

Foreword by **Anders Hejlsberg**



Essential LINQ



Charlie Calvert
Dinesh Kulkarni

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The .NET logo is either a registered trademark or trademark of Microsoft Corporation in the United States and/or other countries and is used under license from Microsoft.

The authors and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside the United States please contact:

International Sales
international@pearson.com

Visit us on the Web: www.informit.com/msdotnetseries

Library of Congress Cataloging-in-Publication Data:

Calvert, Charles

Essential LINQ / Charlie Calvert, Dinesh Kulkarni. — 1st ed.

p. cm.

ISBN 0-321-56416-2 (pbk. : alk. paper) 1. Microsoft LINQ. 2. C# (Computer program language) 3. Query languages (Computer science) 4. Microsoft .NET Framework.

I. Kulkarni, Dinesh, 1968- II. Title.

QA76.7.C35 2009
006.7'882—dc22

2008052508

Copyright © 2009 Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, write to:

Pearson Education, Inc
Rights and Contracts Department
501 Boylston Street, Suite 900
Boston, MA 02116
Fax (617) 671 3447

ISBN-13: 978-0-321-56416-0

ISBN-10: 0-321-56416-2

Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, Indiana.

First printing March 2009



Foreword

For years I have been fascinated with the differences between general-purpose programming languages and databases. Practically every enterprise application built today is coded in a general-purpose programming language and talks to a database, yet the two ecosystems are amazingly different and quite poorly integrated—the impedance mismatch between object-oriented programming and the relational model is the gift that keeps on giving when it comes to application complexity.

But the thing I find particularly puzzling is the lack of query capabilities in general-purpose programming languages. Why is it you can query database tables but not in-memory objects? Why are XPath and XQuery so arbitrarily different from SQL? Why is it so hard to transform data between the object, relational, and XML domains? These are the kinds of questions that launched us on the Language Integrated Query (LINQ) journey. Along the way we got wise to the wonders of functional programming, lambda expressions, type inference, monads, O/R mapping, and all sorts of fascinating computer science. Fortunately, we managed to boil our learnings down to a set of pragmatic language features and APIs that are useful in practically any .NET application.

LINQ extends the .NET Framework and programming languages with a uniform model for querying and transforming in-memory collections, relational data, and XML documents. With LINQ, C# 3.0 and VB 9.0 gain the expressive power of SQL and XQuery to become the first general-purpose programming languages to natively support queries and transformations over all classes of data.

LINQ was a very interesting and unique project to work on. One reason is that it wasn't just about language features. In order to gain experience with the query capabilities we were developing, we needed to validate them against the important data domains—objects, relational, and XML. That led us to create the LINQ to Objects, LINQ to SQL, and LINQ to XML APIs, all of which were built alongside the language features. The synergy and agility we got from having a joint team working on both language and APIs was just amazing—and loads of fun!

Also, LINQ isn't just a single monolithic language feature, but rather a collection of several smaller and individually useful features—such as lambda expressions, extension methods, expression trees, object initializers, and anonymous types—that all come together to form the concept of Language Integrated Query. This made our work much more relevant and leveraged.

Finally, LINQ is big step toward a more declarative style of programming. This may be subtle, but it is really important. Programs written in today's imperative programming languages are too much about the “how” and too little about the “what.” We tend to over-specify the solutions to our programming problems—for example, by deconstructing queries into for loops, if statements, manipulation of temporary collections, and so on. By the time such programs run, it is all but impossible for the execution environment to “understand” what they do. The higher level semantic meaning has been lost in a sea of imperative, low-level instructions that must be blindly executed in exact sequence. This contrasts with LINQ queries, which preserve the programmer's exact intent and allows the execution infrastructure to be much smarter. A great example here is the Parallel LINQ (PLINQ) API that parallelizes query execution on concurrent hardware with practically no changes required to the source code.

Of course, the creation of a new technology such as LINQ is really only the first part of our job. The next step is to find ways to explain our new technology to the world of developers.

Essential LINQ is an important book because it provides a clear, easy-to-understand explanation of what LINQ does, how it does it, and the many practical ways you can use this technology to make your daily programming life easier and more productive.



Both authors of this text bring an important set of skills to this project. Throughout the development of LINQ, I worked daily with Dinesh Kulkarni in this role as Program Manager for the LINQ to SQL project. Few understand LINQ to SQL better than Dinesh, and the many insights he provides into LINQ will prove to be an invaluable tool for any reader of this book. The chapters Dinesh contributed to this book will be a resource that developers will frequently mine for their rich, well-thought-out content.

I've known Charlie Calvert since we worked together on Turbo Pascal and Delphi at Borland International. Charlie is an accomplished author with a gift for finding the key threads in a technology and explaining them to readers in a clear, easy-to-understand prose style. He is also one of nicest people I've met.

Charlie and Dinesh each bring important skills to this project that have enabled them to create an excellent book that shows how LINQ works and the many practical ways you can use it in your daily development process.

Anders Hejlsberg
Redmond, WA
February 2009

3

The Essence of LINQ

NOW THAT YOU'VE SEEN several practical examples of LINQ's syntax, it is time to view the technology from a more theoretical perspective. This chapter covers the seven foundations on which an understanding of LINQ can be built. LINQ is

- Integrated
- Unitive
- Extensible
- Declarative
- Hierarchical
- Composable
- Transformative

These ideas may sound esoteric at first, but I believe you will find them quite easy to understand. LINQ has a fundamental simplicity and elegance. In this chapter and the next, we explore LINQ's architecture, giving you a chance to understand how it was built and why it was built that way. This chapter explains goals that LINQ aims to achieve. The next chapter explains each of the pieces of the LINQ architecture and shows how they come together to achieve those goals.

Integrated

LINQ stands for Language Integrated Query. One of the central, and most important, features of LINQ is its integration of a flexible query syntax into the C# language.

Developers have many tools that have been crafted to neatly solve difficult tasks. Yet there are still dark corners in the development landscape. Querying data is one area in which developers frequently encounter problems with no clear resolution. LINQ aims to remove that uncertainty and to show a clearly defined path that is well-lit and easy to follow.

In Visual Studio 2005, attempts to query data in a SQL database from a C# program revealed an impedance mismatch between code and data. SQL is native to neither .NET nor C#. As a result, SQL code embedded in a C# program is neither type-checked nor IntelliSense-aware. From the perspective of a C# developer, SQL is shrouded in darkness.

Here is an example of one of several different techniques developers used in the past when querying data:

```
SqlConnection sqlConnection = new SqlConnection(connectString);
sqlConnection.Open();
System.Data.SqlClient.SqlCommand sqlCommand = new SqlCommand();
sqlCommand.Connection = sqlConnection;
sqlCommand.CommandText = "Select * from Customer";
return sqlCommand.ExecuteReader(CommandBehavior.CloseConnection)
```

Of these six lines of code, only the last two directly define a query. The rest of the lines involve setup code that allows developers to connect and call objects in the database. The query string shown in the next-to-last line is neither type-checked nor IntelliSense-aware.

After these six lines of code execute, the developers may have more work to do, because the data returned from the query is not readily addressable by an object-oriented programmer. You might have to write more lines of code to access this data, or convert it into a format that is easier to use.

The LINQ version of this same query is shorter, easier to read, color-coded, fully type-checked, and IntelliSense-aware. The result set is cleanly converted into a well-defined object-oriented format:

```
Northwind db = new Northwind(@"C:\Data\Northwnd.mdf");  
  
var query = from c in db.Customers  
            select c;
```

By fully integrating the syntax for querying data into .NET languages such as C# and VB, LINQ resolves a problem that has long plagued the development world. Queries become first-class citizens of our primary languages; they are both type-checked and supported by the powerful IntelliSense technology provided inside the Visual Studio IDE. LINQ brings the experience of writing queries into the well-lit world of the 21st century.

A few benefits accrue automatically as a result of integrating querying into the C# language:

- The syntax highlighting and IntelliSense support allow you to get more work done in less time. The Visual Studio editor automatically shows you the tables in your database, the correctly spelled names and types of your fields, and the operators you can use when querying data. This helps you save time and avoid careless mistakes.
- LINQ code is shorter and cleaner than traditional techniques for querying data and, therefore, is much easier to maintain.
- LINQ allows you to fully harness the power of your C# debugger while writing and maintaining queries. You can step through your queries and related code in your LINQ projects.

If language integration were the only feature that LINQ offered, that alone would have been a significant accomplishment. But we are only one-seventh of the way through our description of the foundations of LINQ. Many of the best and most important features are still to be covered.

Unitive

Before LINQ, developers who queried data frequently needed to master multiple technologies. They needed to learn the following:

- SQL to query a database
- XPath, Dom, XSLT, or XQuery to query and transform XML data

- Web services to access some forms of remote data
- Looping and branching to query the collections in their own programs

These diverse APIs and technologies forced developers to frantically juggle their tight schedules while struggling to run similar queries against dissimilar data sources. Projects often encountered unexpected delays simply because it was easier to talk about querying XML, SQL, and other data than it was to actually implement the queries against these diverse data sources. If you have to juggle too many technologies, eventually something important will break.

LINQ simplifies these tasks by providing a single, unified method for querying diverse types of data. Developers don't have to master a new technology simply because they want to query a new data source. They can call on their knowledge of querying local collections when they query relational data, and vice versa.

This point was illustrated in the preceding chapter, where you saw three very similar queries that drew data from three different data sources: objects, an SQL database, and XML:

```
var query = from c in GetCustomers()
            where c.City == "Mexico D.F."
            select new { City = c.City, ContactName = c.ContactName };

var query = from c in db.Customers
            where c.City == "Mexico D.F."
            select new { City = c.City, ContactName = c.ContactName };

var query = from x in customers.Descendants("Customer")
            where x.Attribute("City").Value == "Mexico D.F."
            select x;
```

As you can see, the syntax for each of these queries is not identical, but it is very similar. This illustrates one of LINQ's core strengths: a single, unitive syntax can be used to query diverse types of data. It is not that you never have to scale a learning curve when approaching a new data source, but only that the principles, overall syntax, and theory are the same even if some of the details differ.

You enjoy two primary benefits because LINQ is unitive:

- The similar syntax used in all LINQ queries helps you quickly get up to speed when querying new data sources.
- Your code is easier to maintain, because you are using the same syntax regardless of the type of data you query.

Although it arises naturally from this discussion, it is worth noting that SQL and other query languages do not have this capability to access multiple data sources with a single syntax. Those who advocate using SQL or the DOM instead of LINQ often forget that their decision forces their team to invest additional time in learning these diverse technologies.

Extensible Provider Model

In this text I have tended to define LINQ as a tool for querying SQL, XML, and the collections in a program. Strictly speaking, this is not an accurate description of LINQ. Although such a view is useful when you first encounter LINQ, it needs to be abandoned if you want to gain deeper insight. LINQ is not designed to query any particular data source; rather, it is a technology for defining *providers* that can be used to access any arbitrary data source. LINQ happens to ship with providers for querying SQL, XML, and objects, but this was simply a practical decision, not a preordained necessity.

LINQ provides developers with a syntax for querying data. This syntax is enabled by a series of C# 3.0 and C# 2.0 features. These include lambdas, iterator blocks, expression trees, anonymous types, type inference, query expressions, and extension methods. All of these features are covered in this book. For now you need only understand that they make LINQ possible.

When Visual Studio 2008 shipped, Microsoft employees frequently showed the image shown in Figure 3.1. Although people tend to think of LINQ as a means of enabling access to these data sources, this diagram actually depicts nothing more than the set of LINQ providers that were implemented by Microsoft at the time Visual Studio shipped. Granted, the team carefully planned which providers they wanted to ship, but their decisions were based on strategic, rather than technical, criteria.

