that the use of *LoadOptions* is possible regardless of the state of *DeferredLoadingEnabled*, and it is useful for improving performance when early loading of related entities (rather than deferred loading) is an advantage for your application. Consider carefully before using *DeferredLoadingEnabled*—it does not produce any error, but it limits the navigability of your data model through graph traversal. However, you must remember that *DeferredLoadingEnabled* is automatically considered to be *false* whenever the *ObjectTrackingEnabled* property (discussed in the next section) is disabled too.

Deferred Loading of Properties

LINQ to SQL provides a deferred loading mechanism that acts at the property level, loading data only when that property is accessed for the first time. You can use this mechanism when you need to load a large number of entities in memory, which usually requires space to accommodate all the properties of the class that correspond to table columns of the database. If a certain field is very large and is not always accessed for every entity, you can delay the loading of that property.

To request the deferred loading of a property, you simply use the *Link<T>* type to declare the storage variable for the table column, as you can see in Listing 5-25.

LISTING 5-25 Deferred loading of properties

```
[Table(Name = "Customers")]
public class DelayCustomer {
    private Link<string> _Address;

    [Column(IsPrimaryKey = true)] public string CustomerID;
    [Column] public string CompanyName;
    [Column] public string Country;

    [Column(Storage = "_Address")]
    public string Address {
        get { return _Address.Value; }
        set { _Address.Value = value; }
    }
}

public static class DeferredLoading {
    public static void DelayLoadProperty() {
        DataContext db = new DataContext(Connections.ConnectionString);
        Table<DelayCustomer> Customers = db.GetTable<DelayCustomer>();
        db.Log = Console.Out;

        var query =
            from   c in Customers
            where  c.Country == "Italy"
            select c;
```

```
        foreach (var row in query) {
            Console.WriteLine(
                "{0} - {1}",
                row.CompanyName,
                row.Address);
        }
    }
}
```

The query that is sent to the database to get the list of Italian customers is functionally equivalent to the following one:

```
SELECT [t0].[CustomerID], [t0].[CompanyName], [t0].[Country]
FROM   [Customers] AS [t0]
WHERE  [t0].[Country] = "Italy"
```

This query does not retrieve the *Address* field. When the result of the query is iterated in the *foreach* loop, the *Address* property of the current *Customer* is accessed for each customer for the first time. This produces a query to the database like the following one to get the *Address* value:

```
SELECT [t0].[Address]
FROM   [Customers] AS [t0]
WHERE  [t0].[CustomerID] = @p0
```

You should use the *Link<T>* type only when the content of a field is very large (which should *not* be the case for the *Address* field example) or when that field is rarely accessed. A field defined with the SQL type *VARCHAR(MAX)* is generally a good candidate, as long as its value is displayed only in a detailed form visible on demand and not on the main grid that shows query results. Using the LINQ to SQL class generator included in Visual Studio, you can use *Link<T>* and set the *Delay Loaded* property of the desired member property to *true*.

Important You need to use the *Link<T>* type on the storage variable for a property of type *T* mapped to the column, as shown in Listing 5-25. You cannot use the *Link<T>* type directly on a public data member mapped to a table column (like all the other fields); if you do, you will get an exception during execution. That run-time error is of type *VerificationException*. Future versions may have a more analytical exception.

Read-Only *DataContext* Access

If you need to access data exclusively as read-only, you might want to improve performance by disabling a *DataContext* service that supports data modification:

```
DataContext db = new DataContext( ConnectionString );
db.ObjectTrackingEnabled = false;
var query = ...
```

The *ObjectTrackingEnabled* property controls the change tracking service described in Chapter 6. By default, *ObjectTrackingEnabled* is set to *true*.

Important Disabling object tracking also disables the deferred loading feature of the same *DataContext* instance. If you want to optimize performance by disabling the object tracking feature, you must be aware of the side effects of disabling deferred loading too. Refer to the "Deferred Loading of Entities" section earlier in this chapter for further details.

Limitations of LINQ to SQL

LINQ to SQL has some limitations when converting a LINQ query to a corresponding SQL statement. For this reason, some valid LINQ to Objects statements are not supported in LINQ to SQL. In this section, we cover the most important operators that you cannot use in a LINQ to SQL query. However, you can use specific T-SQL commands by using the extension methods defined in the *SqlMethods* class, which you will find in the *System.Data.Linq.SqlClient* namespace.

