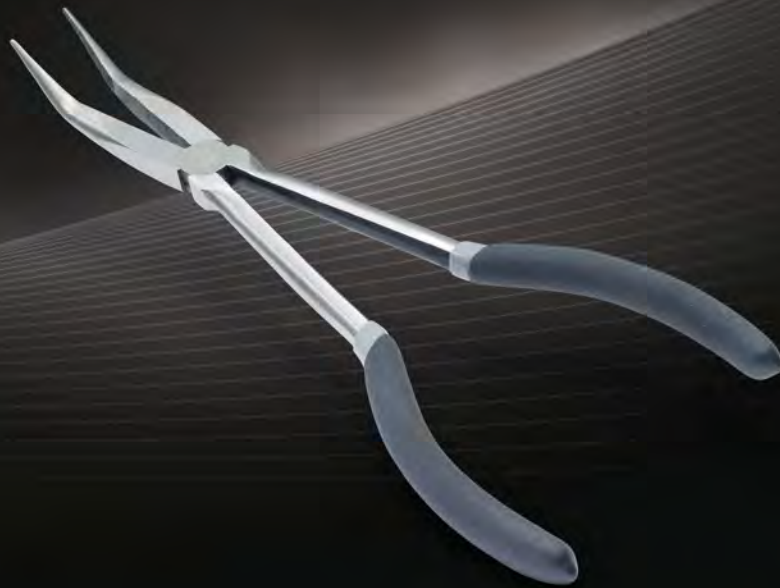


# Programming Microsoft® LINQ in Microsoft .NET Framework 4



Paolo Pialorsi  
Marco Russo

# Programming Microsoft® LINQ in Microsoft .NET Framework 4

***Paolo Pialorsi***

***Marco Russo***

Copyright © 2010 by Paolo Pialorsi and Marco Russo

Complying with all applicable copyright laws is the responsibility of the user. All rights reserved. Without limiting the rights under copyright, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without express written permission of Microsoft Press, Inc.

Printed and bound in the United States of America.

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

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

Microsoft, Microsoft Press, ActiveX, Excel, FrontPage, Internet Explorer, PowerPoint, SharePoint, Webdings, Windows, and Windows 7 are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries. Other product and company names mentioned herein may be the trademarks of their respective owners.

Unless otherwise noted, the example companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted herein are fictitious, and no association with any real company, organization, product, domain name, e-mail address, logo, person, place, or event is intended or should be inferred.

This book expresses the author's views and opinions. The information contained in this book is provided without any express, statutory, or implied warranties. Neither the author, Microsoft Corporation, nor their respective resellers or distributors, will be held liable for any damages caused or alleged to be caused either directly or indirectly by such information.

**Acquisitions and Development Editor:** Russell Jones

**Production Editor:** Adam Zaremba

**Editorial Production:** OTSI, Inc.

**Technical Reviewer:** Debbie Timmins

**Indexing:** Ron Strauss

**Cover:** Karen Montgomery

**Compositor:** Octal Publishing, Inc.

**Illustrator:** Robert Romano

978-0-735-64057-3

*To Andrea and Paola: thanks for your everyday support!*

*—Paolo*



# Contents at a Glance

Part I	<b>LINQ Foundations</b>	
1	LINQ Introduction.....	3
2	LINQ Syntax Fundamentals.....	23
3	LINQ to Objects.....	49
Part II	<b>LINQ to Relational</b>	
4	Choosing Between LINQ to SQL and LINQ to Entities.....	111
5	LINQ to SQL: Querying Data.....	119
6	LINQ to SQL: Managing Data.....	171
7	LINQ to SQL: Modeling Data and Tools.....	205
8	LINQ to Entities: Modeling Data with Entity Framework.....	241
9	LINQ to Entities: Querying Data.....	273
10	LINQ to Entities: Managing Data.....	301
11	LINQ to DataSet.....	343
Part III	<b>LINQ to XML</b>	
12	LINQ to XML: Managing the XML Infoset.....	359
13	LINQ to XML: Querying Nodes.....	385
Part IV	<b>Advanced LINQ</b>	
14	Inside Expression Trees.....	415
15	Extending LINQ.....	465
16	Parallelism and Asynchronous Processing.....	517
17	Other LINQ Implementations.....	563
Part V	<b>Applied LINQ</b>	
18	LINQ in a Multitier Solution.....	577
19	LINQ Data Binding.....	609



# Table of Contents

Preface .....	xvii
Acknowledgments .....	xix
Introduction.....	xxi

## Part I LINQ Foundations

<b>1 LINQ Introduction.....</b>	<b>3</b>
What Is LINQ?.....	3
Why Do We Need LINQ? .....	5
How LINQ Works.....	6
Relational Model vs. Hierarchical/Network Model.....	8
XML Manipulation .....	14
Language Integration.....	17
Declarative Programming.....	17
Type Checking.....	19
Transparency Across Different Type Systems.....	20
LINQ Implementations.....	20
LINQ to Objects.....	20
LINQ to ADO.NET .....	21
LINQ to XML .....	22
Summary.....	22
<b>2 LINQ Syntax Fundamentals .....</b>	<b>23</b>
LINQ Queries .....	23
Query Syntax .....	23
Full Query Syntax .....	28
Query Keywords .....	29
<i>From</i> Clause .....	29
<i>Where</i> Clause.....	32
<i>Select</i> Clause.....	32
<i>Group</i> and <i>Into</i> Clauses .....	33
<i>Orderby</i> Clause .....	35
<i>Join</i> Clause .....	36
<i>Let</i> Clause .....	40
Additional Visual Basic Keywords .....	41

---

What do you think of this book? We want to hear from you!

Microsoft is interested in hearing your feedback so we can continually improve our books and learning resources for you. To participate in a brief online survey, please visit:

[microsoft.com/learning/booksurvey](http://microsoft.com/learning/booksurvey)



Deferred Query Evaluation and Extension Method Resolution . . . . .	42
Deferred Query Evaluation . . . . .	42
Extension Method Resolution . . . . .	43
Some Final Thoughts About LINQ Queries . . . . .	45
Degenerate Query Expressions . . . . .	45
Exception Handling . . . . .	46
Summary . . . . .	48
<b>3 LINQ to Objects . . . . .</b>	<b>49</b>
Query Operators . . . . .	53
The <i>Where</i> Operator . . . . .	53
Projection Operators . . . . .	54
Ordering Operators . . . . .	58
Grouping Operators . . . . .	62
Join Operators . . . . .	66
Set Operators . . . . .	71
Aggregate Operators . . . . .	77
Aggregate Operators in Visual Basic . . . . .	86
Generation Operators . . . . .	88
Quantifier Operators . . . . .	90
Partitioning Operators . . . . .	92
Element Operators . . . . .	95
Other Operators . . . . .	100
Conversion Operators . . . . .	101
AsEnumerable . . . . .	101
ToArray and ToList . . . . .	103
ToDictionary . . . . .	104
ToLookup . . . . .	106
OfType and Cast . . . . .	107
Summary . . . . .	108
<b>Part II LINQ to Relational</b>	
<b>4 Choosing Between LINQ to SQL and LINQ to Entities . . . . .</b>	<b>111</b>
Comparison Factors . . . . .	111
When to Choose LINQ to Entities and the Entity Framework . . . . .	112
When to Choose LINQ to SQL . . . . .	114
Other Considerations . . . . .	116
Summary . . . . .	117

<b>5</b>	<b>LINQ to SQL: Querying Data</b>	<b>119</b>
	Entities in LINQ to SQL	120
	External Mapping	122
	Data Modeling	124
	DataContext	124
	Entity Classes	125
	Entity Inheritance	127
	Unique Object Identity	129
	Entity Constraints	130
	Associations Between Entities	130
	Relational Model vs. Hierarchical Model	138
	Data Querying	138
	Projections	141
	Stored Procedures and User-Defined Functions	142
	Compiled Queries	150
	Different Approaches to Querying Data	152
	Direct Queries	155
	Deferred Loading of Entities	157
	Deferred Loading of Properties	159
	Read-Only <i>DataContext</i> Access	161
	Limitations of LINQ to SQL	161
	Thinking in LINQ to SQL	163
	The <i>IN/EXISTS</i> Clause	163
	SQL Query Reduction	166
	Mixing .NET Code with SQL Queries	167
	Summary	170
<b>6</b>	<b>LINQ to SQL: Managing Data</b>	<b>171</b>
	CRUD and CUD Operations	171
	Entity Updates	172
	Database Updates	179
	Customizing <i>Insert</i> , <i>Update</i> , and <i>Delete</i>	183
	Database Interaction	185
	Concurrent Operations	185
	Transactions	189
	Exceptions	190
	Databases and Entities	192
	Entity Attributes to Maintain Valid Relationships	192
	Deriving Entity Classes	194
	Attaching Entities	197
	Binding Metadata	201
	Differences Between the .NET Framework and SQL Type Systems	204
	Summary	204

<b>7</b>	<b>LINQ to SQL: Modeling Data and Tools</b>	<b>205</b>
	File Types	205
	DBML—Database Markup Language	206
	C# and Visual Basic Source Code	207
	XML—External Mapping File	210
	LINQ to SQL File Generation	211
	SQLMetal	213
	Generating a DBML File from a Database	213
	Generating Source Code and a Mapping File from a Database	214
	Generating Source Code and a Mapping File from a DBML File	216
	Using the Object Relational Designer	216
	<i>DataContext</i> Properties	221
	Entity Class	222
	Association Between Entities	226
	Entity Inheritance	232
	Stored Procedures and User-Defined Functions	235
	Views and Schema Support	238
	Summary	239
<b>8</b>	<b>LINQ to Entities: Modeling Data with Entity Framework</b>	<b>241</b>
	The Entity Data Model	241
	Generating a Model from an Existing Database	241
	Starting from an Empty Model	244
	Generated Code	245
	Entity Data Model (.edmx) Files	248
	Associations and Foreign Keys	250
	Complex Types	254
	Inheritance and Conditional Mapping	257
	Modeling Stored Procedures	259
	Non-CUD Stored Procedures	259
	CUD Stored Procedures	262
	POCO Support	266
	T4 Templates	271
	Summary	272
<b>9</b>	<b>LINQ to Entities: Querying Data</b>	<b>273</b>
	<i>EntityClient</i> Managed Providers	273
	LINQ to Entities	275
	Selecting Single Entities	277
	Unsupported Methods and Keywords	278
	Canonical and Database Functions	279
	User-Defined Functions	281
	Stored Procedures	283

<i>ObjectQuery&lt;T&gt;</i> and <i>ObjectContext</i> .....	284
Lazy Loading .....	284
Include .....	286
<i>Load</i> and <i>IsLoaded</i> .....	288
The <i>LoadProperty</i> Method .....	288
MergeOption .....	290
The <i>ToTraceString</i> Method .....	292
<i>ExecuteStoreCommand</i> and <i>ExecuteStoreQuery</i> .....	293
The <i>Translate&lt;T&gt;</i> Method .....	294
Query Performance .....	296
Pre-Build Store Views .....	296
EnablePlanCaching .....	297
Pre-Compiled Queries .....	297
Tracking vs. No Tracking .....	299
Summary .....	299
<b>10 LINQ to Entities: Managing Data .....</b>	<b>301</b>
Managing Entities .....	301
Adding a New Entity .....	301
Updating an Entity .....	302
Deleting an Entity .....	303
Using <i>SaveChanges</i> .....	304
Cascade Add/Update/Delete .....	305
Managing Relationships .....	309
Using <i>ObjectStateManager</i> and <i>EntityState</i> .....	311
<i>DetectChanges</i> and <i>AcceptAllChanges</i> .....	313
<i>ChangeObjectState</i> and <i>ChangeRelationshipState</i> .....	314
<i>ObjectStateManagerChanged</i> .....	315
EntityKey .....	316
<i>GetObjectByKey</i> and <i>TryGetObjectByKey</i> .....	317
Managing Concurrency Conflicts .....	319
Managing Transactions .....	322
Detaching, Attaching, and Serializing Entities .....	327
Detaching Entities .....	327
Attaching Entities .....	328
<i>ApplyOriginalValues</i> and <i>ApplyCurrentValues</i> .....	330
Serializing Entities .....	333
Using Self-Tracking Entities .....	337
Summary .....	342