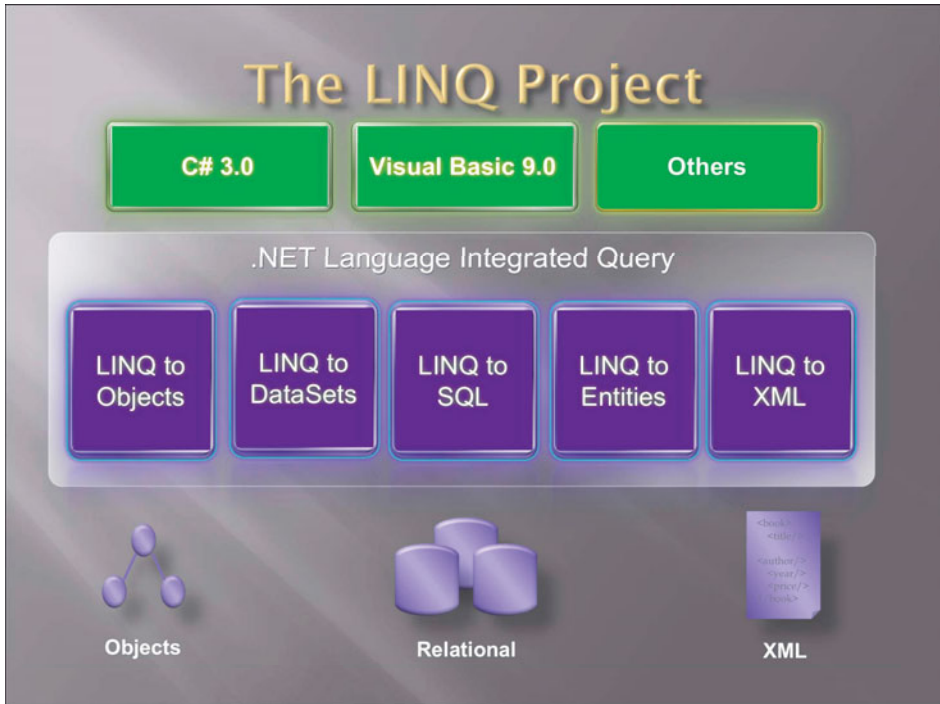


FIGURE 3.1 VB and C# ship with LINQ providers for databases, XML, and data structures found in a typical program.

Using the LINQ provider model, developers can extend LINQ to query other data sources besides those shown in Figure 3.1. The following are a few of the data sources currently enabled by third-party LINQ providers:

- | | |
|--------------------------|---------------------|
| LINQ Extender | LINQ to Google |
| LINQ over C# project | LINQ to Indexes |
| LINQ to Active Directory | LINQ to IQueryable1 |
| LINQ to Amazon | LINQ to JavaScript |
| LINQ to Bindable Sources | LINQ to JSON |
| LINQ to CRM | LINQ to LDAP |
| LINQ to Excel | LINQ to LLBLGen Pro |
| LINQ to Expressions | LINQ to Lucene |
| LINQ to Flickr | LINQ to Metaweb |
| LINQ to Geo | LINQ to MySQL |

LINQ to NCover	LINQ to Sharepoint
LINQ to NHibernate	LINQ to SimpleDB
LINQ to Opf3	LINQ to Streams
LINQ to Parallel (PLINQ)	LINQ to WebQueries
LINQ to RDF Files	LINQ to WMI

These projects are of varying quality. Some, such as the LINQ Extender and LINQ to IQueryable, are merely tools for helping developers create providers. Nevertheless, you can see that an active community is interested in creating LINQ providers, and this community is producing some interesting products. By the time you read this, I'm sure the list of providers will be longer. See Appendix A for information on how to get updated information on existing providers.

One easily available provider called LinqToTerraServer can be found among the downloadable samples that ship with Visual Studio 2008. You can download the VS samples from the release tab found at <http://code.msdn.microsoft.com/csharpsamples>.

After unzipping the download, if you look in the ...\\LinqSamples\\WebServiceLinqProvider directory, you will find a sample called LinqToTerraServer. The TerraServer web site, <http://terraserver-usa.com>, is a vast repository of pictures and information about geographic information. The LinqToTerraServer example shows you how to create a LINQ provider that queries the web services provided on the TerraServer site. For example, the following query returns all U.S. cities and towns named Portland:

```
var query1 = from place in terraPlaces
              where place.Name == "Portland"
              select new { place.Name, place.State };
```

This query returns a number of locations, but here are a few of the more prominent:

```
{ Name = Portland, State = Indiana }
{ Name = Portland, State = Maine }
{ Name = Portland, State = Michigan }
{ Name = Portland, State = Oregon }
{ Name = Portland, State = Texas }
```

```
{ Name = Portland, State = Alabama }  
{ Name = Portland, State = Arkansas }  
{ Name = Portland, State = Colorado }
```

In Chapter 17, “LINQ Everywhere,” you will see examples of several other providers, including LINQ to Flickr and LINQ to SharePoint. It is not easy to create a provider. After the code is written, however, it is easy to use the provider. In fact, you should already have enough familiarity with LINQ to see that it would be easy to modify the preceding query to suit your own purposes.

The LINQ provider model has hidden benefits that might not be evident at first glance:

- It is relatively open to examination and modification. As you read the next few chapters, you will find that most of the LINQ query pipeline is accessible to developers.
- It allows developers to be intelligent about how queries execute. You can get a surprising degree of control over the execution of a query. If you care about optimizing a query, in many cases you can optimize it, because you can see how it works.
- You can create a provider to publicize a data source that you have created. For instance, if you have a web service that you want C# developers to access, you can create a provider to give them a simple, extensible way to access your data.

I will return to the subject of LINQ providers later in the book. In this chapter, my goal is simply to make it clear that LINQ is extensible, and that its provider model is the basis on which each LINQ query model is built.

Query Operators

You don’t always need to use a LINQ provider to run queries against what might—at least at first—appear to be nontraditional data sources. By using the LINQ to Objects provider, and a set of built-in LINQ operators, you can run queries against a data source that does not look at all like XML or SQL data. For instance, LINQ to Objects gives you access to the reflection model that is built into C#.

The following query retrieves all the methods of the `string` class that are static:

```
var query = from m in typeof(string).GetMethods()
            where m.IsStatic == true
            select m;
```

The following are a few of the many results that this query returns:

```
System.String Join(System.String, System.String[])
System.String Join(System.String, System.String[], Int32, Int32)
Boolean Equals(System.String, System.String)
Boolean Equals(System.String, System.String, System.StringComparison)
Boolean op_Equality(System.String, System.String)
Boolean op_Inequality(System.String, System.String)
Boolean IsNullOrEmpty(System.String)
Int32 Compare(System.String, System.String)
Int32 Compare(System.String, System.String, Boolean)
Int32 Compare(System.String, System.String, System.StringComparison)
```

Using the power of LINQ, it is easy to drill into these methods to find out more about them. In particular, LINQ uses the extension methods mentioned in the preceding section to define a set of methods that can perform specific query operations such as ordering and grouping data. For instance, the following query retrieves the methods of the `string` class that are static, finds out how many overloads each method has, and then orders them first by the number of overloads and then alphabetically:

```
var query = from m in typeof(string).GetMethods()
            where m.IsStatic == true
            orderby m.Name
            group m by m.Name into g
            orderby g.Count()
            select new { Name = g.Key, Overloads = g.Count() };

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

The results of this query look like this:

```
{ Overloads = 1, Name = Copy }
{ Overloads = 1, Name = Intern }
{ Overloads = 1, Name = IsInterned }
{ Overloads = 1, Name = IsNullOrEmpty }
```

```
{ Overloads = 1, Name = op_Equality }  
{ Overloads = 1, Name = op_Inequality }  
{ Overloads = 2, Name = CompareOrdinal }  
{ Overloads = 2, Name = Equals }  
{ Overloads = 2, Name = Join }  
{ Overloads = 5, Name = Format }  
{ Overloads = 9, Name = Concat }  
{ Overloads = 10, Name = Compare }
```

This makes it obvious that `Format`, `Compare`, and `Concat` are the most frequently overloaded methods of the `string` class, and it presents all the methods with the same number of overloads in alphabetical order.

You can run this code in your own copy of Visual Studio because the LINQ to Objects provider ships with C# 3.0. Other third-party extensions to LINQ, such as LINQ to Amazon, are not included with Visual Studio. If you want to run a sample based on LINQ to Amazon or some other provider that does not ship with Visual Studio, you must download and install the provider before you can use it.

Declarative: Not How, But What

LINQ is declarative, not imperative. It allows developers to simply state what they want to do without worrying about how it is done.

Imperative programming requires developers to define step by step how code should be executed. To give directions in an imperative fashion, you say, “Go to 1st Street, turn left onto Main, drive two blocks, turn right onto Maple, and stop at the third house on the left.” The declarative version might sound something like this: “Drive to Sue’s house.” One says *how* to do something; the other says *what* needs to be done.

The declarative style has two advantages over the imperative style:

- It does not force the traveler to memorize a long set of instructions.
- It allows the traveler to optimize the route when possible.

It should be obvious that there is little opportunity to optimize the first set of instructions for getting to Sue’s house: You simply have to follow them by rote. The second set, however, allows the traveler to use his or her knowledge of the neighborhood to find a shortcut. For instance, a bike

might be the best way to travel at rush hour, whereas a car might be best at night. On occasion, going on foot and cutting through the local park might be the best solution.

Here is another example of the difference between declarative and imperative code:

```
// imperative style
List<int> imperativeList = new List<int>();
imperativeList.Add(1);
imperativeList.Add(2);
imperativeList.Add(3);

// declarative style
List<int> declarativeList = new List<int> { 1, 2, 3 };
```

The first example details exactly how to add items to a list. The second example states what you want to do and allows the compiler to figure out the best way to do it. As you will learn in the next chapter, both styles are valid C# 3.0 syntax. The declarative form of this code, however, is shorter, easier to understand, easier to maintain, and, at least in theory, leaves the compiler free to optimize how a task is performed.

These two styles differ in both the amount of detail they require a developer to master and the amount of freedom that each affords the compiler. Detailed instructions not only place a burden on the developer, but also restrict the compiler's capability to optimize code.

Let's consider another example of the imperative style of programming. As developers, we frequently end up in a situation where we are dealing with a list of lists:

```
List<int> list01 = new List<int> { 1, 2, 3 };
List<int> list02 = new List<int> { 4, 5, 6 };
List<int> list03 = new List<int> { 7, 8, 9 };

List<List<int>> lists = new List<List<int>> { list01, list02, list03 };
```

Here is imperative code for accessing the members of this list:

```
List<int> newList = new List<int>();

foreach (var item in lists)
{
    foreach (var number in item)
```



```
    {  
        newList.Add(number);  
    }  
}
```

This code produces a single list containing all the data from the three nested lists:

```
1  
2  
3  
4  
5  
6  
7  
8  
9
```

Notice that we have to write nested for loops to allow access to our data. In a simple case like this, nested loops are not terribly complicated to use, but they can become very cumbersome in more complex problem domains.

Contrast this code with the declarative style used in a LINQ program:

```
var newList = from list in lists  
              from num in list  
              select num;
```

You can access the results of these two “query techniques” in the same way:

```
foreach (var item in newList)  
{  
    Console.WriteLine(item);  
}
```

This code writes the results of either query, producing identical results, regardless of whether you used the imperative or declarative technique to query the data:

```
1  
2  
3  
4  
5  
6  
7  
8  
9
```

The difference here is not in the query's results, or in how we access the results, but in how we compose our query against our nested list. The imperative style can sometimes be verbose and hard to read. The declarative code is usually short and easy to read and scales more easily to complex cases. For instance, you can add an `orderby` clause to reverse the order of the integers in your result set:

```
var query = from list in lists
            from num in list
            orderby num descending
            select num;
```

You probably know how to achieve the same results using the imperative style. But it was knowledge that you had to struggle to learn, and it is knowledge that applies only to working with sequences of numbers stored in a `List<T>`. The LINQ code for reordering results, however, is easy to understand. It can be used to reorder not only nested collections, but also SQL data, XML data, or the many other data sources we query using LINQ.

To get the even numbers from our nested lists, we need only do this:

```
var query = from list in lists
            from num in list
            where num % 2 == 0
            orderby num descending
            select num;
```

Contrast this code with the imperative equivalent:

```
List<int> newList = new List<int>();

foreach (var item in lists)
{
    foreach (var number in item)
    {
        if (number % 2 == 0)
        {
            newList.Add(number);
        }
    }
}

newList.Reverse();
```

This imperative style of programming now has an `if` block nested inside the nested `foreach` loops. This is not only verbose and applicable to only a specific type of data, it also can be like a straight jacket for both the compiler and the developer. Commands must be issued and followed in a rote fashion, leaving little room for optimizations.

The equivalent LINQ query expression does not describe in a step-by-step fashion how to query our list of lists. It simply lets the developer state what he wants to do and lets the compiler determine the best path to the destination.

After nearly 50 years of steady development, the possibilities inherent in imperative programming have been extensively explored. Innovations in the field are now rare. Declarative programming, on the other hand, offers opportunities for growth. Although it is not a new field of study, it is still rich in possibilities.

■ Use the Right Tool for the Job

In extolling the virtues of LINQ's declarative syntax, I should be careful not to overstate my case. For instance, the LINQ operator called `ToList` is provided to allow developers to easily translate the sequence of results returned by a LINQ query into a traditional `List<T>`. This functionality is useful because some operations, such as randomly accessing items in a list (`myList[2]`), are more easily performed using the imperative syntax. One of the great virtues of C# 3.0 is that it allows you to easily move between imperative and declarative syntax, allowing you to choose the best tool for the job. My job right now is to help you understand the value of LINQ and the declarative style of programming. LINQ is indeed a very powerful and useful tool, but it is not the solution to all your problems.

Because LINQ is a new technology from Microsoft, you might find it a bit jarring to see me write that declarative programming is not new. In fact, declarative code has been with us nearly as long as imperative code. Some older languages such as LISP (which was first specified in 1958) make heavy use of the declarative style of programming. Haskell and F# are examples of

other languages that use it extensively. One reason LINQ and SQL look so much alike is that they are both forms of declarative programming.

The point of LINQ is not that it will replace SQL, but that it will bring the benefits of SQL to C# developers. LINQ is a technology for enabling a SQL-like declarative programming style inside a native C# program. It brings you the benefits of SQL but adds declarative syntax, as well as syntax highlighting, IntelliSense support, type checking, debugging support, the ability to query multiple data sources with the same syntax, and much more.

Hierarchical

Complex relationships can be expressed in a relational database, but the results of a SQL query can take only one shape: a rectangular grid. LINQ has no such restrictions. Built into its very foundation is the idea that data is hierarchical (see Figure 3.2). If you want to, you can write LINQ queries that return flat, SQL-like datasets, but this is an option, not a necessity.

Grid versus Hierarchies

LINQ's hierarchical data model is more flexible than the grid-like data returned from a SQL query.

Name	Company	OrderId
John	Boring,inc	332121
Mary	RidgeCo, A.E.	322336

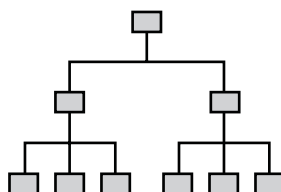


FIGURE 3.2 Both object-oriented languages and the developers who use them have a natural tendency to think in terms of hierarchies. SQL data is arranged in a simple grid.

Consider a simple relational database that has tables called Customers, Orders, and OrderDetails. It is possible to capture the relationship between these tables in a SQL database, but you cannot directly depict the relationship

in the results of a single query. Instead, you are forced to show the result as a join that binds the tables into a single array of columns and rows.

LINQ, on the other hand, can return a set of `Customer` objects, each of which owns a set of 0-to-*n* `Orders`. Each `Order` can be associated with a set of `OrderDetails`. This is a classic hierarchical relationship that can be perfectly expressed with a set of objects:

```
Customer
  Orders
    OrderDetails
```

Consider the following simple hierarchical query that captures the relationship between two objects:

```
var query = from c in db.Customers
            select new { City = c.City,
                       orders = from o in c.Orders
                                select new { o.OrderID }
            };
```

This query asks for the city in which a customer lives and a list of the orders the person has made. Rather than returning a rectangular dataset as a SQL query would, this query returns hierarchical data that lists the city associated with each customer and the ID associated with each order:

```
City=Helsinki  orders=...
orders: OrderID=10615
orders: OrderID=10673
orders: OrderID=10695
orders: OrderID=10873
orders: OrderID=10879
orders: OrderID=10910
orders: OrderID=11005
City=Warszawa  orders=...
orders: OrderID=10374
orders: OrderID=10611
orders: OrderID=10792
orders: OrderID=10870
orders: OrderID=10906
orders: OrderID=10998
```

This result set is multidimensional, nesting one set of columns and rows inside another set of columns and rows.

Look again at the query, and notice how we gain access to the Orders table:

```
orders = from o in c.Orders
```

The identifier `c` is an instance of a `Customer` object. As you will learn later in the book, LINQ to SQL has tools for automatically generating `Customer` objects given the presence of the `Customer` table in the database. Here you can see that the `Customer` object is not flat; instead, it contains a set of nested `Order` objects.

Listing 3.1 shows a simplified version of the `Customer` object that is automatically generated by the LINQ to SQL designer. Notice how LINQ to SQL wraps the fields of the `Customer` table. Later in this book, you will learn how to automatically generate `Customer` objects that wrap the fields of a `Customer` table.

LISTING 3.1 A Simplified Version of the `Customer` Object That the LINQ to SQL Designer Generates Automatically

```
public partial class Customer
{
    ... // Code omitted here
    private string _CustomerID;
    private string _CompanyName;
    private string _ContactName;
    private string _ContactTitle;
    private string _Address;
    private string _City;
    private string _Region;
    private string _PostalCode;
    private string _Country;
    private string _Phone;
    private string _Fax;
    private EntitySet<Order> _Orders;
    ... // Code omitted here
}
```

The first 11 private fields of the `Customer` object simply reference the fields of the `Customer` table in the database. Taken together, they provide a location to store the data from a single row of the `Customer` table. Notice, however, the last item, which is a collection of `Order` objects. Because it is

bound to the Orders table in a one-to-many relationship, each customer has from 0-to-*n* orders associated with it, and LINQ to SQL stores those orders in this field. This automatically gives you a hierarchical view of your data.

The same thing is true of the Order table, only it shows not a one-to-many relationship with the Customer table, but a one-to-one relationship:

```
public partial class Order
{
    ... // Code omitted here
    private int _OrderID;
    private string _CustomerID;
    private System.Nullable<int> _EmployeeID;
    private System.Nullable<System.DateTime> _OrderDate;
    private System.Nullable<System.DateTime> _RequiredDate;
    private System.Nullable<System.DateTime> _ShippedDate;
    private System.Nullable<int> _ShipVia;
    private System.Nullable<decimal> _Freight;
    private string _ShipName;
    private string _ShipAddress;
    private string _ShipCity;
    private string _ShipRegion;
    private string _ShipPostalCode;
    private string _ShipCountry;
    private EntityRef<Customer> _Customer;
    ... // Code omitted here
}
```

Again we see all the fields of the Orders table, their types, and whether they can be set to Null. The difference here is that the last field points back to the Customer table not with an `EntitySet<T>`, but an `EntityRef<T>`. This is not the proper place to delve into the `EntitySet` and `EntityRef` classes. However, it should be obvious to you that an `EntitySet` refers to a set of objects, and an `EntityRef` references a single object. Thus, an `EntitySet` captures a one-to-many relationship, and an `EntityRef` captures a one-to-one relationship.

The point to take away from this discussion is that LINQ to SQL captures not a flat view of your data, but a hierarchical view. A `Customer` class is connected to a set of orders in a clearly defined hierarchical relationship, and each order is related to the customer who owns it. LINQ gives you a hierarchical view of your data.

In a simple case like this, such a hierarchical relationship has obvious utility, but it is possible to imagine getting along without it. More complex

queries, however, are obviously greatly simplified by this architecture. Consider the following LINQ to SQL query:

```
var query = from c in db.Customers
            where c.CompanyName == companyName
            from o in c.Orders
            from x in o.Order_Details
            where x.Product.Category.CategoryName == "Confections"
            orderby x.Product.ProductName
            group x by x.Product.ProductName into g
            orderby g.Count()
            select new { Count = g.Count(), Product = g.Key };
```

Here we use LINQ's hierarchical structure to move from the Customers table to the Orders table to the Order_Details table without breaking a sweat:

```
var query = from c in db.Customers
            from o in c.Orders
            from x in o.Order_Details
```

The next line really helps show the power of LINQ hierarchies:

```
where x.Product.Category.CategoryName == "Confections"
```

The identifier `x` represents an instance of a class containing the data from a row of the Order_Details table. Order_Details has a relationship with the Product table, which has a relationship with the Category table, which has a field called `CategoryName`. We can slice right through that complex relationship by simply writing this:

```
x.Product.Category.CategoryName
```

LINQ's hierarchical structure shines a clarifying light on the relational data in your programs. Even complex relational models become intuitive and easy to manipulate.

We can then order and group the results of our query with a few simple LINQ operators:

```
orderby x.Product.ProductName
group x by x.Product.ProductName into g
orderby g.Count()
```


Trying to write the equivalent code using a more conventional C# style of programming is an exercise that might take two or three pages of convoluted code and involve a number of nested loops and `if` statements. Even writing the same query in standard SQL would be a challenge for many developers. Here we perform the whole operation in nine easy-to-read lines of code.

In this section, I have introduced you to the power of LINQ's hierarchical style of programming without delving into the details of how such queries work. Later in this book you will learn how easy it is to compose your own hierarchical queries. For now you only need to understand two simple points:

- There is a big difference between LINQ's hierarchical structure and the flat, rectangular columns and rows returned by an SQL query.
- Many benefits arise from this more powerful structure. These include the intuitive structure of the data and the ease with which you can write queries against this model.

Composable

The last two foundations of LINQ shed light on its flexibility and power. If you understand these two features and how to use them, you will be able to tap into some very powerful technology. Of course, this chapter only introduces these features; they are discussed in more detail in the rest of the book.

LINQ queries are composable: You can combine them in multiple ways, and one query can be used as the building block for yet another query. To see how this works, let's look at a simple query:

```
var query = from customer in db.Customers
            where customer.City == "Paris"
            select customer;
```

The variable that is returned from the query is sometimes called a computation. If you write a `foreach` loop and display the address field from the customers returned by this computation, you see the following output:

265, boulevard Charonne
25, rue Lauriston

You can now write a second query against the results of this query:

```
query2 = from customer in query
         where customer.Address.StartsWith("25")
         select customer;
```

Notice that the last word in the first line of this query is the computation returned from the previous query. This second query produces the following output:

25, rue Lauriston

LINQ to Objects queries are composable because they operate on and usually return variables of type `IEnumerable<T>`. In other words, LINQ queries typically follow this pattern:

```
IEnumerable<T> query = from x in IEnumerable<T>
                     select x;
```

This is a simple mechanism to understand, but it yields powerful results. It allows you to take complex problems, break them into manageable pieces, and solve them with code that is easy to understand and easy to maintain. You will hear much more about `IEnumerable<T>` in the next chapter.

The next chapter also details a feature called deferred execution. Although it can be confusing to newcomers, one of the benefits of deferred execution is that it allows you to compose multiple queries and string them together without necessarily needing to have each query entail an expensive hit against the server. Instead, three or four queries can “execute” without ever sending a query across the wire to your database. Then, when you need to access the result from your query, a SQL statement is written that combines the results of all your queries and sends it across the wire only once. Deferred execution is a powerful feature, but you need to wait until the next chapter for a full explanation of how and why it works. The key point to grasp now is that it enables you to compose multiple queries as shown here, without having to take an expensive hit each time one “executes.”

■ Discreet Computations and PLINQ

LINQ queries are not only composable, but also discreet. In other words, the computation returned by a query is a single self-contained expression with only a single entry point. This has important consequences for a field of study called Parallel LINQ (PLINQ). Because each computation returned by a query is discreet, it can easily be run concurrently on its own thread. PLINQ is discussed briefly in Chapter 17, “LINQ Everywhere.”

Transformative

SQL is poor at transformations, so we are unaccustomed to thinking about query languages as a tool for converting data from one format to another. Instead, we usually use specialized tools such as XSLT or brute-force techniques to transform data.

LINQ, however, has transformational powers built directly into its syntax. We can compose a LINQ query against a SQL database that effortlessly performs a variety of transforms. For instance, with LINQ it is easy to transform the result of a SQL query into a hierarchical XML document. You can also easily transform one XML document into another with a different structure. SQL data is transformed into a hierarchical set of objects automatically when you use LINQ to SQL. In short, LINQ is very good at transforming data, and this adds a new dimension to our conception of what we can do with a query language.

Listing 3.2 shows code that takes the results of a query against relational data and transforms it into XML.

LISTING 3.2 A Simple Query That Transforms the Results of a LINQ to SQL Query into XML

```
var query = new XElement("Orders", from c in db.Customers
    where c.City == "Paris"
    select new XElement("Order",
        new XAttribute("Address", c.Address),
        new XAttribute("City", c.City)));
```

Embedded in this query is a simple LINQ to SQL query that returns the `Address` and `City` fields from all the customers who live in Paris. In Listing 3.3 I’ve stripped away the LINQ to XML code from Listing 3.2 to show you the underlying LINQ to SQL query.

**LISTING 3.3** The Simple LINQ to SQL Query Found at the Heart of Listing 3.2

```
var query = from c in db.Customers
            where c.City == "Paris"
            select new { c.Address, c.City };
```

Here is the output from Listing 3.3:

```
265, boulevard Charonne
25, rue Lauriston
```

Here is the output from Listing 3.2:

```
<Orders>
  <Order Address="265, boulevard Charonne" City="Paris" />
  <Order Address="25, rue Lauriston" City="Paris" />
</Orders>
```

As you can see, the code in Listing 3.2 performs a transform on the results of the LINQ to SQL query, converting it into XML data.

Because LINQ is composable, the following query could then be used to run a second transform on this data:

```
var query1 = new XElement("Orders", new XAttribute("City", "Paris"),
    from x in query.Descendants("Order")
    where x.Attribute("City").Value == "Paris"
    select new XElement("Address", x.Attribute("Address").Value));
```

This query takes the XML results of the first query and transforms that XML into the following format:

```
<Orders City="Paris">
  <Address>265, boulevard Charonne</Address>
  <Address>25, rue Lauriston</Address>
</Orders>
```

LINQ is constantly transforming one type of data into another type. It takes relational data and transforms it into objects; it takes XML and transforms it into relational data. Because LINQ is extensible, it is at least theoretically possible to use it to tear down the walls that separate any two arbitrary data domains.

Because LINQ is both composable and transformative, you can use it in a number of unexpected ways:

- You can compose multiple queries, linking them in discrete chunks. This often allows you to write code that is easier to understand and maintain than traditional nested SQL queries.
- You can easily transform data from one data source into some other type. For instance, you can transform SQL data into XML.
- Even if you do not switch data sources, you can still transform the shape of data. For instance, you can transform one XML format into another format. If you look back at the section “Declarative: Not How, But What,” you will see that we transformed data that was stored in nested lists into data that was stored in a single list. These kinds of transformations are easy with LINQ.

Summary

In this chapter you have read about the foundations of LINQ. These foundations represent the core architectural ideas on which LINQ is built. Taken together, they form the essence of LINQ. We can summarize these foundations by saying the following about LINQ:

- It is a technique for querying data that is *integrated* into .NET languages such as C# and VB. As such, it is both strongly typed and IntelliSense-aware.
- It has a single *unitive* syntax for querying multiple data sources such as relational data and XML data.
- It is *extensible*; talented developers can write providers that allow LINQ to query any arbitrary data source.
- It uses a *declarative* syntax that allows developers to tell the compiler or provider what to do, not how to do it.
- It is *hierarchical*, in that it provides a rich, object-oriented view of data.
- It is *composable*, in that the results of one query can be used by a second query, and one query can be a subclause of another query. In many cases, this can be done without forcing the execution of any one query until the developer wants that execution to take place.

- It is *transformative*, in that the results of a LINQ query against one data source can be morphed into a second format. For instance, a query against a SQL database can produce an XML file as output.

Scattered throughout this chapter are references to some of the important benefits of LINQ that emerge from these building blocks. Although these benefits were mentioned throughout this chapter, I'll bring them together here in one place as a way of reviewing and summarizing the material discussed in this chapter:

- Because LINQ is integrated into the C# language, it provides syntax highlighting and IntelliSense. These features make it easy to write accurate queries and to discover mistakes at design time.
- Because LINQ queries are integrated into the C# language, it is possible for you to write code much faster than if you were writing old-style queries. In some cases, developers have seen their development time cut in half.
- The integration of queries into the C# language also makes it easy for you to step through your queries with the integrated debugger.
- The hierarchical feature of LINQ allows you to easily see the relationship between tables, thereby making it easy to quickly compose queries that join multiple tables.
- The unitive foundation of LINQ allows you to use a single LINQ syntax when querying multiple data sources. This allows you to get up to speed on new technologies much more quickly. If you know how to use LINQ to Objects, it is not hard to learn how to use LINQ to SQL, and it is relatively easy to master LINQ to XML.
- Because LINQ is extensible, you can use your knowledge of LINQ to make new types of data sources queryable.
- After creating or discovering a new LINQ provider, you can leverage your knowledge of LINQ to quickly understand how to write queries against these new data sources.
- Because LINQ is composable, you can easily join multiple data sources in a single query, or in a series of related queries.

- The composable feature of LINQ also makes it easy to break complex problems into a series of short, comprehensible queries that are easy to debug.
- The transformational features of LINQ make it easy to convert data of one type into a second type. For instance, you can easily transform SQL data into XML data using LINQ.
- Because LINQ is declarative, it usually allows you to write concise code that is easy to understand and maintain.
- The compiler and provider translate declarative code into the code that is actually executed. As a rule, LINQ knows more than the average developer about how to write highly optimized, efficient code. For instance, the provider might optimize or reduce nested queries.
- LINQ is a transparent process, not a black box. If you are concerned about how a particular query executes, you usually have a way to examine what is taking place and to introduce optimizations into your query.

This chapter touched on many other benefits of LINQ. These are described throughout this book. This entire text is designed to make you aware of the benefits that LINQ can bring to your development process. It also shows you how to write code that makes those benefits available to you and the other developers on your team.

The more you understand LINQ, the more useful it will be to you. As I have dug more deeply into this technology, I have found myself integrating LINQ into many different parts of my development process. When I use LINQ, I can get more work done in less time. The more I use it, the more completely these benefits accrue.



Index

Symbols

=> (goes to) operator, 91

A

accessing

- backing fields of properties, 22
- SharePoint sites, 500

Add Connection dialog box, 530

Add Function Import dialog box, 363

Add New Item dialog box, 354, 503

Add statements, 382

adding

- announcements to SharePoint, 501
- attributes to nodes, 36
- stored procedures, 363

Aggregate operator, 210, 216-219

Aggregate operators, 177, 210

Aggregate, 210, 216-219

Average, 210, 214-215

Count, 210-212

LongCount, 210-212

Max, 210-214

Min, 210-214

Sum, 210, 215-216

All operator, 181, 187

Ancestors() method, 408-409

Annotation() method, 461

annotations (XML), 459-461

anonymous types, 19, 79-82

class example, 81

passing out of methods, 81

query example, 82

querying collections of objects, 24-25

Any operator, 181, 186-187

APIs

Flickr, 495

LINQ to XML, 374

classes, 374

nodes, 374

XML attributes, creating, 377

XML declarations, creating, 378-381

XML documents, 378, 382-386

XML elements, creating, 375-377

AsEnumerable operator, 224

ASP.NET data binding, orders form

example, 310-316

Data Source Configuration Wizard, 312-313

filters, 313

grid view, 314

order details in web page, 311

query results, configuring, 313

results, 315

AsParallel() method, 489

assigning delegates, 85

Association attribute (object relationships), 257

associations

EDM, 351

relationships, 236

Athena. *See* LINQ to Flickr

Attach() method (multitier entities), 302-305

attaching multitier entities, 302-305

attributes

adding to nodes, 36

Association (object relationships), 257

Column, 234

InheritanceMapping, 271

Table (entity classes, creating), 233



XML, creating, 377
 XML nodes, 420
 xmlns, 425
 automatic properties, 69-73
 C# 2.0 syntax, 70
 example listing, 69
 OperatorId, 70-71
 OperatorName, 70
 private setter, 73
 prop snippet, 70
 querying collections of objects, 21-22
 Reflector generated code, 71-72
 warnings, 72
 Average operator, 210, 214-215

B

backing fields (properties), 22
 binding orders form example
 ASP.NET data binding, 311-315
 smart client data binding, 306-309
 Build C# key bindings, 544
 business logic
 entities, 474-476
 separation of concerns, 474-476

C

C#
 automatic properties, 69-73
 deferred execution, 111-117
 delegates, 83-89
 Development Center web site, 11
 expression trees, 122-128
 extension methods, 94-103
 IEnumerable<T> interface, 103-110
 initializers, 74-77
 iterators blog, 110
 key bindings, 539-544
 keywords/contextual keywords, 537-539
 lambdas, 89-94
 operator overrides, 117-121
 partial methods, 66-69
 programs, compiling, 524-526
 sample programs, 520-521
 types, 77-82
 version numbers, 521
 XML literals, 461-463
 Calvert, Charlie blog, 10
 Cast operator, 224
 casting XML elements, 397

categories of operators, 175-177
 ChangeConflictException, 295
 character data (XML), 406-407
 Chen, Raymond, 110
 classes
 Customer, 237, 263
 DataContext, 234-236
 entity
 creating, 233-234
 customizing, 341-344
 Enumerable, 179
 Extensions, 391
 GetListEnumerator, 115
 Item, 213
 LINQ to XML
 API, 374
 hierarchy, 422
 Musician, 145
 MyNumberServer, 169-172
 MyPartialClass, 67
 NorthwindDataContext, 338-340
 Order, 237, 263
 partial, adding to entity classes, 343
 persistent, 345-348
 Process, 511
 SpecialString, 97
 tables to classes, mapping, 240-243
 tables, mapping, 233-236
 ValidationEventArgs, 457
 XAttribute, 377
 XContainer, 388-391
 XDocument, 378, 390
 XElement, 376, 390-391
 XML, 387-388
 XmlSchemaSet, 456
 XNamespace, 427
 XmlNode, 389-391
 XObject, 415
 clauses
 from, 17, 132
 group-by
 projections, 154
 query expressions, 132-141
 let, 141-143
 order, 162
 query expressions, 131-132
 select
 projections, 154-155
 query expressions, 17, 132
 where, 17, 132
 code generation web site, 536

- collections
 - initializers, 16, 74-75
 - integers, querying, 14-20
 - collection initializers, 16
 - IEnumerable<T>, 19-20
 - query expressions, 16-18
 - type inferences, 18
 - objects, querying, 20
 - anonymous types, 24-25
 - automatic properties, 21-22
 - object initializers, 22-24
 - properties, 259
- Column attribute, 234
- command-line tools
 - edmgen.exe, 359
 - programs, compiling, 527-528
 - SqlMetal, 243
- commands. *See* methods
- compiling
 - C# programs, 524-526
 - programs from command line, 527-528
 - queries, LINQ to SQL, 277-278
- composability, 7, 58-59, 517
- composing
 - Element operators, 202-203
 - query expressions, 136
 - group-by clauses at the end, 136-139
 - group-by clauses in the middle, 139-141
 - into keyword, 141
 - let clauses, 141-143
- Concat operator, 205
- concurrency, managing, 290-292, 476-477
 - optimistic, 292
 - conflict detection, 292-295
 - conflict resolution, 295-297
 - limitations, 478
 - refreshing entities, 297-299
 - set of changes, 298
 - units of work, 478
- Configure Behavior dialog box, 332
- configuring CUD operations, 332
- conflicts
 - detection, 292-295
 - resolution, 295-297
- connections
 - databases, 528-530
 - Object Relational Designer, 531-533
 - Visual Studio 2008, 528-530
 - managing, 302
 - SharePoint, 503
- Contains operator, 181, 188-190
- content (XML), 372
- context, defining, 479-480
- contextual keywords
 - C#, 537-539
 - defined, 17
 - var, 18
- continuations (query expressions), 141
- Conversion AsEnumerable, 228
- Conversion operators, 176, 223
 - AsEnumerable, 224
 - Cast, 224
 - OfType, 224-227
 - ToArray, 224-226
 - ToDictionary, 224, 227
 - ToList, 224-226
 - ToLookup, 224
- core operators, 177
- costs (performance), 480-481
- Count operator, 210-212
- CreateDatabase() method, 316-317
- CreateDatabase sample program, 450
- CreatePlanets sample program, 381
- create, update, delete. *See* CUD operations
- creating
 - database connections, 528-533
 - Object Relational Designer, 531-533
 - Visual Studio 2008, 528-530
 - databases with LINQ to SQL, 316-318
 - databases programmatically, 442-443
 - data schemas, viewing, 449
 - table declarations, 450
 - tables, creating, 444-447
 - XML data, transferring, 447-449
 - EDMs, 354
 - connections, 356
 - database generation, 354
 - expanding database objects, 356
 - generated model, 358
 - models, adding, 354
 - new items, adding, 354
 - selecting database objects, 356
 - entity classes, 233-234
 - Flickr application, 497-499
 - persistent classes, 345-348
 - XHTML, 431-437
 - XML
 - attributes, 377
 - declarations, 378-381
 - documents, 378, 382-383
 - elements, 375-377
- CRUD (creating, reading, updating, and deleting) operations, 240, 361-362
- CUD (create, update, delete) operations, 330-332

Customer class
 Orders property, 237
 OtherKey property, 237
 relationship with Order class, 263

Customer object
 LINQ to SQL Designer generated example, 55-56
 query listing, 23
 customers, moving orders between, 287
 customizing
 entity classes, 341-343
 lifecycle events, 344