More Info A complete list of unsupported methods and types is available on the "Data Types and Functions (LINQ to SQL)" page of the product documentation, available at *http://msdn.microsoft.com/en-us/library/bb386970.aspx*.

Aggregate Operators

The general-purpose *Aggregate* operator is not supported. However, specialized aggregate operators such as *Count*, *LongCount*, *Sum*, *Min*, *Max*, and *Average* are fully supported.

Any aggregate operator other than *Count* and *LongCount* requires particular care to avoid an exception if the result is *null*. If the entity class has a member of a nonnullable type and you make an aggregation on it, a null result (for example when no rows are aggregated) throws an exception. To avoid the exception, you should cast the aggregated value to a nullable type before considering it in the aggregation function. Listing 5-26 shows an example of the necessary cast.

LISTING 5-26 Null handling with aggregate operators

```
decimal? totalFreight =
    (from   o in Orders
     where  o.CustomerID == "NOTEXIST"
     select o).Min( o => (decimal?) o.Freight );
```

This cast is necessary only if you declared the *Freight* property with *decimal*, as shown in the following code:

```
[Table(Name = "Orders")]
public class Order {
    [Column] public decimal Freight;
}
```

Another solution is to declare *Freight* as a nullable type, using *decimal?*—but it is not a good idea to have different nullable settings between entities and corresponding tables in the database.

More Info You can find a more complete discussion about this issue in the post "LINQ to SQL, Aggregates, EntitySet, and Quantum Mechanics," written by Ian Griffiths and located at *http://www.interact-sw.co.uk/iangblog/2007/09/10/linq-aggregates*.

Partitioning Operators

The *TakeWhile* and *SkipWhile* operators are not supported. *Take* and *Skip* operators are supported, but be careful with *Skip* because the generated SQL query could be complex and not very efficient when skipping a large number of rows, especially when the target database is SQL Server 2000.

Element Operators

The following operators are not supported: *ElementAt*, *ElementAtOrDefault*, *Last*, and *LastOrDefault*.

String Methods

Many of the .NET Framework *String* type methods are supported in LINQ to SQL because T-SQL has a corresponding method. However, there is no support for methods that are culture-aware (those that receive arguments of type *CultureInfo*, *StringComparison*, and *IFormatProvider*) and for methods that receive or return a *char* array.

DateTime Methods

The *DateTime* type in the .NET Framework is different than the *DATETIME* and *SMALLDATETIME* types in SQL Server. The range of values and the precision is greater in the .NET Framework than in SQL Server, meaning the .NET Framework can correctly represent SQL Server types, but not the opposite. Check out the *SqlMethods* extension methods, which can take advantage of several *DateDiff* functions.

LIKE Operator

Although the *LIKE* T-SQL operator is used whenever a *StartsWith*, *EndsWith*, or *Contains* operator is called on a string property, you can use *LIKE* directly by calling the *SqlMethods.Like* method in a predicate.

Unsupported SQL Functionalities

LINQ to SQL does not have syntax to make use of the *STDDEV* aggregation.

Thinking in LINQ to SQL

When you start working with LINQ to SQL, you might have to rethink the ways in which you are accustomed to writing queries, especially if you try to find the equivalent LINQ syntax for a well-known SQL statement. Moreover, a verbose LINQ query might be reduced when the corresponding SQL query is produced. You need to be aware of this change, and you have to fully understand it to be productive in LINQ to SQL. The final part of this chapter introduces you to thinking in LINQ to SQL.

The *IN/EXISTS* Clause

One of the best examples of the syntactic differences between T-SQL and LINQ is the *NOT IN* clause that you can use in SQL. LINQ does not have such a clause, which makes you wonder whether there is any way to express the same concept in LINQ. In fact, there is not always a direct translation for each single SQL keyword, but you can get the same result with semantically equivalent statements, sometimes with equal or better performance.

Consider this SQL query, which returns all the customers who do not have an order in the Orders table:

```
SELECT *
FROM   [dbo].[Customers] AS [t0]
WHERE  [t0].[CustomerID] NOT IN (
    SELECT [t1].[CustomerID]
    FROM   [dbo].[Orders] AS [t1]
)
```

This is not the fastest way to get the desired result. (Using *NOT EXISTS* is our favorite way—more on this shortly.) LINQ does not have an operator directly equivalent to *IN* or *NOT IN*, but it offers a *Contains* operator that you can use to write the code in Listing 5-27. Pay attention to the *not* operator (*!*) applied to the *where* predicate, which negates the *Contains* condition that follows.