<b>11</b>	<b>LINQ to DataSet. . . . .</b>	<b>343</b>
	Introducing LINQ to DataSet. . . . .	343
	Using LINQ to Load a <i>DataSet</i> . . . . .	344
	Loading a <i>DataSet</i> with LINQ to SQL. . . . .	344
	Loading Data with LINQ to DataSet . . . . .	346
	Using LINQ to Query a <i>DataSet</i> . . . . .	348
	Understanding <i>DataTable.AsEnumerable</i> . . . . .	350
	Creating <i>DataView</i> Instances with LINQ. . . . .	351
	Using LINQ to Query a Typed <i>DataSet</i> . . . . .	352
	Accessing Untyped <i>DataSet</i> Data . . . . .	353
	Comparing <i>DataRow</i> Instances . . . . .	353
	Summary. . . . .	355

### Part III LINQ to XML

<b>12</b>	<b>LINQ to XML: Managing the XML Infoset. . . . .</b>	<b>359</b>
	Introducing LINQ to XML. . . . .	360
	LINQ to XML Programming. . . . .	363
	XDocument . . . . .	364
	XElement . . . . .	365
	XAttribute. . . . .	369
	XNode. . . . .	370
	<i>XName</i> and <i>XNamespace</i> . . . . .	372
	Other X* Classes . . . . .	377
	XStreamingElement . . . . .	377
	<i>XObject</i> and Annotations. . . . .	379
	Reading, Traversing, and Modifying XML. . . . .	382
	Summary. . . . .	384
<b>13</b>	<b>LINQ to XML: Querying Nodes. . . . .</b>	<b>385</b>
	Querying XML . . . . .	385
	<i>Attribute, Attributes</i> . . . . .	385
	<i>Element, Elements</i> . . . . .	386
	XPath Axes “Like” Extension Methods . . . . .	388
	<i>XNode</i> Selection Methods. . . . .	392
	InDocumentOrder. . . . .	393
	Understanding Deferred Query Evaluation . . . . .	394
	Using LINQ Queries over XML. . . . .	395
	Querying XML Efficiently to Build Entities. . . . .	397
	Transforming XML with LINQ to XML . . . . .	401
	Support for XSD and Validation of Typed Nodes . . . . .	404
	Support for XPath and <i>System.Xml.XPath</i> . . . . .	407
	Securing LINQ to XML . . . . .	409
	Serializing LINQ to XML. . . . .	410
	Summary. . . . .	412

## Part IV **Advanced LINQ**

<b>14</b>	<b>Inside Expression Trees</b> .....	<b>415</b>
	Lambda Expressions .....	415
	What Is an Expression Tree? .....	417
	Creating Expression Trees .....	418
	Encapsulation .....	420
	Immutability and Modification .....	422
	Dissecting Expression Trees .....	427
	The <i>Expression</i> Class .....	429
	Expression Tree Node Types .....	431
	Practical Nodes Guide .....	435
	Visiting an Expression Tree .....	439
	Dynamically Building an Expression Tree .....	451
	How the Compiler Generates an Expression Tree .....	451
	Combining Existing Expression Trees .....	454
	Dynamic Composition of an Expression Tree .....	459
	Summary .....	463
<b>15</b>	<b>Extending LINQ</b> .....	<b>465</b>
	Custom Operators .....	465
	Specialization of Existing Operators .....	470
	Dangerous Practices .....	473
	Limits of Specialization .....	474
	Creating a Custom LINQ Provider .....	483
	The <i>IQueryable</i> Interface .....	484
	From <i>IEnumerable</i> to <i>IQueryable</i> and Back .....	486
	Inside <i>IQueryable</i> and <i>IQueryProvider</i> .....	488
	Writing the <i>FlightQueryProvider</i> .....	491
	Summary .....	515
<b>16</b>	<b>Parallelism and Asynchronous Processing</b> .....	<b>517</b>
	Task Parallel Library .....	517
	The <i>Parallel.For</i> and <i>Parallel.ForEach</i> Methods .....	518
	The <i>Parallel.Invoke</i> Method .....	520
	The <i>Task</i> Class .....	521
	The <i>Task&lt;TResult&gt;</i> Class .....	522
	Controlling Task Execution .....	523
	Using Tasks for Asynchronous Operations .....	531
	Concurrency Considerations .....	535
	PLINQ .....	540
	Threads Used by PLINQ .....	540
	Implementing PLINQ .....	543
	Consuming the Result of a PLINQ Query .....	544

	Controlling Result Order in PLINQ . . . . .	550
	Processing Query Results . . . . .	552
	Handling Exceptions with PLINQ . . . . .	553
	Canceling a PLINQ Query . . . . .	554
	Controlling Execution of a PLINQ Query . . . . .	556
	Changes in Data During Execution . . . . .	557
	PLINQ and Other LINQ Implementations . . . . .	557
	Reactive Extensions for .NET . . . . .	559
	Summary . . . . .	561
<b>17</b>	<b>Other LINQ Implementations . . . . .</b>	<b>563</b>
	Database Access and ORM . . . . .	563
	Data Access Without a Database . . . . .	565
	LINQ to SharePoint Examples . . . . .	567
	LINQ to Services . . . . .	570
	LINQ for System Engineers . . . . .	571
	Dynamic LINQ . . . . .	572
	Other LINQ Enhancements and Tools . . . . .	572
	Summary . . . . .	574
 <b>Part V Applied LINQ</b>		
<b>18</b>	<b>LINQ in a Multitier Solution . . . . .</b>	<b>577</b>
	Characteristics of a Multitier Solution . . . . .	577
	LINQ to SQL in a Two-Tier Solution . . . . .	579
	LINQ in an n-Tier Solution . . . . .	580
	Using LINQ to SQL as a DAL Replacement . . . . .	580
	Abstracting LINQ to SQL with XML External Mapping . . . . .	581
	Using LINQ to SQL Through Real Abstraction . . . . .	584
	Using LINQ to XML as the Data Layer . . . . .	593
	Using LINQ to Entities as the Data Layer . . . . .	596
	LINQ in the Business Layer . . . . .	599
	Using LINQ to Objects to Write Better Code . . . . .	600
	<i>IQueryable&lt;T&gt;</i> vs. <i>IEnumerable&lt;T&gt;</i> . . . . .	602
	Identifying the Right Unit of Work . . . . .	606
	Handling Transactions . . . . .	606
	Concurrency and Thread Safety . . . . .	607
	Summary . . . . .	607

<b>19</b>	<b>LINQ Data Binding</b> .....	<b>609</b>
	Using LINQ with ASP.NET.....	609
	Using <i>LinqDataSource</i> .....	610
	Using <i>EntityDataSource</i> .....	625
	Binding to LINQ Queries .....	633
	Using LINQ with WPF .....	637
	Binding Single Entities and Properties .....	637
	Binding Collections of Entities .....	642
	Using LINQ with Silverlight .....	647
	Using LINQ with Windows Forms.....	652
	Summary.....	655
	<b>Index</b> .....	<b>657</b>

---

**What do you think of this book? We want to hear from you!**

Microsoft is interested in hearing your feedback so we can continually improve our books and learning resources for you. To participate in a brief online survey, please visit:

[microsoft.com/learning/booksurvey](https://microsoft.com/learning/booksurvey)





# Preface

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

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

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

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

Enjoy the reading!



# Acknowledgments

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

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

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

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

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

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



# Introduction

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

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

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

## Who Is This Book For?

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

## Organization of This Book

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

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

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

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

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

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

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

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

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

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

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

## Conventions and Features in This Book

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

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

## System Requirements

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

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

## The Companion Website

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

## Find Additional Content Online

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



## Errata & Book Support

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

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

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

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

## We Want to Hear from You

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

*<http://www.microsoft.com/learning/booksurvey>*

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

## Stay in Touch

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

# Part I

# LINQ Foundations

In this part:

Chapter 1: LINQ Introduction . . . . .	3
Chapter 2: LINQ Syntax Fundamentals . . . . .	23
Chapter 3: LINQ to Objects . . . . .	49



## Chapter 1

# LINQ Introduction

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

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

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

## What Is LINQ?

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



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

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

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

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

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

### Extending LINQ

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

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

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

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



**More Info** PLINQ is covered in Chapter 16, “Parallelism and Asynchronous Processing.”

## Why Do We Need LINQ?

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

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

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

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

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

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

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

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

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

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

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

```
Customer[] Customers;
```

Customer data could reside in a *DataTable* in a *DataSet*:

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

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

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

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

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

## How LINQ Works

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

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

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

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

The compiler generates this code:

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

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

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

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

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

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



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



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

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

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

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

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



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

## Relational Model vs. Hierarchical/Network Model

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

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

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

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

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

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

**LISTING 1-1** Type declarations with simple relationships

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

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

**LISTING 1-2** Type declarations with two-way relationships

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

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

```
Product[] products;
```

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

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

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

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

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

Data like the following will be returned:

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

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

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

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

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

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

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

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

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

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

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



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

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

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

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

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

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

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

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

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



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

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

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

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

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

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

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

```

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

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

```

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

## XML Manipulation

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

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

```

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

```

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

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

```

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

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

```

```

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

```

You can also use an XQuery to select nodes:

```

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

```

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

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

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

```

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

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

```

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



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

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

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

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

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

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

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

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



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

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



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

## Language Integration

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

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

The previous example is a simplified version of this code:

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

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

## Declarative Programming

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

In SQL, you can write the following:

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

In C#, you would probably write this:

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



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

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

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



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

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

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

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

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

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

## Type Checking

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

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

## Transparency Across Different Type Systems

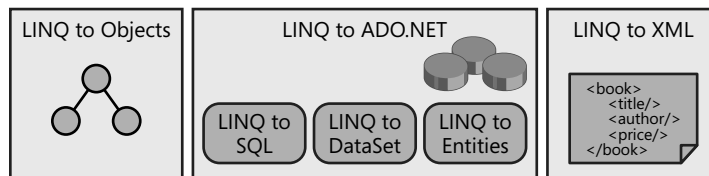
If you think about the type system of the .NET Framework and the type system of SQL Server, you will realize they are different. Using LINQ gives precedence to the .NET Framework type system, because it is the one supported by any language that hosts a LINQ query. However, most of your data will be saved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



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

## LINQ Implementations

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



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

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

### LINQ to Objects

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



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

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

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

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

## LINQ to ADO.NET

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

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

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

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



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

## LINQ to XML

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

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

This query corresponds to the following C# syntax:

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



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

## Summary

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

## Chapter 5

# LINQ to SQL: Querying Data

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

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

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

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

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



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

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

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

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



## Entities in LINQ to SQL

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



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

**LISTING 5-1** Entity definition for LINQ to SQL

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

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

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

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

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



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

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

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



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

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

#### LISTING 5-2 Simple LINQ to SQL query

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

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

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

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



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

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

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

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

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

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

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

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

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

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

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

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

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

## External Mapping

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

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

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

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



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

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

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

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



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

## Data Modeling

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



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

## DataContext

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



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

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

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

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

    public Table<Customer> Customers;
}
```



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

## Entity Classes

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

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

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



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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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



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

## Entity Inheritance

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

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

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

**LISTING 5-3** Hierarchy of classes based on contacts

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

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



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

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



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

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

**LISTING 5-4** Queries using a hierarchy of entity classes

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

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

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

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

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

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

## Unique Object Identity

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



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

**LISTING 5-5** Object identity

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

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

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

## Entity Constraints

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

## Associations Between Entities

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

## EntityRef

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

**LISTING 5-6** Association *EntityRef*

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

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

    private EntityRef<Customer> _Customer;
}
```