 partial classes, adding, 343
 service contract compatibility, 342
 WCF DataContract attribute, 341-342
 generated code, 337-338
 entity classes, 341-344
 NorthwindDataContext class, 338-340

D

data binding (LINQ to SQL), 305
 ASP.NET, 310, 313, 316
 smart client, 306, 309-310
 Data Definition Language (DDL), 269
 data shaping, 470-472
 Data Source Configuration Wizard, 306
 data selection, 313
 launching, 312
 data sources
 adding new, 306
 choosing, 306
 Data Sources window, populating, 308
 enabled by third-party LINQ providers, 44
 Data Transfer Objects (DTOs), 471
 Database Explorer, 437
 databases
 connections, 528-530
 Object Relational Designer, 531-533
 Visual Studio 2008, 528-530
 creating with LINQ to SQL, 316-318
 creating programmatically, 442-443
 data schemas, viewing, 449
 table declarations, 450
 tables, creating, 444-447
 XML data, transferring, 447-449
 Northwind
 DataContext class, customizing, 338-340
 sample programs, 521
 scalar-valued functions, adding, 328
 stored procedures multiple results, 324

stored procedures single results, 322
 TVFs, adding, 326
 queries, 26-30
 relational, 247
 DataContext class
 concurrency management, 478
 mapping classes to tables, 234-236
 performance optimization, 483
 persistence-related services, 474
 DDL (Data Definition Language), 269
 Debugger integration with LINQ, 112
 Debugging C# key bindings, 543
 declarations
 operators, 180
 XML, 371-373
 creating, 378-381
 DOCTYPE, 434
 declarative programming, 2, 7, 48-52, 383, 516
 default namespaces (XML), 426-429
 DefaultIfEmpty operator, 198, 203-204
 deferred execution, 111-117
 compiler code produced upon calling
 GetSequence() method, 114
 foreach code execution, 113-114
 iterator sequence example, 111-112
 overview, 59
 projections, 156-159
 queries, 265
 reasons for, 116
 deferred loading (LINQ to SQL), 264-266
 delegates, 83-86
 assigning, 85
 data sorting pattern, 85
 generic, 86-89
 MyDelegate example, 83-84
 DeleteOnSubmit() method, 284
 deleting
 entities, 284-285
 XML nodes, 417-418
 DescendantNodes() method, 402-404
 DescendantNodesAndSelf() method, 404
 Descendants() method, 397-400
 descendants (XML), 397-400
 dialog boxes
 Add Connection, 530
 Add Function Import, 363
 Add New Item, 354, 503
 Configure Behavior, 332
 New Project, 524
 discreet computations, 60
 Distinct operator, 204, 207

DOCTYPE declarations, 434
 documents (XML)
 creating, 378, 382-383
 RSS feeds, loading, 385
 saving to disks, 383, 386
 dotting through relationships, 258
 DTDs (Document Type Definitions), 453
 DTOs (Data Transfer Objects), 471
 dynamic SQL, 468-469

E

eager loading (LINQ to SQL), 266-268
 Edit C# key bindings, 540
 editing XML nodes, 418-419
 EDM (Entity Data Model), 349-352
 associations, 351
 creating, 354
 connections, 356
 database generation, 354
 expanding database objects, 356
 generated model, 358
 models, adding, 354
 new items, adding, 354
 selecting database objects, 356
 CRUD operations, 361-362
 generated code, 359-360
 Generator, 359
 relationships, 351
 stored procedures, 362
 adding, 363
 mapping to methods, 363
 parameters, mapping, 365
 update procedure, 364
 edmggen.exe command-line tool, 359
 EF (Entity Framework), 349
 associations, 351
 CRUD operations, 361-362
 EDM, 350-352
 entity models, creating, 354
 connections, 356
 database generation, 354
 expanding database objects, 356
 generated model, 358
 models, adding, 354
 new items, adding, 354
 selecting database objects, 356
 generated code, 359-360
 layers, 352-353
 relationships, 351
 stored procedures, 362
 adding, 363
 mapping to methods, 363
 parameters, mapping, 365
 update procedure, 364
 Element operators, 176, 198
 composing, 202-203
 DefaultIfEmpty, 198, 203-204
 ElementAt, 198, 202
 ElementAtOrDefault, 198
 First, 198
 FirstOrDefault, 198-199
 Last, 198, 200
 LastOrDefault, 198-200
 Single, 198, 201
 SingleOrDefault, 198
 ElementAt operator, 198, 202
 ElementAtOrDefault operator, 198
 elements (XML), 373
 casting, 397
 creating, 375-377
 nodes, adding, 421
 ElementsAfterSelf() method, 409
 ElementsBeforeSelf() method, 409
 Empty operator, 181, 186
 enabling data sources, 44
 ends (entity relationships), 351
 entities. *See also* objects
 business logic, 474-476
 classes
 creating, 233-234
 customizing, 341-343
 concurrent changes, 290-292
 optimistic. *See* optimistic concurrency
 refreshing entities, 297-299
 set of changes, 298
 connections, managing, 302
 CUD operations, configuring, 332
 data binding, 305
 ASP.NET, 310, 313, 316
 smart client, 306, 309-310
 data shaping, 470-472
 deleting, 284-285
 inheritance, 269-274
 class diagram, 269
 class hierarchy on designer surface, 274
 designers, 273
 InheritanceMapping attribute, 271
 inserting, 282-283
 lifecycle, 282, 344
 loading, 264
 deferred, 264-266
 eager, 266-268
 modifying, 239-240
 multitier, attaching, 302-305
 object identity, 255-256

persistence, 473-474
 refreshing, 297-299
 relationships, 256-351

- Association attribute, 257
- collection properties, 259
- dotting through, 258
- joining tables, 260-262
- managing, 286-288
- mapping, 264

 retrieving, 252-254
 security, improving, 484-486
 submitting changes, 288-290
 transactions, 299-302
 updating entities, 285-286
 Entity Data Model. *See* EDM
 Entity Data Model Wizard, 354

- connections, 356
- database generation, 354
- expanding database objects, 356
- generated model, 358
- selecting database objects, 356

 Entity Framework. *See* EF
 EntityClient Data Provider layer, 352
 EntityRef type (Order class), 237
 Enumerable class, 179
 Equality operators, 177
 equijoins, 145
 Except operator, 205-207
 explicit conversion operators, 413
 Express Tree Visualizer, 125

- lambda expression, parsing, 126
- LINQ to SQL code, parsing, 127
- opening, 125
- popup menu item example, 127

 Expression property, 173
 expression trees, 122-125, 128

- Expression Tree Visualizer, 125-127
- lambda based example, 123
- parsing operations, 123-124

 expressions (query)

- clauses, 131-132
- composing, 136-143
- continuations, 141
- IEnumerable<T> interface, 168
- IQueryable<T> interface, 168, 172-173
- joins, 143-153
- MyNumberServer class, 169-172
- nomenclature, 130
- projections, 153
 - deferred execution, 156-159
 - new class in a select clause example, 154
 - overview, 154-156

- SelectMany operator overloads, 164-167
- SelectMany operators, 159-164
 - transforming objects into XML, 155
- range variables, 133-135
- Set operators, 208-210

 extensibility, 7, 516
 extensible provider model, 43-46

- data sources enabled by third-party LINQ providers, 44
- LinqToTerraServer example, 45
- query operators, 46-48

 extension methods, 2, 94-98

- scoping, 98-103
- static method string class example, 95-96

 Extensions class, 391

F

FakeWeatherData program, 491-492
 famous Romans code reuse example, 178
 File C# key bindings, 540
 files (XML)

- creating, 34-36
- loading, 536
- parsing, 31-34

 filters, choosing, 313
 finding providers, 536
 First operator, 198
 FirstFourPlanets.xml, 392-393

- names of planets code, 393
- output, 394

 FirstOrDefault operator, 198-199
 flattening group joins, 261
 Flickr, 495-499
 FlickrXplorer application, 496
 foreach loops

- deferred execution, 113-114
- operator code reuse, 178
- querying collections of integers, 19

 foundational qualities of LINQ, 516-517
 foundations

- composability, 58-59
- declarative programming, 48-52
- extensibility, 43-46
 - data sources enabled by third-party LINQ providers, 44
 - LinqToTerraServer example, 45
 - query operators, 46-48
- hierarchies, 53-58
- integration, 40-41
- transformational powers, 60-61
- unified method for querying, 42-43

from clauses (query expressions), 17, 132
 from keyword (range variables), 135
 fundamentals (XML), 370-371, 374
 future
 LINQ to Flickr, 495
 API, 495
 applications, creating, 497-499
 download, 496
 FlickrXplorer application, 496
 web site, 495
 LINQ to SharePoint
 announcements, adding, 501
 assemblies, registering, 502
 connections, 503
 example site, 501
 LINQ to SharePoint Entity Wizard, 503
 queries, 508
 requirements, 500
 site access, 500
 site lists, choosing, 505
 Solution Explorer example, 507
 template, choosing, 503
 web site, 499
 wizard choices, reviewing, 50
 PLINQ, 488-490
 complexities, 494
 FakeWeatherData program, 491-492
 IParallelEnumerble interface, 489
 LINQ query time differences, measuring, 493-494
 Parallel Computing team web site, 488
 ParallelQuery<int>, 489
 performance improvements, 490

G

generated code
 automatic properties, 71-72
 customizing, 337-338
 entity classes, 341-344
 NorthwindDataContext class, 338-340
 EF, 359-360
 Generation operators, 177, 181
 All, 181, 187
 Any, 181, 186-187
 Contains, 181, 188-190
 Empty, 181, 186
 Range, 181-183
 Repeat, 181-185
 generic delegates, 86-89
 generics
 overview, 9
 syntax, 10

GetChangeSet() method, 298
 GetListEnumerator class, 115
 GetProcesses() method, 511
 GetSequence() method, 113
 GettingStartedWithLinqToXML sample program, 382
 goes to (=>) operator, 91
 graphical designer for mapping. *See* Object Relational Designer
 group joins, 147-149, 261
 group-by clauses
 projections, 154
 query expressions, 132
 at the end, 136-139
 in the middle, 139-141
 GroupByOperators sample program
 group-by clauses, 139
 let clause, 142
 Grouping operators, 177

H

handling conflicts, 295-297
 hierarchies, 53-58
 Customer object generated by LINQ to SQL Designer example, 55-56
 two object relationship, 54
 hierarchy, 7, 516
 LINQ to XML classes, 422
 XML, 372, 387-388
 Hossain, Mehruz, 496

I

identifiers (XML namespaces), 424
 identity (objects), 255-256
 IEnumerable<T> interface, 103-105
 Element operators, composing, 202-203
 enumeration, 106-108
 implementing with yield return, 108
 iterators, 108-110
 operator support, 119
 query expressions, 168
 querying collections of integers, 19-20
 sequences, 106
 IEnumerator<T> interface, 107
 IEqualityComparer interface, 188
 IL (Intermediate Language), 22
 immediate loading, 266-268
 imperative programming, 48, 383
 inferred types, 78-79
 Informit web site, 11

inheritance (LINQ to SQL), 269-274
 class diagram, 269
 class hierarchy on designer surface, 274
 designers, 273
 InheritanceMapping attribute, 271
 InheritanceMapping attribute, 271
 initializers, 74
 collection, 16, 74-75
 object, 76-77
 objects, 22-24
 inner joins, 143-146
 inserting
 entities, 282-283
 XML nodes, 420-421
 InsertOnSubmit() method, 283
 installing
 .NET Framework, 522
 SQL Server Express, 523
 Visual Studio Express, 522
 integration, 7, 40-41, 516
 IntelliSense C# key bindings, 541
 interfaces
 IEnumerable<T>, 103-105
 Element operators, composing, 202-203
 enumeration, 106-108
 implementing with yield return, 108
 iterators, 108-110
 operator support, 119
 query expressions, 168
 querying collections of integers, 19-20
 sequences, 106
 IEnumerator<T> interface, 107
 IEqualityComparer, 188
 IParallelEnumerable<int>, 489
 IQueryable<T>, 172-173
 XmlLineInfo, 415
 Intermediate Language (IL), 22
 Intersect operator, 204-207
 into keyword (query expressions), 141
 into operators, 150
 IParallelEnumerable interface, 489
 IQueryable<T> interface, 172-173
 IQueryable Toolkit, 496
 IsPrimaryKey property (Column attribute), 234
 IsState() method, 96
 Item class, 213
 XmlLineInfo interface, 415

J-K

Join operators, 176, 260-262
 joining tables, 260-262

joins, 143
 equijoins, 145
 group, 147-149, 261
 inner, 143-146
 left outer, 149-151
 LINQ to SQL, 153
 object-oriented model, 152-153
 web site, 536

key bindings (C#), 539-544
 keywords
 C#, 537-539
 contextual, 17-18
 from, 135
 into, 141
 var, 78-79
 Kulkarni, Dinesh blog, 11

L

lambdas, 2, 89-92
 => (goes to) operator, 91
 C# 2.0 anonymous methods, compared, 92
 exercises, 92-94
 expression trees based on lambdas
 example, 123
 local variables, 91
 methods, compared, 90
 query expressions role, 118
 syntax, 89
 Language Integrated Query. *See* LINQ
 Last operator, 198-200
 LastOrDefault operator, 198-200
 layers (EF), 352-353
 left outer joins, 149-151
 let clauses (query expressions), 141-143
 lifecycle (entities), 240, 282
 deleting entities, 284-285
 entity class customizations, 344
 inserting entities, 282-283
 managing relationships, 286-288
 submitting changes, 288-290
 updating entities, 285-286
 line numbers (XML nodes), 414-417
 LINQ (Language Integrated Query), 1
 building blocks, 517
 foundational qualities, 516-517
 overview, 3
 LINQ to Entities. *See* EF
 LINQ to Flickr, 495
 API, 495
 applications, creating, 497-499

- download, 496
- FlickrXplorer application, 496
- web site, 495
- LINQ to Objects
 - LINQ to SQL, compared, 8-9, 105
 - providers for existing technologies, 509
 - system processes, querying, 510-511
- LINQ to Relational technology, 366