LISTING 5-27 Use of *Contains* to get an *EXISTS/IN* equivalent statement

```
public static void DemoContains() {
    nwDataContext db = new nwDataContext(Connections.ConnectionString);
    db.Log = Console.Out;

    var query =
        from c in db.Customers
        where !(from o in db.Orders
                select o.CustomerID)
               .Contains(c.CustomerID)
        select new { c.CustomerID, c.CompanyName };

    foreach (var c in query) {
        Console.WriteLine(c);
    }
}
```

The following code is the SQL query generated by LINQ to SQL:

```
SELECT [t0].[CustomerID], [t0].[CompanyName]
FROM   [dbo].[Customers] AS [t0]
WHERE  NOT (EXISTS(
    SELECT NULL AS [EMPTY]
    FROM   [dbo].[Orders] AS [t1]
    WHERE  [t1].[CustomerID] = [t0].[CustomerID]
    ))
```

Using this approach to generate SQL code is not only semantically equivalent, but it also executes faster. If you look at the input/output (I/O) operation made by SQL Server 2005, the first query (using *NOT IN*) executes 364 logical reads on the Orders table, whereas the second query (using *NOT EXISTS*) requests only 5 logical reads on the same Orders table. That is a big difference. In this case, LINQ to SQL is the best choice.

The same *Contains* operator might generate an *IN* operator in SQL, for example, if it is applied to a list of constants, as in Listing 5-28.

LISTING 5-28 Use of *Contains* with a list of constants

```
public static void DemoContainsConstants() {
    nwDataContext db = new nwDataContext(Connections.ConnectionString);

    var query =
        from   c in db.Customers
        where  (new string[] { "London", "Seattle" }).Contains(c.City)
        select new { c.CustomerID, c.CompanyName, c.City };

    Console.WriteLine(query);

    foreach (var c in query) {
        Console.WriteLine(c);
    }
}
```

The SQL code generated by LINQ to SQL is simpler to read than the original query:

```
SELECT [t0].[CustomerID], [t0].[CompanyName], [t0].[City]
FROM   [dbo].[Customers] AS [t0]
WHERE  [t0].[City] IN ("London", "Seattle")
```

The LINQ query is counterintuitive in that you must specify the *Contains* operator on the list of constants, passing the value to look for as an argument—exactly the opposite of what you need to do in SQL:

```
where (new string[] { "London", "Seattle" }).Contains(c.City)
```

After years of experience in SQL, it is more comfortable to imagine this hypothetical *IsIn* syntax:

```
where c.City.IsIn( new string[] { "London", "Seattle" } )
```

However, it is probably only a question of time before you get used to the new syntax. In fact, the semantics of *Contains* corresponds exactly to the argument's position. To make the code clearer, you could simply declare the list of constants outside the query declaration, in a *cities* array, for example:

```
var cities = new string[] { "London", "Seattle" };
var query =
    from   c in db.Customers
    where  cities.Contains(c.City)
    select new { c.CustomerID, c.CompanyName, c.City };
```

Note Creating the *cities* array outside the query instead of putting it in the *where* predicate simply improves code readability, at least in LINQ to SQL. From a performance point of view, only one *string* array is created in both cases. The reason is that in LINQ to SQL, the query defines only an expression tree, and the array is created only once to produce the SQL statement. In LINQ to SQL, unless you execute the same query many times, performance is equivalent under either approach (object creation inside or outside a predicate). This is different in LINQ to Objects, in which the predicate condition in the *where* clause would be executed for each row of the data source.

SQL Query Reduction

Every LINQ to SQL query is initially represented in memory as an expression tree. The LINQ to SQL engine converts this tree into an equivalent SQL query, visiting the tree and generating the corresponding code. However, theoretically this translation can be made in many ways, all producing the same results, even if not all the translations are equally readable or perform as well. The actual implementation of LINQ to SQL generates good SQL code, favoring performance over query readability, although the readability of the generated code is often quite acceptable.

More Info You can find more information about query reduction in a LINQ provider in the following post from Matt Warren: "LINQ: Building an IQueryable Provider - Part IX," located at *http://blogs.msdn.com/mattwar/archive/2008/01/16/linq-building-an-iqueryable-provider-part-ix.aspx*. Implementation of a query provider is covered in Chapter 15, "Extending LINQ."