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



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

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

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

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

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

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



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

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

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

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

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

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

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

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

## EntitySet

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

**LISTING 5-7** Association *EntitySet* (visible)

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

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

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

**LISTING 5-8** *Association EntitySet* (hidden)

```

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

    private EntitySet<Order> _Orders;

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

```

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

```

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

```

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

```

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

```

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

```

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

```

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

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

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

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

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

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

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

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

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

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

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



```

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

```



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

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

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

```

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

```

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

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

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

```

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

```

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

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

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

## Graph Consistency

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

### Change Notification

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

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



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

## Relational Model vs. Hierarchical Model

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

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



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

## Data Querying

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

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

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

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

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

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

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

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



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

#### LISTING 5-11 Query manipulation

```
static void QueryManipulation() {  
    DataContext db = new DataContext( ConnectionString );  
    Table<Customer> Customers = db.GetTable<Customer>();  
    db.Log = Console.Out;  
  
    // All Customers  
    var query =  
        from c in Customers  
        select new {c.CompanyName, c.State, c.Country };
```

```

DisplayTop( query, 10 );

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

DisplayTop( query, 10 );

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

DisplayTop( query, 10 );
}

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

```



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

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

```

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

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

```

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

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

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

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



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

## Projections

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

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

The whole LINQ query is translated into this SQL statement:

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

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

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

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

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

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

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



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

## Stored Procedures and User-Defined Functions

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





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

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

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

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



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

The *IExecuteResult* interface has a simple definition:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

**LISTING 5-13** Stored procedure with multiple results

```

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

```

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

```

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

```

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



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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

## User-Defined Functions

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

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

**LISTING 5-14** Scalar-valued UDF

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

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

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

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

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

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

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

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

**LISTING 5-15** Table-valued UDF

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

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

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

The generated SQL query will be similar to this one:

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

## Compiled Queries

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



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

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

**LISTING 5-16** Compiled query in a local scope

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

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

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

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

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

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



**LISTING 5-17** Compiled query assigned to a static member

```

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

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

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

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

```

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

## Different Approaches to Querying Data

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

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

```

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

```

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

**LISTING 5-18** Query using *Join*

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

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

**LISTING 5-19** Query using *Association*

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

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



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

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

**LISTING 5-20** Access through *Entity*

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

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

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

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

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

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

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

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



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

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

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

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

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

## Direct Queries

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

**LISTING 5-22** Direct query

```

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

    UNION ALL

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

```

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

```

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

```

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



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

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

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

**LISTING 5-23** Direct query with parameters

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

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

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

## Deferred Loading of Entities

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

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

**LISTING 5-24** Deferred loading of entities

```

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

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

    QueryOrder(db);
}

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

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

    QueryOrder(db);
}

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

```

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

```

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

```

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

## Deferred Loading of Properties

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

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

**LISTING 5-25** Deferred loading of properties

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

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

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

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

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



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

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

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

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

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

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



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

## Read-Only *DataContext* Access

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

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

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



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

## Limitations of LINQ to SQL

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



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

## Aggregate Operators

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

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

**LISTING 5-26** Null handling with aggregate operators

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

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

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

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



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

## Partitioning Operators

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

## Element Operators

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

## String Methods

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

## DateTime Methods

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

## LIKE Operator

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

## Unsupported SQL Functionalities

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

# Thinking in LINQ to SQL

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

## The IN/EXISTS Clause

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

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

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

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

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

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

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

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

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

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

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

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

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

```

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

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

    Console.WriteLine(query);

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

```

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

```

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

```

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

```

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

```

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

```

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

```

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

```

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

```



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

## SQL Query Reduction

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



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

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

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

**LISTING 5-29** Example of query reduction

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

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

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

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

## Mixing .NET Code with SQL Queries

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

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

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

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

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

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

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



**LISTING 5-31** String manipulation extension method

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

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

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

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

```

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

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

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

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

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

```

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

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

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

**LISTING 5-34** LINQ query using native string manipulation

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

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

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

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

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

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

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

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

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

## Summary

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

# Index

## Symbols

- (decrement operator), 417
- ++ (increment operator), 417
- == operator, 36

## A

- abstract domain model, 586
- AcceptAllChanges method (ObjectContext), 313–314
- Access code-generation property, 222–240, 236
- accessing untyped DataSet data, 353
- Access property for data members of entity, 225
- access through entity (listing), 154
- Action property, 315
- Active Directory, LINQ to, 571
- AddAfterSelf method of XNode (listing), 370
- Add New Item dialog box (O/R Designer), 216
- AddObject method, 302
- AddToCustomers method, 302
- AdxStudio xRM SDK, 567
- Aggregate keyword, 41
- Aggregate operators
  - Average operator, 82–83
  - Count operator, 77
  - general-purpose, 161
  - LongCount operator, 77
  - Min/Max operators, 81–82
  - overview, 77
  - Sum operator, 78–79
  - use of, 83–86, 467
  - in Visual Basic, 86–88
- All operator, 91
- ALTER commands, 245
- Amazon, LINQ to, 570
- Amazon SimpleDB, 565
- AncestorsAndSelf extension method, 390
- Ancestors method, 388–390
- annotations, XObject and, 379–382
- anonymous delegates, defining, 26
- anonymous types, defined, 75
- Any method, 90–91
- Any operator applied to all customer orders (listing), 91

- Applied Microsoft .NET Framework Programming (Microsoft Press), 607
- ApplyCurrentValues method, 330–333
- ApplyOriginalValues method, 330–333
- ArgumentException errors, 81
- ArgumentNullException errors, 53
- ArgumentOutOfRangeException error, 98
- AsEnumerable
  - extension method, 350
  - implementation, 556
  - method, 44
  - operator, 101–103
- AsOrdered operator, 550–551
- AsParallel extension method, 540
- ASP.NET, using LINQ with, 609
- AsQueryable operator, 513
- AssociateWith<T> method, 136
- Association attribute, 130
- Association EntityRef (listing), 131
- Association EntitySet (listing), 133
- associations, foreign keys
  - and, 250–254
- asynchronous operations, tasks for, 531–535
- Asynchronous Programming Model (APM), 531
- Atom Publishing Protocol (ATOM/APP), 565
- attaching entities, 197–201, 328–330
- Attach method, 199–201
- attributes
  - attribute-based mapping, 211–212
  - entity attributes for valid relationships, 192–194
- Auto Generated Value data property, 225
- Auto-Sync data property, 225
- AutoSync parameter (Column attribute), 178–179
- Average operator, 82–83, 161

## B

- Base Class
  - code-generation property (DataContext), 222
  - Discriminator Value inheritance property, 234
- BinaryExpression class, 437
- BinaryFormatter, 333
- binding
  - collections of entities, 642–647
  - to LINQ queries, 633–637
  - metadata (databases and entities), 201–204
  - single entities and properties, 637–642
- BlockingCollection<T> class, 538
- BLToolkit, 563
- Browser, Model, 249
- BuildString method, 430
- business layer (BIZ), 577, 599–600

## C

- C# language
  - C# code equivalent to LINQ query over Reflection (listing), 19
  - code defining customer types, 49–51
  - C# query expression with left outer join (listing), 39
  - C# sample of usage of let clause (listing), 40
  - C# syntax to define default namespace and custom prefix for namespace (listing), 376
  - foreach loops, 18
  - LINQ over C#, 573
  - query expression to group developers by programming language (listing), 33
  - query expression used with exception handling (listing), 47
  - query expression with an inner join, 37
  - query expression with group join (listing), 38
  - query expression with Orderby clause (listing), 35
  - query expression with Where clause (listing), 32

- C# language (*continued*)
  - sample of usage of let clause (listing), 40
  - source code (LINQ to SQL), 207–209
- CanBeNull parameter, 126
- CancellationToken, cancelling tasks with, 528
- canonical functions (LINQ to Entities), 279–281
- Cardinality association property, 228
- Cardinality property, 231–232
- Cascade Delete, 254
- cascading deletes and updates (databases), 175–177
- CAST operator, 107–108, 204
- ChangeConflictException error, 185
- change notification for entities (LINQ to SQL), 137
- ChangeObjectState method, 314–315
- ChangeRelationshipState method, 314–315
- change tracking service, 172, 175
- Changing/Changed LINQ to XML events, 380
- Child classes, 226–227
- Child Property association property, 228
- child tasks, creating, 526
- chunking readers, 399
- classes
  - derived from
    - Expression, 434–435
    - deriving entity, 194–196
    - exception (LINQ to SQL), 190
- client-server architecture, 577
- CLR via C# (Microsoft Press), 538
- Code-Generation properties
  - for data members of entity, 225
  - for DataContext, 221–222
  - for Data function, 236
  - for Entity class, 223
- code listings
  - abstract and generic interface
    - defining Data Mapper for every entity in domain model, 588
  - abstract interface for Data Mapper acting as CRUD on Customer entity, 587
  - access through entity, 154
  - Access through Entity with single statement, 155
  - AddAfterSelf method of XNode, 370
  - adding new Customer instance to collection of Customers, 301
  - adding new Customer instance with related Orders, 305
  - Aggregate operator to get item with minimum price, 467
  - All operator applied to customer orders to check quantity, 91
  - alternative way to filter a sequence for only Visible customers, 472
  - annotations applied to an XElement instance, 379
  - Any operator applied to all customer orders, 91
  - .aspx page excerpt using editable LinqDataSource control, 619
  - .aspx page using EntityDataSource control to render list of Northwind customers into a GridView control, 626
  - .aspx page using LinqDataSource control to render list of Northwind customers into GridView control, 611
  - .aspx page using LinqDataSource control to render Northwind customers in a ListView with filtering and paging through DataPager control, 616
  - .aspx page with LinqDataSource control linked to set of user-defined entities, 625
  - AsQueryable definition
    - as extension method of FlightStatusService, 513
  - AsQueryable, effects of using, 486
  - Association EntityRef, 131
  - Association EntitySet (visible), 133
  - attach entity to DataContext, providing its original state for optimistic concurrency, 199
  - attaching entity toObjectContext, correctly handling EntityState and its OriginalValues, 331
  - attribute and child element management using XElement methods, 383
  - autogenerated code for Customer entity, 247
  - autogenerated code for model-first entity data model, which inherits fromObjectContext, 246
  - Average operator signatures
    - applied to product prices, 82
  - Average price of products ordered by customers, 87
  - BaseDal abstract base class for a LINQ to XML-based data layer, 596
  - BaseDal type implementation based on the Entity Framework, 598
  - browsing results of Descendants method invocation, 389
  - C# code equivalent to LINQ query over Reflection, 19
  - C# query expression referencing external method that throws fictitious exception, 46
  - C# query expression to group developers by programming language, 33
  - C# query expression used with exception handling, 47
  - C# query expression with a group join, 38
  - C# query expression with a left outer join, 39
  - C# query expression with an inner join, 37
  - C# query expression with Orderby clause, 35
  - C# query expression with Where clause, 32
  - C# sample of usage of let clause, 40
  - C# syntax to define default namespace and custom prefix for namespace, 376
  - calling GetCustomersByCountry business method, 604
  - Call to a lambda expression from an expression tree, 421
  - Call to external methods from an expression tree, 420
  - CancellationToken to cancel a task, 528
  - ChangeObjectState method of ObjectStateManager, 314
  - child task, creating, 526–527
  - chunked reading of XML tree using secured XmlReader, 409
  - class declaration mapped on database table with LINQ to SQL, 12
  - class entity source code in C#, 208
  - classes to remove Invoke and make parameter substitution, 457
  - class of a page reading RSS feed using a LINQ to XML query, 633