- LINQ to SharePoint, 499
 - announcements, adding, 501
 - assemblies, registering, 502
 - connections, 503
 - example site, 501
 - LINQ to SharePoint Entity Wizard, 503
 - queries, 508
 - requirements, 500
 - SharePoint site access, 500
 - site access, 500
 - site lists, choosing, 505
 - Solution Explorer example, 507
 - template, choosing, 503
 - web site, 499
 - wizard choices, reviewing, 506
- LINQ to SharePoint Entity Wizard
 - choices, reviewing, 506
 - connections, 503
 - first page, 503
 - site lists, choosing, 505
- LINQ to SQL
 - concurrent changes, 290-292
 - optimistic. *See* optimistic concurrency
 - refreshing entities, 297-299
 - set of changes, 298
 - connections, managing, 302
 - CRUD, 240
 - data binding, 305
 - ASP.NET, 310, 313, 316
 - smart client, 306, 309-310
 - databases, creating, 316-318
 - entities
 - deleting, 284-285
 - inserting, 282-283
 - modifying, 239-240
 - submitting changes, 288-290
 - relationships. *See* entities, relationships
 - retrieving, 252-254
 - updating entities, 285-286
 - expression tree, parsing, 127
 - inheritance, 269-274
 - Join operator, 260-261
 - joins, 153
 - LINQ to Objects, compared, 8-9, 105
 - LinqToSqlWithoutDesigner example, 26-30
 - loading options, 264-268
 - mapping classes to tables, 233-236
 - mapping tools
 - command-line, 243
 - graphical designer. *See* Object Relational Designer
 - multitier entities, attaching, 302-305
 - object identity, 255-256
 - overview, 231-233
 - performance
 - compiled queries, 277-278
 - queries versus results, 275-276
 - relationships, 236-239
 - security, 278-279, 484-486
 - stored procedures, 319
 - CUD operations, 330-332
 - executing, 322
 - mapping, 320
 - relationships, loading, 333-334
 - returning entity type example, 322-323
 - returning multiple results, 324-326
 - returning rows with key values
 - example, 322
 - scalar-valued functions, 328-329
 - TVFs, 326-328
 - transactions, 299-302
 - transformation into XML, 60
 - translating, 248-250
 - base class library methods, 250
 - scalar UDFs, 251
 - T-SQL methods, 251
- LINQ to SQL Designer
 - Customer object example, 55-56
 - generated code, customizing, 337-338
 - entity classes, 341-344
 - NorthwindDataContext class, 338-340
- LINQ to XML
 - Ancestors() method, 408-409
 - annotations, 459-461
 - character data, 406-407
 - classes, 374, 422
 - databases, creating programmatically, 442-443
 - data schemas, viewing, 449
 - table declarations, 450
 - tables, creating, 444-447
 - XML data, transferring, 447-449
 - declarations, 434
 - DescendantNodes() method, 402-404
 - DescendantNodesAndSelf() method, 404

- descendants, 397-400
- ElementsBeforeSelf()/ElementsAfterSelf()
 - methods, 409
- explicit conversion operators, 413
- namespaces, 424-426
 - accidentally omitting, 436
 - default, 426-429
 - identifiers, 424
 - prefixes, 425
 - schema validation, 458-459
 - XHTML, creating, 435
- nodes, 374
 - adding, 420-421
 - attributes, 420
 - deleting, 417-418
 - editing, 418-419
 - elements, 421
 - line numbers, 414, 417
 - missing, 411-414
 - parent nodes, 407-409
 - queries, 392
 - composition, 400-402
 - FirstFourPlanets.xml, 392-394
 - schema validation, 451-459
 - text node searches, 405
 - transformations, 429
 - relational data into XML, 437-442
 - XHTML, creating, 431-437
 - XML between formats, 430-431
 - XML data into databases, 447-449
- XML
 - attributes, creating, 377
 - declarations, creating, 378-381
 - documents, creating, 378, 382-383
 - documents, saving to disks, 383, 386
 - elements, creating, 375-377
 - literals, 461-463
- XML files
 - creating, 34-36
 - parsing, 31-34
- XNamespace class, 427
- LinqDataSource application example
 - Data Source Configuration Wizard, 312-313
 - filters, 313
 - grid view, 314
 - query results, configuring, 313
 - results, 315
- LINQExtender, 496
- LinqToSqlWithoutDesigner example, 26-30
- LinqToTerraServer provider, 45
- List<T>, declaring, 104
- listings
 - anonymous class example, 81
 - automatic properties
 - C# 2.0 syntax, 70
 - example, 69
 - Reflector generated code, 71-72
 - Contains operator, 188
 - Count operator, 211
 - Customer objects
 - generated by LINQ to SQL Designer, 55-56
 - query, 23
 - databases, creating programmatically
 - table declarations, 450
 - tables, creating, 444-445
 - deferred execution, iterator sequence
 - example, 111-112
 - DescendantNodes() method, 403
 - DescendantNodesSelf() method, 404
 - Descendants program, 398-399
 - Distinct operator, 207
 - enumerating with IEnumerable<T>
 - interface, 107
 - Except method, 208
 - expression trees
 - lambda based, 123
 - parsing operations, 123-124
 - extension methods
 - scoping, 99-103
 - static method string class example, 95-96
 - FirstFourPlanets.xml, 392-393
 - names of planets code, 393
 - output, 394
 - Flickr application, 498
 - generic delegates example, 88
 - IEnumerable<T> interface, implementing
 - with yield return, 108
 - inner joins, 143-145
 - Intersect operator, 206
 - left outer joins, 149
 - LINQ to SQL, LinqToSqlWithoutDesigner
 - example, 26-30
 - LINQ to XML
 - customers who live in Mexico D.F., 31
 - data stored in Customers.xml file, 32
 - Min/Max operators
 - highest/lowest values in a sequence, 212
 - Item class/GetItems() method, 212
 - min/max values for complex types with
 - multiple fields, 213
 - MyDelegate example, 83-84
 - MyNumberServer class, 169

- object-oriented join model, 152
- operator overrides, Where operator, 119-120
- OrderByDescending operator, 221
- Ordering operators, 220
- partial method example, 67-68
- projections
 - deferred execution, 158
 - new class in a select clause example, 154
 - SelectMany operators, 159
 - transforming objects into XML, 155
- query expressions
 - group-by clauses, 139
 - let clause, 142
- querying collections of integers, 14
- relational data transformation into XML, 437-439
- Repeat operator, 184
- system processes, querying, 510-511
- ToArray operator, 226
- transformations, LINQ to SQL query into XML, 60
- Union operator, 205
- XHTML documents
 - creating, 432-434
 - example, 435
- XML
 - annotations, 459
 - character data, 406-407
 - declarations, creating, 378-379
 - documents, creating one node at a time, 382
 - documents, example, 370
 - documents, saving to disks, 383
 - files, creating, 34
 - nodes, deleting, 417
 - schema validation example, 455-456
 - single node access, 401
- Load() method, 385
- loading
 - entities, 264
 - LINQ to SQL
 - deferred, 264-266
 - eager, 266-268
 - relationships, 333-334
 - XML files, 536
- local variables, 91
- locating operators, 179-181
- LongCount operators, 210-212

M

- managing
 - concurrency, 476-478
 - connections, 302
 - entity relationships, 286-288
 - transactions, 299-302
- many-to-many relationships, 262
- mapping
 - classes to tables, 233-236
 - relationships, 264
 - stored procedures, 320-322
 - methods, 363
 - parameters, 365
 - tables to classes, 240-243
 - TVFs, 327
- Max operators, 210-214
- metadata declarations, 121
- methods
 - Ancestors(), 408-409
 - Annotation(), 461
 - anonymous types, passing, 81
 - AsParallel(), 489
 - Attach(), 302-305
 - CreateDatabase(), 316-317
 - DeleteOnSubmit(), 284
 - DescendantNodes(), 402-404
 - DescendantNodesAndSelf(), 404
 - Descendants(), 397-400
 - ElementsAfterSelf(), 409
 - ElementsBeforeSelf(), 409
 - extension, 2, 94-103
 - GetChangeSet(), 298
 - GetProcesses(), 511
 - GetSequence(), 113
 - InsertOnSubmit(), 283
 - IsState(), 96
 - lambdas, compared, 90
 - LINQ to SQL translation, 250-251
 - Load(), 385
 - MoveNext(), 115
 - MyPartialMethod(), 67
 - OnCreated(), 339
 - OnRequiredDateChanged(), 344
 - OnRequiredDateChanging(), 344
 - partial, 66-69
 - ProcessList(), 434
 - Refresh(), 297
 - Resolve(), 297
 - scalar UDFs, 251
 - ShowExcept(), 208

ShowIntersect(), 206
 ShowRepeat(), 184
 ShowUnion(), 205
 StartsWith(), 250
 stored procedures, mapping, 363
 SubmitChanges(), 288-290
 overriding, 340
 performance optimization, 483
 T-SQL, 251
 ToString(), 384
 Validate(), 456
 ValidationEventHandler(), 457
 WriteFile(), 68
 XContainer class, 391
 XElement class, 391
 XElement class, 391
 Microsoft
 MVC download, 496
 Office SharePoint Server 2007 VHD, 500
 Min operator, 210-214
 minimal update statements, 286
 missing nodes (XML), 411-414
 modifying entities, 239-240
 ModifyNodes program, 417
 MoveNext() method, 115
 multitier entities, attaching, 302-305
 Musician class, 145
 MVC download, 496
 MyDelegate example, 83-84
 MyNumberServer class, 168-172
 MyPartialClass class, 67
 MyPartialMethod() method, 67

N

namespaces
 System.Diagnostics, 510-511
 System.Xml.Linq, 32
 XML, 424-426
 accidentally omitting, 436
 default, 426-429
 identifiers, 424
 prefixes, 425
 schema validation, 458-459
 XHTML, creating, 435
 Navigation C# key bindings, 541
 .NET
 Development web site, 535
 Framework
 installing, 522
 version numbers, 521
 web site, 536

New Project dialog box, 524
 nodes (XML), 374
 adding, 420-421
 attributes, 36, 420
 deleting, 417-418
 editing, 418-419
 elements, 421
 line numbers, 414, 417
 missing, 411-414
 parents, 407-409
 nomenclature (query expressions), 130
 Northwind database
 connections
 Object Relational Designer, 531-533
 Visual Studio 2008, 528-530
 customizing, 338-340
 DataContext class, customizing, 338-340
 download, 521
 sample programs, 521
 scalar-valued functions, adding, 328
 stored procedures, 322-324
 TVFs, adding, 326

O

object initializers, 22-24, 76-77
 object-oriented join model, 152-153
 Object Relational Designer
 database connections, creating, 531-533
 mapping tables to classes, 240-243
 Object Services layer, 352
 objects. *See also* entities
 collections, querying, 20
 anonymous types, 24-25
 automatic properties, 21-22
 object initializers, 22-24
 Customer, 23, 55-56
 hierarchy example, 54
 identity, 255-256
 lifecycle, 240
 references, 255
 relationships, 236, 256-259
 Association attribute, 257
 collection properties, 259
 defining, 236-238
 dotting through, 258
 joining tables, 260-262
 mapping, 264
 querying across, 238-239
 OfType operator, 224-227
 OnCreated() method, 339
 OnRequiredDateChanged() method, 344

- OnRequiredDateChanging() method, 344
- operations
 - CRUD, 240, 361-362
 - CUD, 330-332
 - parsing, 123-124
- OperatorId automatic property, 70-71
- OperatorName automatic property, 70
- operators
 - Aggregate, 177, 210
 - Aggregate, 210, 216-219
 - Average, 210, 214-215
 - Count, 210-212
 - LongCount, 210-212
 - Max, 210-214
 - Min, 210-214
 - Sum, 210, 215-216
- categories, 175-177
- choosing, 177
- Conversion, 223
 - AsEnumerable, 224, 228
 - Cast, 224
 - OfType, 224-227
 - ToArray, 224-226
 - ToDictionary, 224, 227
 - ToList, 224-226
 - ToLookup, 224
- core, 177
- declarations, 180
- Element, 176, 198
 - composing, 202-203
 - DefaultIfEmpty, 198, 203-204
 - ElementAt, 198, 202
 - ElementAtOrDefault, 198
 - First, 198
 - FirstOrDefault, 198-199
 - Last, 198-200
 - LastOrDefault, 198-200
 - Single, 198, 201
 - SingleOrDefault, 198
- Equality, 177
- explicit conversion, 413
- famous Romans code reuse example, 178
- foreach loop code reuse example, 178
- Generation, 177, 181
 - All, 181, 187
 - Any, 181, 186-187
 - Contains, 181, 188-190
 - Empty, 181, 186
 - Range, 181-183
 - Repeat, 181-185
- Grouping, 177
- IEnumerable<T> interface support, 119
 - into, 150
 - Join, 176, 260-262
 - locating, 179-181
 - Ordering, 176, 219
 - OrderBy, 219-221
 - OrderByDescending, 219-222
 - Reverse, 219
 - Revise, 223
 - ThenBy, 219, 222-223
- overriding, 117-121
- Partitioning, 176, 192-193
 - Skip, 192-193
 - SkipWhile, 192, 196-197
 - Take, 192-193
 - TakeWhile, 192-195
- Projection, 177
- query, 46-48
- Restriction, 177
- SelectMany, projections, 159-167
- SequenceEqual, 190-192
- Set, 176, 204
 - Concat, 205
 - Distinct, 204, 207
 - Except, 205-207
 - Intersect, 204-207
- query expressions, 208-210
- SequenceEqual, 205
- Union, 204-206
- web site, 536
- Where
 - filtering nodes, 34
 - implementing, 119-120
- optimistic concurrency, 292
 - conflict detection, 292-295
 - conflict resolution, 295-297
 - limitations, 478
- optimizing performance, 482-484
- Order class
 - EntityRef type, 237
 - relationship with Customer class, 263
 - Storage property, 237
- OrderByDescending operator, 219-222
- OrderBy operator, 219-221
- Ordering operators, 176, 219
 - OrderBy, 219-221
 - OrderByDescending, 219-222
 - Reverse, 219
 - Revise, 223
 - ThenBy, 219, 222-223
- OrdersByCustomer stored procedure
 - loading Customer.Orders, 334
 - returning entity types, 322-323

- orders form example, smart client data
 - binding, 306
- orders, moving between customers, 287
- Orders property (Customer class), 237
- OtherKey property (Customer class), 237
- overriding
 - operators, 117-121