We described this quality of LINQ to SQL because it is important to know that unnecessary parts of the query are removed before the query is sent to SQL Server. You can use this knowledge to compose LINQ queries in many ways—for example, by appending new predicates and projections to an originally large selection of rows and columns, without worrying too much about unnecessary elements left in the query.

The LINQ query in Listing 5-29 first makes a query on Customers, which filters those customers with a *CompanyName* longer than 10 characters. Those companies are then filtered by *Country*, operating on the anonymous type generated by the inner query.

LISTING 5-29 Example of query reduction

```
var query =
    from s in (
        from   c in db.Customers
        where  c.CompanyName.Length > 10
        select new { c.CustomerID, c.CompanyName, c.ContactName, c.City,
                     c.Country, c.ContactTitle, c.Address }
    )
    where s.Country == "UK"
    select new { s.CustomerID, s.CompanyName, s.City };
```

Despite the length of the LINQ query, here is the SQL query it generates:

```
SELECT [t0].[CustomerID], [t0].[CompanyName], [t0].[City]
FROM   [dbo].[Customers] AS [t0]
WHERE  ([t0].[Country] = @p0) AND (LEN([t0].[CompanyName]) > @p1)
```

The generated SQL query made two important reductions. First, the *FROM* operates on a single table instead of a *SELECT … FROM (SELECT … FROM …)* composition that would normally be made when translating the original query tree. Second, unnecessary fields have been removed; only *CustomerID*, *CompanyName*, and *City* are part of the *SELECT* projection because they are the only fields necessary to the consumer of the LINQ query. The first reduction improves query readability; the second improves performance because it reduces the amount of data transferred from the database server to the client.

Mixing .NET Code with SQL Queries

As noted previously, LINQ to SQL has some known limitations with regard to using the full range of the .NET Framework features, not all of which can be entirely translated into corresponding T-SQL operations. This does not necessarily mean that you *cannot* write a query containing an unsupported method, but you should be aware that such a method cannot be translated into T-SQL and must be executed locally on the client. The side effect of this can be that sections of the query tree that depend on a .NET Framework method without a corresponding SQL translation will be executed completely as a LINQ to Objects operation, meaning that all the data must be transferred to the client to apply the required operators.

You can see this effect with some examples. Consider the LINQ query in Listing 5-30.

LISTING 5-30 LINQ query with a native string manipulation in the projection

```
var query1 =
    from   p in db.Products
    where  p.UnitPrice > 50
    select new {
        ProductName = "** " + p.ProductName + " **",
        p.UnitPrice };
```

The generated SQL query embodies the string manipulation of the *ProductName*:

```
SELECT ("** " + [t0].[ProductName]) + " **" AS [ProductName],
       [t0].[UnitPrice]
FROM [dbo].[Products] AS [t0]
WHERE [t0].[UnitPrice] > 50
```

Now suppose you move the string concatenation operation into a .NET Framework extension method, like that shown in Listing 5-31.

LISTING 5-31 String manipulation extension method

```
static public class Extensions {
    public static string Highlight(this string s) {
        return "** " + s + " **";
    }
}
```

Then you can modify the LINQ query using the *Highlight* method as in Listing 5-32.

LISTING 5-32 LINQ query calling a .NET Framework method in the projection

```
var query2 =
    from   p in db.Products
    where  p.UnitPrice > 50
    select new {
        ProductName = p.ProductName.Highlight(),
        p.UnitPrice };
```

The result produced by *query2* in Listing 5-32 is the same as the one produced by *query1* in Listing 5-30. However, the SQL query sent to the database is different because it lacks the string manipulation operation:

```
SELECT [t0].[ProductName] AS [s],
       [t0].[UnitPrice]
FROM   [dbo].[Products] AS [t0]
WHERE  [t0].[UnitPrice] > 50
```

The *ProductName* field is returned as *s* and will be used as an argument to the *Highlight* call. For each row, a call to the .NET Framework *Highlight* method will be made. This is not an issue when you are directly consuming the *query2* results. However, if you turn the same operation into a subquery, the dependent queries cannot be translated into a native SQL statement. For example, consider *query3* in Listing 5-33.