- class of page explicitly querying Northwind's customers via LINQ to SQL, 634
- class of page updating Northwind customer instance with LINQ to SQL, 637
- Click event of Button element, 641
- compiled query assigned to static member, 152
- compiled query in local scope, 151
- ComplexBinding XAML, 645
- Concat operator to concatenate Italian customers with US customers, 100
- concurrency conflicts, how to manage, 321
- console consumer application code excerpt, 342
- Contact type definition, 638
- Contains operator applied to first customer's orders, 92
- Contains operator to get an EXISTS/IN equivalent statement, 164
- Contains operator with list of constants, 165
- ContinueWhenAll method, 525
- ContinueWith method, use of, 524, 524–525
- Count operator applied to customer order, 77
- Custom aggregate function to calculate standard deviation of set of Double values, 87
- custom CUD operations for Contact entity, 265
- Customer and Order type definitions, 397
- Customer and Order types defined in independent domain mode, 585
- CustomerDal implementation using the Entity Framework under the covers, 597
- customer entity modified, 199
- CustomerManager type providing a property of type List<Customer>, 624
- Customers and average order amounts, 83
- Customers and orders paired with month of execution, 85
- Customers most expensive orders, 84
- Customer type based on external XML mapping, 584
- customizing DataContext of a LinqDataSource control, 622
- custom ObjectContext type to bind custom entities with Entity Framework, 268
- custom ObjectContext using DomainModel entities, 597
- custom ObjectContext with POCO entities, 268
- custom operator example, entities used in, 465
- custom selection pattern for EntityDataSource control using explicit LINQ query, 632
- custom selection pattern for LinqDataSource control using explicit LINQ query, 623
- custom Where extension method defined for type Customers, 102
- database update calling SaveChanges method of Entity Framework, 14
- database update calling SubmitChanges method of LINQ to SQL, 13
- DataLoadOptions and AssociateWith<T> method, 136
- DataLoadOptions and LoadWith<T>, 135
- DataSet relationships in LINQ queries, 350
- DataGridView, creating in traditional way, 351
- DataGridView, creating using LINQ query over a DataTable, 351
- DBML file excerpt, 206
- DDL defining CUD stored procedures to manage Customers, 262
- DDL defining GetCustomersByAddress stored procedure, 260
- Declaration of Inc as anonymous method, 415
- Declaration of Inc as lambda expression, 416
- DefaultIfEmpty operator syntax with default(T) and custom default value, 100
- DefaultRowComparer.Default as equality comparer calling Intersect operator, 354
- deferred loading of entities, 158
- deferred loading of properties, 159
- defining DataContext based on external XML mapping file, 582
- definition of XML element using DOM, 366–367
- definition of XML element using Visual Basic XML literals, 367
- degenerate query expression over a list of type Developers, 45
- delegate and lambda expression syntaxes, comparison between, 416
- deleting Customer instance, setting CustomerID of related Orders to NULL, 308
- deleting existing Customer instance, 303
- DescendantNodes method invocation, 391
- detaching an entity and attaching it to different ObjectContext, 328
- direct query, 156
- direct query with parameters, 157
- DisplayVisitor class, testing, 446–447
- DisplayVisitor specialization of ExpressionVisitor class, 445
- Distinct operator applied to query expression, 72
- Distinct operator applied to the list of products used in orders, 71
- ElementAt and ElementAtOrDefault operator syntax examples, 98
- Empty operator initializes empty set of customers, 90
- encapsulating APM calls into Task instances, 531
- encapsulating EAP calls into a Task, 533
- Entity class with binding information saved in external XML file, 201, 202
- Entity Data Model Designer displaying complex property, 255
- EntityDataSource control to render Northwind customers into a ListView control with filtering and paging through a DataPager control, 630
- entity definition for LINQ to SQL, 120
- entity inheritance at conceptual level, 258
- entity serialization, 197
- Event handler for selection change in a combo box, 649

code listings (*continued*)

- Exception handling for a PLINQ query, 553
- ExecuteFunction method to call stored procedure, 284
- ExecuteStoreCommand method of theObjectContext, 293
- explicit escaping of XML text, 368
- explicit type casting using XElement content, 368
- expression tree assignment, 453
- expression tree combination, 456, 458
- Expression tree visit based on a lambda expression approach, 449–451
- Expression visitor algorithm implemented through a lambda expression, 447–449
- ExpressionVisitor class, Visit method in, 440–442
- factorial of a number using Range operator, 89
- filtering customers by country selected in a combo box in LINQ to Entities, 654
- filtering customers by country selected in a combo box in LINQ to SQL, 654
- final implementation of part of CustomerDal type, 591
- first hypothetical implementation of part of CustomerDal type, 590
- First operator used to select first US customer, 96
- flattened list of orders made by Italian customer, 56
- Flight class definition, including related AirportInformation class, 493
- FlightQuery class implementing IQueryable<Flight>, 499
- FlightQueryParameters class definition, 503
- FlightQueryProvider class implementing IQueryProvider, 502
- FlightQueryTranslator member declarations, 504
- FlightQueryTranslator.TranslateStandardComparisons implementation, 509
- FlightQueryTranslator.VisitBinaryComparison implementation, 507
- FlightQueryTranslator.VisitBinary implementation, 506
- FlightQueryTranslator.VisitMethodCall implementation, 505
- FlightSearch methods as entry points in FlightStatusService, 492
- Foreign Key Associations between Customers and Orders NorthwindEntities nw = new NorthwindEntities(), 253
- fragment of XML file of orders, 14
- generic utility method for cloning entities, 334
- GetObjectByKey and TryGetObjectByKey methods, 318
- GroupBy operator used to group customers by Country, 64–65
- GroupJoin operator used to map products with orders, 69
- handling concurrency exception while editing a data item through a LinqDataSource control, 621
- handling custom selection pattern for a LinqDataSource control using a stored procedure, 623
- handling exceptions from child tasks, 530–531
- handling exceptions from tasks, 529
- helper IEnumerable<T> extension method, 345
- Helper methods DumpChanges and Dump, 172
- hierarchy of classes based on contacts, 127
- hypothetical implementation of BaseDal type, 589
- hypothetical “LINQ style” business method to refund customers for orders delivered late, 601
- implementation of APM using Task, 533
- Implementation of GetCustomersByCountry method in BIZ, 603
- implementation of WhereKey operator, 475–476
- Independent Associations between Customers and Orders, 251–252
- Initialization of controls at form load event, 653
- INotifyCollectionChanged interface signature and related types, 643
- INotifyPropertyChanging and INotifyPropertyChanged interface signatures and related types, 640
- Intersect and Except operators applied to products set, 75–76
- invoking GetCustomersByAddress method and method definition in ObjectContext, 261
- invoking SaveChanges correctly under a TransactionScope in distributed transaction, 325
- invoking SaveChanges under TransactionScope, 323
- IQueryable content obtained from using AsQueryable, 489
- IVisible interface and Customer class, definition of, 470–471
- joining two DataTable objects with LINQ, 349
- Join operator query expression syntax, 68
- Join operator used to map orders with products, 67
- KeyWrapper class, implementing, 476
- lambda expression and expression tree assignments, 451
- lambda expression and expression tree declarations, difference between, 419
- lambda expression assignment, 452
- lazy loading behavior of Entity Framework 4, 284
- legacy style business method to refund customers for orders delivered late, 600
- LINQ based implementation of ReadAll method of CustomerDal, 602
- LINQ queries applied to FlightStatusService, 514
- LINQ queries on a SortedDictionary, 477–478
- LINQ query calling a .NET Framework method in a where predicate, 170
- LINQ query calling .NET Framework method in the projection, 168
- LINQ query combining native and custom string manipulation, 168
- LINQ query merging LINQ to XML and LINQ to Objects, 396

- LINQ query over result of XPathSelectElements extension method, 409
- LINQ query over XML based on XSD typed approach, 404
- LINQ query retrieving list of temporary files larger than 10,000 bytes, ordered by size, 21
- LINQ query using native string manipulation, 169
- LINQ to Entities query that compiles but does not run on DBMS, 278
- LINQ to Entities query using canonical functions, 280
- LINQ to Entities query using custom UDF, 282
- LINQ to Entities query with filtering variable in query expression, 276
- LINQ to Object query, 540
- LINQ to SQL query, 121
- LINQ to XML and query expressions to query XML content, 395
- LINQ to XML declaration of XML namespace with custom prefix, 374
- LINQ to XML namespace declaration, 372
- LINQ to XML query based on Attributes extension methods, 386
- LINQ to XML query based on Element extension method, 386–387
- LINQ to XML query based on Elements using a namespace, 387
- LINQ to XML query for namespace against an XElement instance, 375
- LINQ to XML query that extracts city nodes from list of customers, 394
- LINQ to XML sentence merged with LINQ queries, 371
- list of orders made by Italian customers, 56
- list of Quantity and IdProduct of orders made by Italian customers, 57
- loading a DataSet by using a DataAdapter, 344
- loading both customers and their orders, 270
- loading Customer and Order instances from XML data source using a chunking reader, 399
- loading Customer and Order instances from XML data source via LINQ to XML, 398
- loading data into existing DataTable with CopyToDataTable, 348
- loading DataSet using LINQ query, 347
- loading DataSet using LINQ to SQL queries, 346
- loading Northwind DataContext with custom XML mapping file, 583
- Max operator applied to custom types, with value selector, 82
- MergeOption behavior with custom projected result sets, 292
- Microsoft Visual C# source for the XAML window, 639
- MinItem operator, definition of generic version of, 469
- MinItem operator, use of, 470
- Min operator applied to order quantities, 81
- Min operator applied to wrong types (throwing ArgumentException), 81
- MinPrice operator for Quote class, 468
- modifying an expression tree node with code that does not compile, 424
- multiple XML namespaces within a single XElement declaration, 373
- nested task, creating, 526
- NodesBeforeSelf and NodesAfterSelf, use of, 392
- NorthwindCustomersWithOrders. XML persistence file, 593
- null handling with aggregate operators, 162
- ObjectContext.ExecuteStoreQuery method, 294
- ObjectContext.LoadProperty method, 289
- ObjectContext.Translate<T> method, 294–295
- object identity, 130
- ObjectQuery<T>.Include method, 286
- ObjectQuery<T>.Load method, 288
- ObjectQuery<T>.MergeOption property, 290
- ObjectQuery<T>.ToTraceString method, 292
- ObjectStateManager, working with, 312
- OnlyVisible operator to get only Visible customers, 472
- output XML with transformed list of customers, 401
- Parallel.ForEach statement, 519
- Parallel.For statement, 518
- Parallel.Invoke statement, 520
- partial methods of an autogenerated DataContext, 580
- PLINQ query, 541
- PLINQ query adding and removing order-preservation condition, 551
- PLINQ query, cancellation of, 555
- PLINQ query using inverted processing model, 549
- PLINQ query using stop-and-go processing model, 547
- PLINQ query with lowest latency offered by using ParallelMergeOptions, 545
- PLINQ query with order preservation, 550
- PLINQ used with LINQ to XML query, 558
- POCO custom entities modified to support navigation properties between Customer and Order, 270
- pre-compiling LINQ to Entities queries returning anonymous types, 298
- pre-compiling LINQ to Entities query, 297
- ProcessFilters class, use of, 460
- product entity class declaration using Entity Framework, 13
- Products and their ordered amounts, 85
- projection with index argument in selector predicate, 55
- queries on SortedDictionary using optimized Where operator, 482–483
- queries using hierarchy of entity classes, 128
- query expression in Visual Basic, 24–27
- query expression ordered using the comparer provided by Comparer<T>.Default, 60



code listings (*continued*)