 - SubmitChanges() method, 340

P

- Paldino, Nick, 81
- Parallel Computing team web site, 488
- Parallel LINQ. *See* PLINQ
- ParallelQuery<int>, 489
- parent nodes (XML), 407-409
- parsing
 - operations, 123-124
 - XML files, 31-34
- partial classes, adding to entity class, 343
- partial methods, 66
 - example listing, 67-68
 - rules, 68-69
- Partitioning operators, 176, 192-193
 - Skip, 192-193
 - SkipWhile, 192, 196-197
 - Take, 192-193
 - TakeWhile, 192-195
- passing anonymous types out of methods, 81
- performance
 - context, defining, 479-480
 - costs, 480-481
 - improving, 490
 - LINQ to SQL
 - compiled queries, 277-278
 - queries versus results, 275-276
 - optimizing, 482-484
- persistent classes, creating, 345-348
- persistence
 - DataContext services, 474
 - entities, 473-474
 - ignorance, 347
 - separation of concerns, 473-474
- PLINQ (Parallel LINQ), 60, 488-490
 - complexities, 494
 - FakeWeatherData program, 491-492
 - IParallelEnumerable, 489
 - LINQ query time differences, measuring, 493-494
 - Parallel Computing team web site, 488
 - ParallelQuery<int>, 489
 - performance improvements, 490

- prefixes (XML namespaces), 425
- pre-LINQ mainstream programming
 - languages versus queries, 515-516
- private setter (automatic properties), 73
- Process class, 511
- ProcessList() method, 434
- Projection operators, 177
- projections, 153
 - deferred execution, 156-159
 - new class in a select clause example, 154
 - overview, 154-156
 - retrieving, 252-254
 - SelectMany operator overloads, 164-167
 - SelectMany operators, 159-164
 - SQL, 155
 - transforming objects into XML, 155
- prop snippet, 70
- properties
 - automatic, 69-73
 - querying collections of objects, 21-22
 - backing fields, 22
 - collection, 259
 - Expression, 173
 - IsPrimaryKey (Column attribute), 234
 - Orders (Customer class), 237
 - OtherKey (Customer class), 237
 - Storage (Orders class), 237
- providers
 - finding, 536
 - LinqToTerraServer, 45
 - third-party, 44

Q

- queries
 - collections of integers, 14-20
 - collections of objects, 20-25
 - databases, 26-30
 - deferred execution, 265
 - LINQ to XML, 392-394
 - LINQ to SQL, 275-278
 - PLINQ, 490
 - SharePoint, 508
 - system processes, 510-511
 - unified method, 42-43
 - versus pre-LINQ mainstream
 - programming languages, 515-516
 - XML composition, 400-402
- query expressions
 - beginning/ending, 16
 - clauses, 131-132
 - composing, 136-143

- continuations, 141
- IEnumerable<T>, 168
- IQueryable<T>, 168, 172-173
- joins, 143-153
- lambdas, 118
- MyNumberServer class, 169-172
- nomenclature, 130
- projections, 153
 - deferred execution, 156-159
 - new class in a select clause example, 154
 - overview, 154-156
 - SelectMany operator overloads, 164-167
 - SelectMany operators, 159-164
 - transforming objects into XML, 155
- querying collections of integers, 16-18
- range variables, 133-135
- Set operators, 208-210
- translating into query methods, 118
- query methods, translating from query expressions, 118
- query operators, 46-48

R

- Range operator, 181-183
- range variables (query expressions), 133-135
 - identifiers, introducing, 133
 - introducing, 135
 - types, explicitly stating, 134
- Refactor C# key bindings, 543
- reference sources, 10, 535
- references (objects), 255
- Referer automatic property generated code, 71-72
- Refresh() method, 297
- refreshing entities, 297-299
- relational data, transforming into XML, 437-442
- relational databases, 247
- relationships
 - association, 236
 - dotting through, 258
 - ends, 351
 - entities, 256-351
 - Association attribute, 257
 - collection properties, 259
 - dotting through, 258
 - joining tables, 260-262
 - managing, 286-288
 - mapping, 264
 - loading, 333-334
 - many-to-many, 262

- objects, 236-239
 - unary, 351
- Repeat operator, 181-185
- Resolve() method, 297
- resources, 10, 535
- Restriction operators, 177
- retrieving entities, 252-254
- Reverse operator, 219
- Revise operator, 223
- Root MSDN Library web site, 535
- root nodes (XML), 371
- RSS feeds, loading to XML documents, 385
- rules, partial method example, 68-69

S

- sample sources, 10
- saving XML documents, 383, 386
- scalar UDFs, LINQ to SQL translation, 251
- scalar-valued functions, 328-329
- schema validation (XML), 451
 - documents, 452-453
 - DTD specification, 453
 - namespaces, 458-459
 - validating example, 455-457
 - XSD files, 454
- scoping extension methods, 98-103
- security
 - improving, 484-486
 - LINQ to SQL, 278-279
- select clauses
 - projections, 154-155
 - query expressions, 17, 132
- SelectMany operators, 159-167
- self-referencing relationships, 351
- separation of concerns
 - business logic, 474-476
 - persistence, 473-474
- SequenceEqual operator, 190-192, 205
- Server Explorer, 437
- Set operators, 176, 204
 - Concat, 205
 - Distinct, 204, 207
 - Except, 205-207
 - Intersect, 204-207
 - query expressions, 208-210
 - SequenceEqual, 205
 - Union, 204-206
- SharePoint. *See* LINQ to SharePoint
- ShowExcept() method, 208
- ShowIntersect() method, 206

- ShowRepeat() method, 184
 - ShowUnion() method, 205
 - SimpleJoin program
 - group joins, 147
 - inner joins, 143-145
 - left outer joins, 149
 - simultaneous changes (entities), 290-292
 - optimistic, 292
 - conflict detection, 292-295
 - conflict resolution, 295-297
 - limitations, 478
 - refreshing entities, 297-299
 - set of changes, 298
 - single node access listing, 401
 - Single operator, 198-201
 - SingleOrDefault operator, 198
 - Skip operator, 192-193
 - SkipWhile operator, 192, 196-197
 - smart client data binding, orders form
 - example, 306-310
 - binding object, choosing, 306
 - data source type, choosing, 306
 - Data Sources window, populating, 308
 - master-details form design view, 309
 - new data source, adding, 306
 - order form design view, 308
 - Smet, Bart De, 499
 - snippets, 70
 - Solution Explorer (SharePoint example), 507
 - source code web site, 519
 - SpecialString class, 97
 - SQL. *See also* LINQ to SQL
 - dynamic, 468-469
 - joins, 153
 - projections, 155-156
 - security, improving, 484-486
 - Server Express, installing, 523
 - SqlMetal, 243
 - StartsWith() method, 250
 - Storage property (Order class), 237
 - stored procedures, 319
 - CUD operations, 330-332
 - configuring on entities, 332
 - override methods, 330
 - update mapped to a method example, 331
 - EF, 362
 - adding, 363
 - mapping to methods, 363
 - parameters, mapping, 365
 - update procedure, 364
 - executing, 322
 - mapping, 320
 - relationships, loading, 333-334
 - returning
 - entity type example, 322-323
 - multiple results, 324-326
 - rows with key values example, 322
 - scalar-valued functions, 328-329
 - TVFs, 326-328
 - SubmitChanges() method, 288-290
 - overriding, 340
 - performance optimization, 483
 - Sum operator, 210, 215-216
 - syntax, 2, 10
 - system processes, querying, 510-511
 - System.Data.Linq web site, 536
 - System.Diagnostics namespace, 510-511
 - System.Linq web site, 536
 - System.Xml.Linq namespace, 32
 - System.Xml.Linq web site, 536
- ## T
- T-SQL methods, 251
 - Table attribute (entity classes, creating), 233
 - tables
 - classes to tables, mapping, 233
 - DataContext class, 234-236
 - entity classes, creating, 233-234
 - Object Relational Designer, 240-243
 - SqlMetal, 243
 - databases
 - creating, 444-447
 - declarations, 450
 - joining, 260-262
 - table-valued functions (TVFs), 326-328
 - Take operator, 192-193
 - TakeWhile operator, 192-195
 - TerraServer web site, 45
 - text node searches (XML), 405
 - ThenBy operator, 219, 222-223
 - third-party LINQ providers, 44
 - ToArray operator, 224-226
 - ToDictionary operator, 224, 227-228
 - ToList operator, 224-226
 - ToLookup operator, 224
 - ToString() method, 384
 - transactions, managing, 299-302
 - transformations, 60-61
 - LINQ to SQL query into XML, 60
 - select clause projections, 155
 - XML, 429
 - relational data into XML, 437-442
 - XHTML, creating, 431-437

- XML between formats, 430-431
- XML data into databases, 447-449
- transformativity, 8, 517
- translating LINQ to SQL, 248-250
 - base class library methods, 250
 - scalar UDFs, 251
 - T-SQL methods, 251
- troubleshooting resources, 10
- TVPs (table-valued functions), 326-328
- types
 - anonymous, 24-25
 - C# 3.0, 77-82
 - EntityRef, 237
 - IEnumerable<T>, 19-20
 - inferences, querying collections of integers, 18
 - range variables, 134
 - XElement, 33

U-V

- UDFs (user-defined functions), 251
- unary relationships, 351
- unified method for querying, 42-43
- Union operator, 204-206
- unitive LINQ, 7, 516
- units of work
 - concurrency management, 478
 - performance optimization, 483-484
- Update Wizard (EF designer), 363
- updating entities, 285-286
- using directives, 376
- Validate() method, 456
- ValidationEventArgs class, 457
- ValidationEventHandler() method, 457
- var contextual keyword, 18
- var keyword (type inferences), 78-79
- variables
 - local, 91
 - range, 133-135
- Visual Basic, XML literals, 461-463
- Visual Studio, database connections, creating, 528-530
- Visual Studio Express, 522

W

- WCF (Windows Communication Foundation), 341
- WCF DataContract attribute (entity class customizations), 341-342

- web sites
 - C#
 - Development Center, 11
 - iterators blog, 110
 - sample programs, 521
 - Calvert, Charlie blog, 10
 - code generation, 536
 - database connection blog, 528
 - EDM Generator, 359
 - Flickr, 495
 - download, 496
 - FlickrXplorer application, 496
 - Informit, 11
 - joins, 536
 - Kulkarni, Dinesh, 11
 - LINQExtender, 496
 - LINQ IQueryable Toolkit, 496
 - Microsoft Office SharePoint Server 2007 VHD, 500
 - MVC download, 496
 - .NET
 - Development, 535
 - Framework, 522, 536
 - Northwind database, 521
 - operators, 536
 - Parallel Computing team, 488
 - PLINQ complexities, 494
 - providers, finding, 536
 - references, 535
 - Root MSDN Library, 535
 - SharePoint, 499
 - source code, 519
 - SQL Server Express, 523
 - System.Data.Linq, 536
 - System.Linq, 536
 - System.Xml.Linq, 536
 - TerraServer, 45
 - Visual Studio Express, 522
- where clauses (query expressions), 17, 132
- Where operator
 - filtering nodes, 34
 - implementing, 119-120
- Windows C# key bindings, 542
- Windows Communication Foundation (WCF), 341
- wizards
 - Data Source Configuration, 306, 312-313
 - Entity Data Model, 354-358
 - LINQ to SharePoint Entity, 503-506
 - Update (EF designer), 363
- WriteFile() method, 68

X-Z

- XAttribute class, 377
- XContainer class, 388-391
- XDocument class, 378, 390
- XElement class, 33, 376
 - methods, 391
 - XDocument class, compared, 390
- XHTML, creating, 431-437
- XML. *See also* LINQ to XML
 - Ancestors() method, 408-409
 - annotations, 459-461
 - attributes, creating, 377
 - character data, 406-407
 - class hierarchy, 387-388, 422
 - content, 372
 - databases, creating programmatically, 442-443
 - data schemas, viewing, 449
 - table declarations, 450
 - tables, creating, 444-447
 - XML data, transferring, 447-449
 - declarations, 371-373
 - creating, 378-381
 - DOCTYPE, 434
 - DescendantNodes() method, 402-404
 - DescendantNodesAndSelf() method, 404
 - descendants, 397-400
 - documents
 - creating, 378, 382-383
 - example, 370
 - RSS feeds, loading, 385
 - saving to disks, 383, 386
 - elements, 373
 - casting, 397
 - creating, 375-377
 - ElementsBeforeSelf()/ElementsAfterSelf()
 - methods, 409
 - Extensions class, 391
 - files
 - creating, 34-36
 - loading, 536
 - parsing, 31-34
 - FirstFourPlanets.xml, 392-394
 - fundamentals, 370-371, 374
 - hierarchy, 372
 - IXmlLineInfo interface, 415
 - literals, 461-463
 - namespaces, 424-426
 - accidentally omitting, 436
 - default, 426-429
 - identifiers, 424
 - prefixes, 425
 - schema Validation, 458-459
 - XHTML, creating, 435
 - nodes
 - adding, 420-421
 - attributes, 420
 - deleting, 417-418
 - editing, 418-419
 - elements, 421
 - line numbers, 414, 417
 - missing, 411-414
 - parent, 407-409
 - root, 371
 - queries, 392-394, 400-402
 - schema validation, 451
 - document not well formed example, 452-453
 - document validity, 453
 - DTD specification, 453
 - namespaces, 458-459
 - validating example, 455-457
 - XSD files, 454
 - single node access, 401
 - text node searches, 405
 - transformations, 429
 - relational data into XML, 437-442
 - XHTML, creating, 431-437
 - XML between formats, 430-431
 - XML data into databases, 447-449
 - using directives, 376
 - ValidationEventArgs class, 457
 - XContainer class, 389-391
 - XDocument class compared to XElement
 - class, 390
 - XElement class, 391
 - XmlSchemaSet class, 456
 - XNamespace class, 427
 - XNode class, 389-391
 - XObject class, 392, 415
 - xmlns attribute, 425
 - XmlSchemaSet class, 456
 - XNamespace class, 427
 - XNode class, 389-391
 - XObject class, 392, 415
 - XSD files, 454

yield returns, 172