LISTING 5-33 LINQ query combining native and custom string manipulation

```
var query3 =
    from a in (
        from   p in db.Products
        where  p.UnitPrice > 50
        select new {
            ProductName = p.ProductName.Highlight(),
            p.UnitsInStock,
            p.UnitPrice
        }
    )
    select new {
        ProductName = a.ProductName.ToLower(),
        a.UnitPrice };
```

The SQL query produced by *query3* in Listing 5-33 is the same as the one produced by *query2* in Listing 5-32, despite the addition of another string manipulation (*ToLower*) to *ProductName*:

```
SELECT [t0].[ProductName] AS [s],
       [t0].[UnitPrice]
FROM   [dbo].[Products] AS [t0]
WHERE  [t0].[UnitPrice] > 50
```

If you remove the call to *Highlight* and restore the original string manipulation directly inside the LINQ query, you will get a complete native SQL query again, as shown in Listing 5-34.

LISTING 5-34 LINQ query using native string manipulation

```
var query4 =
    from a in (
            from   p in db.Products
            where  p.UnitPrice > 50
            select new {
                ProductName = "** " + p.ProductName + " **",
                p.UnitPrice
            }
    )
    select new {
        ProductName = a.ProductName.ToLower(),
        a.UnitPrice
    };
```

The *query4* in Listing 5-34 produces the following SQL query, which does not require further manipulations by .NET Framework code:

```
SELECT LOWER([t1].[value]) AS [ProductName], [t1].[UnitPrice]
FROM (
    SELECT ("** " + [t0].[ProductName]) + " **" AS [value],
           [t0].[UnitPrice]
    FROM [dbo].[Products] AS [t0]
    ) AS [t1]
WHERE [t1].[UnitPrice] > 50
```

Until now, we have seen that there is a possible performance implication only when using a .NET Framework method that does not have a corresponding SQL counterpart. However, there are situations that cannot be handled by the LINQ to SQL engine and which throw an exception at execution time—for example, if you try to use the result of the *Highlight* call in a *where* predicate as shown in Listing 5-35.

LISTING 5-35 LINQ query calling a .NET Framework method in a *where* predicate

```
var query5 =
    from   p in db.Products
    where  p.ProductName.Highlight().Length > 20
    select new {
        ProductName = p.ProductName.Highlight(),
        p.UnitPrice
    };
```

At execution time, trying to access to the *query5* result (or asking for the generated SQL query) will raise the following exception:

```
System.NotSupportedException
Method 'System.String Highlight(System.String)'
has no supported translation to SQL.
```

As you have seen, it is important to understand what operators are supported by LINQ to SQL, because the code could work or break at execution time, depending on the use of such operators. It is hard to define a rule of thumb other than to avoid the use of unsupported operators. If you think that a LINQ query is composable and can be used as a source to build another query, the only safe guideline is to use operators supported by LINQ to SQL.

Summary

This chapter covered LINQ to SQL features used to query data. With LINQ to SQL, you can query a relational structure stored in a SQL Server database so that you can convert LINQ queries into native SQL queries and access UDFs and stored procedures if required. LINQ to SQL handles entity classes that map an underlying physical database structure through attributes or external XML files. Stored procedures and UDFs can be mapped to methods of a class representing a SQL Server database. LINQ to SQL supports most of the basic LINQ features that you saw in Chapter 3.

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X

Z

Paolo Pialorsi

Paolo Pialorsi is a consultant, trainer, and author who specializes in developing distributed applications architectures and Microsoft SharePoint enterprise solutions. He is a founder of DevLeap, a company focused on providing content and consulting to professional developers. Paolo wrote *Programming Microsoft LINQ* and *Introducing Microsoft LINQ* both published by Microsoft Press, and is the author of three books in Italian about XML and Web Services. He is also a regular speaker at industry conferences.

Marco Russo

Marco Russo is a founder of DevLeap. He is a regular contributor to developer user communities and is an avid blogger on Microsoft SQL Server Business Intelligence and other Microsoft technologies. Marco provides consulting and training to professional developers on the Microsoft .NET Framework and Microsoft SQL Server. He wrote *Programming Microsoft LINQ* and *Introducing Microsoft LINQ* with Paolo Pialorsi, *Expert Cube Development with Microsoft SQL Server 2008 Analysis Services* with Alberto Ferrari and Chris Webb, and is the author of two books in Italian about C# and the common language runtime.

What do you think of this book?

We want to hear from you!

To participate in a brief online survey, please visit:

microsoft.com/learning/booksurvey

Tell us how well this book meets your needs—what works effectively, and what we can do better. Your feedback will help us continually improve our books and learning resources for you.

Thank you in advance for your input!

Microsoft® Press

Stay in touch!

To subscribe to the *Microsoft Press® Book Connection Newsletter*—for news on upcoming books, events, and special offers—please visit:

microsoft.com/learning/books/newsletter