- query expression over immutable list of Customers obtained by ToList operator, 104
- query expression over list of Customers, 102–103
- query expressions with join between data sources, 30
- query expression translated into basic elements, 27
- query expression using Into clause, 35
- query expression using ToList to copy result of query over products, 104
- query expression with descending orderby clause, 58
- query expression with group by syntax, 64
- query expression with orderby and thenby, 60
- QueryFilter class definition, 494
- querying DataTable with LINQ, 349
- querying entities by using LINQ to Entities, 275
- querying typed DataSet with LINQ, 353
- querying untyped DataSet with LINQ, 353
- query manipulation, 139
- query of current processes filtered by filterExpression, 459
- query reduction example, 166
- query using Association, 153
- query using Join, 153
- query with paging restriction, 54
- query with restriction, 53
- query with restriction and index-based filter, 53
- quotes collection
  - initialization, 466
- race condition in a parallel operation, 536
- race condition, safe code without, 537
- reading entities through the Entity Framework as a list of records, 273
- reading entities through the Entity Framework as a list of typed objects, 274
- reading XML file of orders by using XmlReader, 14
- reading XML file using LINQ to XML, 15
- reading XML file using LINQ to XML and Visual Basic syntax, 16
- Read method of CustomerDal based on LINQ to XML, 594
- remote execution of LINQ query with an asynchronous completion event, 649, 651
- Repeat operator used to repeat same query, 89
- replace customer on existing orders, 176
- Retry loop for concurrency conflict, 185
- Reverse operator applied to query expression with orderby and thenby, 62
- Sample calls to FlightSearch using C# 2.0, 496
- Sample calls to FlightSearch using C# 3.0, 497
- Sample calls to FlightSearch using LINQ, 498
- sample LINQ query over set of developers, 42
- sample type using XElement, serializable with DataContractSerializer, 411
- scalar-valued UDF, 148
- Selecting event of a LinqDataSource control, 619
- self-tracking Customer entity, 338
- serialized Customer entity, 198
- serializing Customer instance using BinaryFormatter, 333
- serializing Customer instance using DataContractSerialize, 335
- serializing Customer instance using XmlSerializer, 334
- set of months generated by Range operator to filter orders, 89
- set operators applied to query expressions, 75
- Silverlight application hosted in a web page, 648
- single operator syntax examples, 97
- source XML with list of customers, 401
- specialization of Min operator for Quote instances, 473
- specialization of standard Where operator optimized for SortedDictionary, 479–481
- specialization of Where standard query operator, 471
- specialized Min operator on sequences of Quote instances, 473–474
- specialized SubmitChanges to log modified entities, 180
- specialized SubmitChanges to solve circular references of Employee entities, 182
- specialized Where for types implementing IVisible interface, 471–472
- SQL code generated by Entity Framework loading orders, 287
- SQL code generated by the Entity Framework to dynamically load orders, 285
- SQL code generated for LINQ to Entities query, 280, 282
- SQL code generated for the LINQ to Entities query, 285
- SQL code to generate custom UDF, 281
- stored procedure declaration, 143
- stored procedure to override an update, 184
- stored procedure with multiple results, 146
- string manipulation extension method, 168
- submitting changes to database, 179
- substitution in an expression tree, 425, 426
- Sum operator applied to customer orders, 78–79
- table-valued UDF, 149
- tag replacement using XElement ReplaceWith method, 383
- Take operator applied to extract top customers ordered by order amount, 93
- TakeWhile operator applied to extract top customers forming 80 percent of orders, 94
- Task class, use of, 521
- Task<TResult> class, use of, 522
- ToDictionary operator applied to customers, 105
- ToLookup operator used to group orders by product, 106
- transaction controlled by TransactionScope, 189
- transaction controlled through DataContext.Transaction property, 190
- TranslateAirportInformation Comparison and TranslateTimeSpanComparison implementation, 511
- T-SQL code generated from LINQ to Entities query, 276, 277

- type declarations with simple relationships, 9
- type declarations with two-way relationships, 10
- UDF import statement in inherited ObjectContext, 282
- Union operator applied to sets of Integer numbers, 73
- Union operator applied to sets of products, 73
- updating Customer instance with related Orders, 307
- updating existing Customer instance, 303
- updating Northwind's customer instance with LINQ to SQL, 636
- User-defined Customer and Order entities not related to a LINQ to SQL data model, 624
- user-explicit LINQ query in the Page\_Load event, 633
- using custom validation rule while updating customer information through an editable LinqDataSource control, 620
- using LinqDataSource control to render filtered, ordered list of Northwind's customers into GridView control, 614
- VisitBinary method in ExpressionVisitor class, 443–444
- VisitConstant and VisitParameter methods in ExpressionVisitor class, 443
- VisitLambda method in ExpressionVisitor class, 444–445
- VisitMethodCall and VisitInvocation methods in ExpressionVisitor class, 444–445
- VisitUnary, VisitConstant, and VisitMemberAccess implementations in FlightQueryTranslator, 512
- Visual Basic implicit join statement, 40
- Visual Basic query expression with join between data sources, 31–32
- Visual Basic XML literals, 362
- Visual Basic XML literals and global XML namespaces, 376
- Visual Basic XML literals used to declare XML content with default XML namespace, 373
- Visual Basic XML literals used to declare XML namespace with custom prefix, 375
- Visual Basic XML literal used to transform XML, 403
- XAML code for WPF window with a ListBox bound to LINQ to SQL entity set, 643
- XAML sample window with simple data-binding definition, 637
- XAML window with data binding against Northwind customer instance, 639
- XAttribute, using, 369
- XElement constructed using LINQ to XML, 366
- XElement serialization using DataContractSerializer, 411
- XML construction using DOM in C#, 360
- XML document built using LINQ to XML API, 405
- XML document illustrating searching with LINQ to XML, 388
- XML for orders, creating with Visual Basic XML literals, 16
- XML functional construction, 361
- XML mapping file excerpt, 210
- XML mapping file for Northwind DataContext, 582
- XML names, manual escaping of, 369
- XML schema definition for sample list of customers, 405
- XML tree modification events handling, 380
- XML validation using Validate extension method, 406
- XPathEvaluate extension method, 408
- XSLT transformation using XslCompiledTransform and CreateNavigator extension method, 408
- XSLT used to transform XML, 402
- XStreamingElement, bad usage of, 379
- XStreamingElement together with a chunking XmlReader, 378
- XStreamingElement, usage of, 378
- Zip operator applied to sets of Integer numbers and days of week, 76
- code, using LINQ to Objects to write, 600–601
- Collaborative Application Markup Language (CAML), 566
- collections, concurrent, 538
- collectionSelector projection, 57
- column attributes for currency control, 188–189
- column mapping, 125
- Common Table Expressions (CTEs), 155
- comparers, custom, 60, 64
- compiled data queries, 150–152
- compiled query assigned to a static member (listing), 152
- CompiledQuery.Compile static methods, 151
- compiled query in local scope (listing), 151
- Compile method, 420
- compilers, generating expression trees with, 451–454
- Compile static method, 298
- complex types (Entity Framework 4), 254–256
- Concat (concatenation) operator, 100
- Conceptual Schema Definition Language (CSDL), 248
- concurrency
  - conflicts, managing, 319–322
  - thread safety and, 607
- concurrent collections, 538
- concurrent database operations, 185–188
- ConditionalExpression class, 438–439
- conditional mapping, inheritance and, 257–259
- Configure Data Source wizard, 611–613, 626–627
- Console.WriteLine method, 102
- ConstantExpression instances, 437
- Contains extension method, 92
- Contains operator, 164
- Contains operator applied to first customer's orders (listing), 92
- ContextCreated event, 621
- ContextDisposing event, 621
- Context Namespace code-generation property (DataContext), 222
- ContinueWhenAll method, 525
- ContinueWith method, 524
- Conversion operators (LINQ to Objects)
  - AsEnumerable operator, 101–103
  - Cast operator, 107–108
  - OfType operator, 107–108
  - query evaluation and, 43

Conversion operators (LINQ to Objects) (*continued*)

- ToArray/ToList operators, 103–104
- ToDictionary extension method, 104–105
- ToLookup operator, 106–107
- CONVERT operator, 204
- CopyToDataTable<T> extension method, 346–347
- core library, LINQ, 26–27
- Count operator, 77, 161
- Count operator applied to customer order (listing), 77
- CreateDatabase method, 203
- CreateQuery method, 491, 501, 502
- CRUD and CUD operations
  - cascading deletes and updates, 175–177
  - database updates, 179–180
  - entity states, 177–178
  - entity synchronization, 178–179
  - entity updates, 172–175
  - overview, 171
  - SubmitChanges, overriding, 180–183
- CUD operations. *See* CRUD and CUD operations
- CUD stored procedures, 262–266
- curly notation, 157
- currency control, column attributes for, 188–189
- custom comparers, 60, 64
- Customers and average order amounts (listing), 83
- Customers most expensive orders (listing), 84
- custom extension methods, 41
- customizing insert/update/delete statements, 183–184
- custom LINQ provider
  - cost-benefit balance of, 514–515
  - creating, 483
- customObjectContext type with lazy loading enabled, 271–272
- custom operators, 465–470
- custom selections
  - using with EntityDataSource, 632
  - using with LinqDataSource control, 622–623
- custom types, using LinqDataSource with, 623–625

## D

data
 

- handling modifications of with LinqDataSource control, 619–622

- handling modifications with EntityDataSource, 632
- paging with DataPager/EntityDataSource controls, 630–631
- paging with LinqSource and DataPage controls, 615–619
- data access layer (DAL), 577, 580–581
- data access without databases
  - LINQ to SharePoint examples, 567–570
  - overview, 565–566
- database interaction (LINQ to SQL)
  - column attributes for currency control, 188–189
  - concurrent operations, 185–188
  - database reads, 191
  - database writes, 192–193
  - DataContext
    - construction, 190–191
    - entity manipulation, 192
    - exception classes, 190
    - transactions, 189–190
- databases
  - database access and ORM, 563–564
  - database functions (LINQ to Entities), 279–281
  - database updates, 179–180
  - generating DBML files from existing, 211
  - generating EDM from existing, 241–243
- databases and entities
  - attaching entities, 197–201
  - binding metadata, 201–204
  - creating databases from entities, 203–204
  - creating entities from databases, 203
  - deriving entity classes, 194–196
  - entity attributes for valid relationships, 192–194
- data binding, using LINQ queries for, 609
- DataContext access, read-only, 161
- DataContext class, 121, 124–125
- DataContext construction (databases), 190–191
- DataContext.GetChangeSet method, 180
- DataContext properties, 221–222
- DataContext.Transaction property, 190
- DataContract attribute, 197
- DataContract serializers, 335

- Data Description Language (DDL) files, 245
- Data function, 236–237
- Data group properties (entity class), 222
- data layers
  - data layer factory, 589
  - using LINQ to Entities as, 596–599
  - using LINQ to XML as, 593–596
- Data Mapper, defined, 586
- DataMember attribute, 197
- data modeling (LINQ to SQL)
  - DataContext class, 124–125
  - entity classes, 125–127
  - entity constraints, 130
  - entity inheritance, 127–129
  - relational vs. hierarchical model, 138
  - unique object identity, 129–130
- DataObjects.NET, 563
- DataPager control, 615–619, 630–631
- data querying (LINQ to SQL)
  - basics, 138–141
  - classic and alternative approaches to, 152–155
  - compiled queries, 150–152
  - direct queries, 155–157
  - entities, deferred loading of, 157–158
  - projections, 141–142
  - properties, deferred loading of, 159–161
  - read-only DataContext access, 161
  - stored procedures, 142–148
  - user-defined functions, 148–150
- DataRowComparer class, 354
- DataRow instances, comparing, 353–354
- DataServiceCollection class, 650
- DataSets
  - accessing untyped DataSet data, 353
  - loading with DataAdapter, 344–345
  - loading with LINQ to SQL, 344–346
  - querying with LINQ, 348–349
  - typed DataSet, querying with LINQ, 352–353
- Data Source Configuration Wizard (Visual Studio), 655
- DataTable.AsEnumerable extension method, 350
- DataGridView instances, creating with LINQ, 351–353

- DateTime methods, 163
- DBML (Database Markup Language)
  - files, 206–207
  - files, creating from scratch, 212
  - files, generating source code and mapping file from (SQLMetal), 216
  - files, loading, 123
  - generating files from existing databases, 211
  - generating file with SQLMetal, 213–214
- DbType parameter (Column attribute), 203
- Decimal type accumulator, 85
- declarative programming, 17–19
- decrement (-) operators, 417
- DeepEqual method (XNode), 371
- DefaultIfEmpty extension method, 39
- DefaultIfEmpty operator, 99
- Default Methods properties (entity class), 222–223
- deferred execution method, 122
- deferred loading
  - DeferredLoadingEnabled setting, 191
  - of entities, 157–158
  - of properties, 159–161
  - usage of, 135
- deferred query evaluation, 42–43, 394–395
- degenerate query
  - expressions, 45–46
- delegate and lambda expression syntaxes, comparing (listing), 416
- DeleteAllOnSubmit<T> method, 175
- Deleted entity state, 178
- delete operations
  - intercepting, 184
  - mapping to, 237–238
- DELETE statements
  - customizing, 183–184
  - SQL, 173–175
- deleting
  - entities, 303–304
  - event handlers, 620
- Derived Class Discriminator Value inheritance property, 234
- deriving entity classes, 194–196
- DescendantsAndSelf extension method, 390
- Descendants method, 388–390
- detaching entities, 327–328
- DetectChanges method (ObjectContext), 313–314

- dictionary, one-to-many, 106
- direct data queries, 155–157
- Discriminator inheritance property, 234
- DISTINCT clause, 71–72
- Distinct keyword, 41
- Document Object Model (DOM), 5, 359
- Double static method, 421
- DryadLINQ, 572
- DuplicateKeyException, 192
- dynamic composition of expression trees, 459–463
- dynamic LINQ, 572–573

## E

- EDM (Entity Data Model)
  - EdmGen.exe command-line tool, 241
  - .edmx files, 248–249
  - Entity Designer Database Generation Power Pack, 245
  - generated code, 245–248
  - generating from existing databases, 241–243
  - overview, 241
  - starting from empty model, 244–245
- ElementAt/ElementAtOrDefault methods, 98
- Element extension method, 386–387
- element operators, 95–100, 162
- Element property, 315
- elementSelectors, 63, 65
- ElementType property, 491
- Embedded SQL, 6
- Empty operator, 90
- EnablePlanCaching property (ObjectQuery<T> type), 297
- encapsulation (expression trees), 420–422
- entities
  - access through entity (listing), 154
  - access through entity with single statement (listing), 155
  - adding new, 301–302
  - attaching, 197–201, 328–330
  - binding collections of, 642–647
  - creating databases from, 203–204
  - creating from databases, 203
  - properties for data members of entity, 225
  - deferred loading of, 157–158
  - deleting, 303–304
  - deriving entity classes, 194–196
- detaching, 327–328
- entity attributes for valid relationships, 192–194
- entity classes, 125–127
- entity classes in O/R Designer, 222–223
- EntityClient managed providers, 273–275
- EntityCommand class, 274
- EntityConnection class, 274
- entity constraints, 130
- Entity Data Model Designer, 113
- Entity Data Model Designer (LINQ to Entities), 114
- EntityDataReader, 274
- EntityFunctions class, 279
- entity inheritance, 127–129
- EntityKey type, 316–318
- entity manipulation, 192
- entity members (O/R Designer), 224–225
- Entity Namespace code-generation property (DataContext), 222
- EntityObject base class, 112
- EntityObject Generator template (ADO.NET), 272
- entity relationships, 10
- EntitySet<T>, 605
- EntitySet<T> property, 177
- Entity SQL syntax, 274
- EntityState property, 311–313
- entity states, 177–178
- entity synchronization, 178–179
- entity updates, 172–175
- generating source code with attribute-based mapping, 211–212
- generating source code with external XML mapping file, 212
- in LINQ to SQL, 120–123
- managing relationships, 309–310
- querying XML efficiently to build entities, 397–401
- self-tracking, 337–342
- serializing, 197–198, 333–337
- tracking vs. no tracking of, 299
- updating, 302–303

- entities, associations between
  - association properties, 227–229
  - Cardinality property, 231–232
  - change notification, 137
  - entity inheritance, 232–234
  - EntityRef, 131–133
  - EntitySet, 133–137
  - fundamentals, 226–227
  - graph consistency, 137

entities, associations between  
*(continued)*  
 one-to-one  
   relationships, 229–231  
 overview, 130  
 EntityDataSource control  
   handling data modifications  
     with, 632  
   paging data with, 630–631  
   using, 625–629  
   using custom selections with, 632  
   using in combination with LINQ to  
   Entities queries, 609  
 Entity Framework. *See also* LINQ to  
 Entities  
   many-to-many relationships  
     and, 113  
   POCO support in version  
     4, 266–271  
   vs. standard ADO.NET data  
     access, 274  
 EnumerableRowCollection  
   class, 343  
 enumeration, inverted, 548–549  
 EqualityComparer<T>.Default, 92  
 EqualityComparer<TKey>.  
   Default, 64  
 equality comparisons, 36  
 equality operator, 101–102  
 equals keyword, 36  
 Equals methods, 72, 75  
 Event-based asynchronous pattern  
   (EAP), 531  
 exceptions  
   exception classes (LINQ to  
   SQL), 190  
   from child tasks, handling, 530  
   handling, 46–48  
   handling with PLINQ, 553–554  
   from tasks, handling, 529  
 Except operator, 72–75  
 ExecuteFunction method, 284  
 ExecuteMethodCall, 143  
 ExecuteQuery method, 157  
 ExecuteQuery<T> method  
   (DataContext class), 155  
 ExecuteStoreCommand/  
   ExecuteStoreQuery methods  
   (ObjectContext type), 293–295  
 explicit relationships, 10  
 Expression parameter (Column  
   attribute), 203  
 expression trees  
   anatomy of, 427–429  
   BinaryExpression class, 437  
   classes derived from  
     Expression, 434–435  
   combining existing, 454–459

  compiler generation of, 451–454  
   ConditionalExpression  
     class, 438–439  
   ConstantExpression  
     instances, 437  
   creating, 418–420  
   defined, 25, 417–418  
   dynamic composition  
     of, 459–463  
   encapsulation, 420–422  
   Expression class, 429–430  
   ExpressionType  
     enumeration, 431–434  
   immutability and modification  
     of, 422–427  
   InvocationExpression node  
     type, 439  
   LambdaExpression and  
     ParameterExpression, 436  
   lambda expressions, 415–417  
   MethodCallExpression class, 438  
   node types, 431–434  
   overview, 415  
   practical nodes guide, 435–436  
   visiting. *See* visiting expression  
   trees

  ExpressionType.Quote node, 505  
 extending LINQ, 4–5  
 extension methods  
   defined, 7  
   direct call to, 28–29  
   resolution of, 43–45  
 extensions, LINQ, 563–564  
 external mapping  
   LINQ to SQL, 122–123  
   XML file, 210, 212

## F

factory methods (Expression  
 class), 430  
 FetchXML, 566  
 Field<T> custom extension  
   methods, 349  
 file generation, LINQ to  
   SQL, 211–213  
 First method, 95–96  
 FirstOrDefault method, 96  
 Fisher, Jomo, 447, 481  
 flat IEnumerable<Order> result  
   type, 56  
 Flickr, LINQ to, 570  
 FlightQueryProvider (example)  
   FlightStatusService class, 492–499  
   implementing ExpressionVisitor in  
   FlightQueryTranslator, 503–512  
   implementing IQueryable in  
   FlightQuery, 499–501

  implementing IQueryable  
   in, 502–503  
   working with, 513–515  
   writing, 491  
 FlightQueryTranslator, 503–512  
 foreach loops, 6, 18, 139  
 Foreign Key Association, 251  
 ForeignKeyReferenceAlreadyHas  
   ValueException, 192  
 foreign keys, 9, 130, 133, 250–254  
 For method (Parallel class), 518  
 FOR XML AUTO queries, 70  
 Fowler, Martin, 180  
 from clauses  
   basics, 29–31  
   defining multiple data sources, 31  
   join clauses and, 30  
   in query expression listing, 25  
 Func declarations, 416  
 func function, 84  
 functional construction, 361  
 function attribute (UDFs), 148  
 functions, pure, 557

## G

generation operators, 88–90  
 Genom-e, 564  
 GetChangeSet method  
   (DataContext), 172  
 GetCommand method (DataContext  
   class), 122  
 GetHashCode, 72, 75  
 GetObjectByKey method, 317–319  
 GetParameterValue method, 144  
 GetResult<T> calls, 146  
 GetSchemaInfo method, 404  
 GetTable<T> method, 121  
 graph consistency (entities), 137  
 graph traversal, 152  
 GridView control, 611  
 Griffiths, Ian, 162  
 GroupBy operator, 62  
 Group clauses  
   basics, 33–35  
   ending queries with, 28  
 grouping operators, 62–66  
 GroupJoin operator, 69–71  
 group joins, 38

## H

Helper.DumpChanges method, 172  
 hierarchical/network  
   relationships, 9  
 hierarchy, entity, 195  
 Highlight method, 168  
 Huagati tools, 113

**I**

IDENTITY keyword, 126

IEnumerable interface, converting to IQueryable, 486–488

IEnumerable<TElement>, 63

IEnumerable<T> interface, 23, 29, 559, 602–605

IEnumerable<XAttribute> type, 385

IEnumerator<T> interface, 559

IGrouping<TKey, TElement> generic interface, 63

immutability/modification of expression trees, 422–427

impedance mismatch, 5

implicit associations, 153

IMultipleResults, 147

Include method
 

- ObjectQuery<T> type, 286–287
- ObjectSet<TT> class, 270

increment (++) operators, 417

Independent Associations, 250

Indexed LINQ, 573

InDocumentOrder extension method, 393

IN/EXISTS clause, 163–165

inheritance
 

- conditional mapping and, 257–259
- entity inheritance, 196, 232–234

InheritanceMapping attribute, 127

Inheritance Modifier code-generation property, 222, 223, 236

Inheritance Modifier property for data members of entity, 225

innerKeySelector, 67

InsertAllOnSubmit<T> method, 175

Inserting event handlers, 620

Insert operations
 

- intercepting, 184
- mapping to, 237–238

INSERT statements, 173–174, 183–184

intercepting insert/update/delete operations, 184

Intersect operator, 72–75

into clauses
 

- basics, 33–35
- to create new Group objects, 34

InvalidOperationException error, 97

inverted enumeration (PLINQ), 548–549

InvocationExpression node, 421, 439

Invoke node type, 457

IQueryable interface
 

- anatomy of, 488–491
- basics, 484–487
- converting to
  - IEnumerable, 486–488
- implementing in
  - FlightQuery, 499–501

IQueryable<T> interface, 23, 29, 602–605

IQueryProvider
 

- anatomy of, 488–491
- implementing in
  - FlightQueryProvider, 502–503

IsComposable argument (UDFs), 144

IsDBGenerated Boolean flag, 188

IsDBGenerated parameter, 126

IsForeignKey argument (Association attribute), 131

IsLoaded property, 288–289

is operator, 259

IsPrimaryKey property, 125

IsUniqueKey argument (Association attribute), 131

IsVersion Boolean flag, 188

iterative vs. declarative constructs, 18

**J**

JavaScript Object Notation (JSON)
 

- Json.NET 3.5 library, 571
- LINQ to, 571

Join clauses, 36–40

joining two DataTable objects with LINQ, 349

join operators, 66–71

**K**

Key property, 33

keySelector, 105

keySelector argument, 59

keySelector predicate, 63

keywords
 

- query. *See* query keywords
- unsupported by LINQ to
  - Entities, 278–279

KeyWrapper class, 476

**L**

LambdaExpression nodes, 436

lambda expressions, 26–27, 59, 67, 415–417

language integration
 

- declarative programming, 17–19
- overview, 17
- transparency across type systems, 20
- type checking, 19

Last/LastOrDefault operators, 96–97, 162

lazy loading, 268–269, 284–286

LDAP, LINQ to, 571

left outer joins, 38, 69

let clauses, 28, 40–41

LIKE operator, 163

Link<T> type, 160

LinqConnect, 564

LinqDataSource control
 

- using custom selections with, 622–623
- handling data modifications with, 619–622
- paging data with, 615–619
- using, 610–616
- using with custom types, 623–625

LINQKit library, 463

LINQ (Microsoft Language Integrated Query)
 

- basic workings of, 6–8
- in business layer, 599–600
- creating custom provider, 483
- creating DataView instances with, 351–353
- dynamic LINQ, 572–573
- enhancements and tools, 572–573
- extending, 4–5
- extensions, 563–564
- full query syntax, 28–29
- implementations of, 20–22
- language integration in.
  - See* language integration
- LINQExtender, 573
- LINQ over C#, 573
- LINQPad, 573
- LINQ to Expressions, 573
- LINQ to Geo, 573
- in n-tier solutions, 580
- operators, 350
- overview of, 3–5
- queries, binding to, 633–637
- query example, 6–7
- query expression syntax vs. SQL, 80–81
- querying DataSets with, 348–349
- querying typed DataSets with, 352–353
- query keywords. *See* query keywords

- LINQ (Microsoft Language Integrated Query) (*continued*)
  - query syntax, 23–28
  - reasons for using, 5–6
  - relational vs. hierarchical/network model, 8–14
  - for system engineers, 571
  - using with ASP.NET, 609
  - using with Silverlight, 647–652
  - using with Windows Forms, 652–655
  - using with WPF, 637
  - XML manipulation and, 14–16
- LINQ to Active Directory, 571
- LINQ to ADO.NET, 21
- LINQ to Amazon, 570
- LINQ to DataSet
  - accessing untyped DataSet data, 353
  - AsEnumerable extension method, 350
  - DataRow instances, comparing, 353–354
  - DataView instances, creating with LINQ, 351–353
  - defined, 21
  - loading DataSet with DataAdapter, 344–345
  - loading DataSet with LINQ to SQL, 344–346
  - loading data with, 346–348
  - overview, 343
  - typed DataSet, querying with LINQ, 352–353
- LINQ to Entities
  - adding new entities, 301–302
  - ApplyCurrentValues method, 330–333
  - ApplyOriginalValues method, 330–333
  - associations and foreign keys, 250–254
  - attaching entities, 328–330
  - canonical and database functions, 279–281
  - cascade add/update/delete, 305–309
  - choosing LINQ to SQL over, 114–116
  - comparing to LINQ to SQL, 116–117
  - complex types, 254–256
  - concurrency conflicts, managing, 319–322
  - defined, 21
  - deleting entities, 303–304
  - detaching entities, 327–328
  - EntityClient managed providers, 273–275
  - Entity Data Model (EDM). See EDM (Entity Data Model)
  - See EDM (Entity Data Model)
  - factors for comparing with LINQ to SQL, 111
  - inheritance and conditional mapping, 257–259
  - managing relationships, 309–310
  - modeling stored procedures. See stored procedures, modeling
  - ObjectStateManager, 311–313
  - POCO support in Entity Framework 4, 266–271
  - querying entities of EDMs with, 275–277
  - query performance, 296–298
  - SaveChanges method, 304–305
  - self-tracking entities, 337–342
  - serializing entities, 333–337
  - single entities, selecting, 277–278
  - stored procedures, 283–284
  - T4 templates, 271–272
  - transactions, managing, 322–327
  - unsupported methods and keywords, 278–279
  - updating entities, 302–303
  - user-defined functions, 281–282
  - using as data layer, 596–599
  - when to choose over LINQ to SQL, 112–114
- LINQ to Exchange, 4
- LINQ to Flickr, 570
- LINQ to Google
  - implementation, 570
- LINQ to JSON, 571
- LINQ to LDAP, 4, 571
- LINQ to Objects
  - basics, 20
  - Conversion operators. See Conversion operators (LINQ to Objects)
  - example, 23–28
  - query operators. See query operators (LINQ to Objects)
  - sample data for examples, 49–91
  - using to write better code, 600–601
- LINQ to services, 570–571
- LINQ to SharePoint
  - (examples), 567–570
- LINQ to SQL
  - abstracting with XML external mapping, 581–584
  - choosing LINQ to Entities over, 112–114
  - comparing to Entity Framework, 116–117
  - CRUD and CUD operations. See CRUD and CUD operations
  - See CRUD and CUD operations
  - customizing insert/update/delete statements, 183–184
  - as DAL replacement, 580–581
  - database interaction. See database interaction (LINQ to SQL)
  - database model schema (Northwind), 610
  - databases and entities. See databases and entities
  - data modeling. See data modeling (LINQ to SQL)
  - data querying. See data querying (LINQ to SQL)
  - DBML files, 206–207
  - defined, 21
  - entities in, 120–123
  - external mapping, 122–123
  - factors for comparing with LINQ to Entities, 111
  - file generation, 211–213
  - file types for creating entities, 205–206
  - limitations of, 161–163
  - LinqToSqlMapping.xsd schema file, 210
  - loading DataSets with, 344–346
  - O/R Designer. See O/R (Object Relational) Designer
  - overview, 119
  - simple query (listing), 121
  - source code (C# & Visual Basic), 207–209
  - SQLMetal. See SQLMetal
  - SQL queries, mixing .NET code with, 167–170
  - SQL query reduction, 166–167
  - support of T-SQL statements, 117
  - thinking in, 163–166
  - in two-tier solutions, 579–580
  - using through real abstraction, 584–593
  - when to choose over Entity Framework, 114–116
  - XML external mapping file, 210
- LINQ to Streams
  - implementation, 572
- LINQ to WMI, 571
- LINQ to XML. See *also* XML (Extensible Markup Language)
  - basics, 22
  - deferred query evaluation, 394–395
  - events, 380

- introduction to, 360–363
- LINQ query expressions, using
  - over XML nodes, 395–401
- LINQ to XML namespace
  - declaration (listing), 372
- LINQ to XML query for
  - namespace against an XElement instance (listing), 375
- LINQ to XML sentence merged
  - with LINQ queries (listing), 371
- LINQ to XML sentence merged
  - with LINQ queries, using Visual Basic XML literals (listing), 371
- and query expressions to query
  - XML content (listing), 395
- query based on Attributes
  - extension methods (listing), 386
- query based on Element
  - extension method (listing), 386
- querying XML. *See* XML, querying
- query that extracts city nodes
  - from list of customers (listing), 394
- reading XML file using
  - (listing), 15
- securing, 409–410
- serializing, 410–411
- support for XPath/System.Xml.XPath, 407–409
- support for XSD, 404–407
- transforming XML with, 401–404
- using as data layer, 593–596
- using in Microsoft Visual Basic
  - syntax, 15
- validation of typed
  - nodes, 404–407
- LINQ to XML programming
  - framework
    - basics, 363–364
    - XAttribute class, 369
    - XComment class, 377
    - XDeclaration class, 377
    - XDocument class, 364–365
    - XDocumentType class, 377
    - XElement class, 365–369
    - XName and XNamespace classes, 372–377
    - XNode class, 370–371
    - XObject class, 379–382
    - XProcessingInstruction class, 377
    - XStreamingElement class, 377–379
    - XText class, 377
  - listings, code. *See* code listings
  - list of orders made by Italian customers (listing), 56

- list of Quantity and IdProduct
  - of orders made by Italian customers (listing), 57–58
- literals
  - Visual Basic XML, 16
  - XML, 362
- LLBLGen Pro runtime, 564
- loading
  - Customer and Order instances
    - from XML data source using a chunking reader (listing), 399
  - data into existing DataTable with CopyToDataTable (listing), 348
  - DataSets with
    - DataAdapter, 344–345
  - DataSet with LINQ to
    - SQL, 344–346
  - data with LINQ to
    - DataSet, 346–348
  - Load method, 288–289, 367
  - LoadOption enumeration, 348
  - LoadProperty method, 288–289
- logical operators, 32–33
- LongCount operator, 77, 161
- Lucene Information Retrieval
  - System, 573

## M

- many-to-many relationships, 113
- mapping
  - conceptual schema to storage
    - schema, 248
  - conditional, 257–259
  - files, external XML, 201–202
  - files, generating with
    - SQLMetal, 214–215
  - to Delete/Insert/Update
    - operations, 237–238
- Max operator, 82, 161
- members, entity, 224–225
- MergeOption property
  - (ObjectQuery<T> type), 290–292
- metadata, binding (databases and entities), 201–204
- MetaLinq, 573
- methods
  - MethodCallExpression class, 438
  - MethodCallExpression node, 505
  - method syntax, 41
  - unsupported by LINQ to
    - Entities, 278–279
- Microsoft Developer Network, 3
- Microsoft Dynamics CRM, 566
- Microsoft Excel, 566–567
- Microsoft SharePoint, 566

- Microsoft SharePoint 2010
  - Developer Reference (Microsoft Press), 570
- Microsoft SharePoint
  - Foundation, 566
- Microsoft SQL Server Data Services (SSDS), 565
- Microsoft Visual C# filter
  - condition, 136
- Microsoft Word documents,
  - querying, 566
- MidpointRounding
  - enumeration, 204
- MinItem operator, 468–469
- Min/Max operators, 81–82, 161, 467
- MinPrice operator, 468
- mixed query syntax, 41
- modeling stored procedures.
  - See* stored procedures, modeling
- MonthComparer, custom, 61
- MSVS folder, 205
- multiple From clauses, 32
- multiple-value keys, 33
- multithreaded environment, safe
  - programming in, 535
- multitier solutions, characteristics
  - of, 577–579

## N

- Name code-generation property
  - DataContext, 222
  - defined, 236
  - entity class, 223
- Name property for data members
  - of entity, 225
- native extension methods, 41
- nested tasks, creating, 526
- .NET Framework
  - code, mixing with SQL
    - queries, 167–170
  - ExpressionVisitor differences
    - between versions 3.5 and 4.0, 440
  - Math.Round method, 204
  - .NET Framework 4 Windows
    - Workflow Foundation, 245
  - .NET Framework 2.0 System.Xml
    - classes, 14
  - Reactive Extensions (Rx)
    - for, 559–560
    - type system, 20
    - vs. SL type systems, 204
- NHibernate, 564



## nodes

in expression trees. *See* expression trees

NodesBeforeSelf/NodesAfterSelf methods, 392

## non-CUD stored

procedures, 259–262

non-updatable views, 125

Northwind database, 205, 579

NOT EXISTS clause, 164

NOT IN clause, 163

n-tier solutions, LINQ in, 580

Nullable data property, 225

nullable types, 78, 82, 156

null handling with aggregate operators (listing), 162

Numeric types, 78–82

## O

ObjectChangeTracker class, 340

ObjectContext class, 246

ObjectContext.SaveChanges method, 304

ObjectContext type

ExecuteStoreCommand/

ExecuteStoreQuery

methods, 293–295

Translate<T> method, 294–295

object identity (listing), 130

object initializers, 142

ObjectQuery<T> type

EnablePlanCaching property, 297

Include method, 286–287

lazy loading, 284–286

Load method, 288–289

LoadProperty method, 288–289

MergeOption property, 290–292

ToTraceString method, 292–293

Object Relational Designer (Visual Studio), 183, 196

Object Relational Mapping (ORM), 21, 250

ObjectStateManager, 311–313

ObjectStateManagerChanged event, 315

object tracking, disabling, 161

ObjectTrackingEnabled

property, 161

OfType operator, 107–108

one-to-many dictionary, 106

one-to-many relationships, 130

one-to-one relationships, 229–231

OnlyVisible operator, 472

Open Database Connectivity (ODBC), 5

Open Data (OData) Protocol, 565

## operators

partitioning, 92, 162

projection, 54–58

quantifier, 90–92

query. *See* query operators (LINQ to Objects)

operators, custom, 465–470. *See also* individual operators

operators, specialization of existing

alternative way to filter a sequence for only Visible

customers (listing), 472

dangerous practices, 473–474

implementation of WhereKey operator (listing), 475–476

IVisible interface and

Customer class, definition of (listing), 470–471

KeyWrapper class, implementing (listing), 476

limits of specialization, 474–482

LINQ queries on a

SortedDictionary

(listing), 477–478

OnlyVisible operator to get only Visible customers (listing), 472

queries on SortedDictionary using optimized Where operator

(listing), 482–483

specialization of Min operator for Quote instances (listing), 473

specialization of standard

Where operator optimized

for SortedDictionary

(listing), 479–481

specialization of Where standard query operator (listing), 471

specialized Min operator on sequences of Quote instances

(listing), 473–474

specialized Where for types

implementing IVisible interface (listing), 471–472

OptimisticConcurrency

Exception, 330

orderby clauses

basics, 35–36

usage example, 29

OrderByDescending extension

method, 59

orderby keyword, 58

OrderBy/OrderByDescending

operators, 58–59, 550

ordering operators, 58–61

ORM (Object Relational Model), 563–564

O/R (Object Relational) Designer association between entities.

*See* entities, associations between

creating hierarchy of classes

in, 232–233

DataContext properties, 221–222

entity classes in, 222–223

entity members, 224–225

functions of, 207

fundamentals, 216–220

stored procedures, 235–238

user-defined functions, 235–238

views and schema support, 238

OUTER APPLY join, 153

outerKeySelector predicate, 67

OverflowException errors, 82

overriding

SubmitChanges, 180–183

## P

pagination data

with EntityDataSource and

DataPage, 630–631

with LinqDataSource and

DataPage controls, 615–619

at persistence layer level, 54

ParallelEnumerable class, 544

Parallel.For and Parallel.ForEach

methods, 518–520

Parallel.Invoke method, 520–521

ParameterExpression class, 436

parameterized constructors, 142

parent/child relationships, 9

Parent classes, 226–227

Parent Property association

properties, 228

Parse static method, 367

Participating association

property, 228

partitioning operators, 92–95, 162

“persistence ignorance” approach to

modeling, 244

Pialorsi, Paolo, 570

PingTask static function, 534

pipelined processing

(PLINQ), 545–547

PIVOT command, 155

placeholders notation, 157

PLINQO, 113

PLINQ (Parallel LINQ)

changes in data during query

execution, 557–558

controlling result order

in, 550–552

handling exceptions

with, 553–554

- implementing, 543–544
- inverted enumeration, 548–549
- overview, 540
- pipelined processing, 545–547
- processing query results, 552–553
- queries, cancellation of, 554–555
- queries, consuming result of, 544–550
- queries, controlling execution of, 556–557
- stop-and-go processing, 547–548
- threads used by, 540–543
- using with other LINQ providers, 557–559
- POCO
  - Entity Generator template (ADO.NET), 272
  - support for, 112
  - support in Entity Framework 4, 266–271
- PossiblyModified entity state, 178
- pre-building store views, 296
- pre-compiled queries, 297–299
- presentation layer, 577
- Primary Key data property, 225
- primary keys, 130, 133
- ProcessFilters class, 460
- projection operators, 54–58
- projections (data querying), 141–142
- properties
  - association, 227–229
  - binding, 637–642
  - for data members of entities, 225
  - Code-Generation, 236
  - DataContext, 221–222
  - deferred loading of, 159–161
  - entity class, 222
  - inheritance, 234
- PropertyChanging method (DataContext), 137
- pure functions, defined, 557

## Q

- quantifier operators, 90–92
- queries
  - associative, 152
  - changes in data during execution of (PLINQ), 557–558
  - compiled query assigned to a static member (listing), 152
  - compiled query in local scope (listing), 150
  - compiling, 298
  - direct, 155–157

- LINQ query calling a .NET Framework method in a where predicate (listing), 170
- LINQ query calling .NET Framework method in the projection (listing), 168
- LINQ query combining native and custom string manipulation (listing), 168
- LINQ query using native string manipulation, 169
- PLINQ, cancellation of, 554–555
- PLINQ, consuming result of, 544–550
- PLINQ, controlling execution of, 556–557
- pre-compiled, 297–299
- processing results of (PLINQ), 552–553
- query evaluation, deferred, 42–43
- query manipulation (listing), 139
- queryTyped/queryFiltered, 129
- query using Association, 153
- query using Join (listing), 153
- query with paging restriction (listing), 54
- query with restriction and index-based filter (listing), 53
- query with restriction (listing), 53
- SQL, mixing .NET with, 167–170
- SQL query reduction, 166–167
- SQL query reduction example (listing), 166
- using hierarchy of entity classes (listing), 128
- query expressions
  - in C#, 24
  - deferred query evaluation, 42–43
  - defined, 25
  - degenerate, 45–46
  - with descending orderby clause (listing), 58
- Distinct operator applied to (listing), 72
- extension method
  - resolution, 43–45
- with group by syntax (listing), 64
- with group join (listing), 38
- with inner join (listing), 37
- using Into clause (listing), 35
- with join between data sources (listing), 30–32
- with join ... into clause (listing), 70
- with left outer join (listing), 39
- with orderby and thenby (listing), 60
- with orderby clause with multiple ordering conditions (listing), 36

## query operators (LINQ to Objects)

- with Orderby clause (listing), 35
- ordered using comparer provided by Comparer<T>.Default (listing), 60
- over immutable list of Customers obtained by ToList operator (listing), 104
- over list of Customers converted with AsEnumerable operator (listing), 103
- over list of Customers (listing), 102
- Set operators applied to, 75
- SQL vs. LINQ query expression syntax, 80
- syntax of Select operator, 55
- to group developers by programming language (listing), 33–34
- translated into basic elements (listing), 27
- used with exception handling (listing), 47
- using ToList to copy result of query over products, 104
- in Visual Basic (listing), 24–27
- with Where clause (listing), 32
- querying
  - DataSets with LINQ, 348–349
  - data table with LINQ, 349
  - typed DataSets with LINQ, 352–353
  - untyped DataSet with LINQ (listing), 353
  - XML efficiently to build entities, 397–401
- query keywords
  - Aggregate keyword, 41
  - Distinct keyword, 41
  - from clauses, 29–31
  - group clauses, 33–35
  - into clauses, 33–35
  - join clauses, 36–40
  - let clauses, 40–41
  - orderby clauses, 35–36
  - select clauses, 32
  - Skip keyword, 41
  - Skip While keyword, 41
  - Take keyword, 41
  - Take While, 41
  - where clauses, 32
- query operators (LINQ to Objects)
  - aggregate operators, 77–86
  - aggregate operators in Visual Basic, 86–88
  - Concat (concatenation) operator, 100
  - element operators, 95–100

query operators (LINQ to Objects)  
*(continued)*  
 generation operators, 88–90  
 grouping operators, 62–66  
 join operators, 66–71  
 ordering operators, 58–62  
 partitioning operators, 92–95  
 projection operators, 54–58  
 quantifier operators, 90–92  
 SequenceEqual extension  
 method, 101  
 set operators, 71–77  
 Where operator, 53–54  
 query syntax  
 full, 28–29  
 simple example, 23–28

## R

race conditions, 536–537  
 Range operator, 88–89  
 range variables, 29  
 Reactive Extensions (Rx) for  
 .NET, 559–560  
 Read method implementation, 595  
 read-only DataContext access, 161  
 Read Only data property, 225  
 reads, database, 191  
 real object-oriented  
 abstraction, 584–593  
 recursive lambda expressions, 417  
 reference type vs. value type  
 semantic, 75–76  
 Refresh method, 129, 186–187, 322  
 RefreshMode enumeration, 187  
 Register method, 528  
 relational databases  
 LINQ to SQL entities and, 192  
 relationships between file  
 types, 211  
 relational vs. hierarchical model  
 (LINQ to SQL), 8–14, 138  
 relationships (entities),  
 managing, 309–310  
 remote data storage, 565  
 Remove method  
 (XElement), 383–384  
 Repeat operator, 89–90  
 ReplaceWith method  
 (XElement), 383–384  
 resolution, extension  
 method, 43–45  
 Restriction operators, 53, 90  
 result order, controlling in  
 PLINQ, 550–552  
 resultSelector predicate, 84  
 resultSelector projection, 57  
 resultSelectors, 65  
 ResultType attributes, 146  
 ReturnValue property, 236–237  
 ReturnValue read-only  
 property, 144  
 reverse operator, 61–62  
 Richter, Jeffrey, 538, 607  
 RIGHT OUTER JOIN, 69

## S

SampleSiteDataContext class, 567  
 SaveChanges method, 304–305  
 Save method overloads, 367  
 SaveOptions method, 304  
 scalar-valued UDF (listing), 148  
 schemas  
 O/R Designer and, 238  
 XML, 403  
 securing LINQ to XML, 409–410  
 Select clauses  
 basics, 32  
 ending queries with, 28  
 SelectMany operator, 56–59  
 Select method, 26  
 Select operator, 54–55  
 SELECT SQL statements, 171  
 self-tracking entities, 337–342  
 Self-Tracking Entity Generator  
 template (ADO.NET), 272  
 SequenceEqual extension  
 method, 101  
 serializing  
 LINQ to XML, 410–411  
 Serialization Mode code-  
 generation property  
 (DataContext), 222  
 serialization of entities, 197–198,  
 333–337  
 Server Data Type data  
 property, 225  
 services, LINQ to, 570–571  
 set operators, 71–77  
 SharePoint, LINQ to  
 (examples), 567–570  
 Show Only DataContext Objects  
 check box, 612  
 Silverlight, using LINQ  
 with, 647–652  
 SimpleDB, Amazon, 565  
 Simple Object Access Protocol  
 (SOAP) services, 359  
 single entities, binding, 637–642  
 Single operator, 97–98, 141, 152,  
 153  
 SingleOrDefault operator, 97–98  
 single-value keys, 33  
 Skip and Skip While keywords, 41  
 Skip and SkipWhile operators, 95,  
 162  
 SoapFormatter, 334  
 SortedDictionary, creating, 475  
 source code  
 C# and Visual Basic, 207–209  
 entity, generating with attribute-  
 based mapping, 211–212  
 entity, generating with external  
 XML mapping file, 212  
 generating with  
 SQLMetal, 214–215  
 Source data property, 225  
 SPMetal.exe, 566  
 SQLMetal  
 generating database DBML file  
 with, 213–214  
 generating source code and  
 database mapping file  
 with, 214–215  
 generating source code and  
 mapping file from DBML  
 file, 216  
 overview, 213  
 /serialization:unidirectional  
 parameter in, 197  
 tool, 137  
 SQL Servers  
 parameterized queries to, 150  
 type system, 20  
 SQL (Structured Query Language)  
 Embedded SQL, 7  
 queries, mixing .NET code  
 with, 167–170  
 query reduction, 166–167  
 ROUND operator, 204  
 SQLCLR stored procedure, 4  
 SqlConnection class, 121  
 SqlFunctions class, 279  
 SQL vs. LINQ queries, 12  
 SQL vs. LINQ query expression  
 syntax, 80  
 states, entity, 177–178  
 static methods, 45, 430  
 STDDEV aggregation, 163  
 Stock and Quote example (custom  
 operators), 465  
 stop-and-go processing  
 (PLINQ), 547–548  
 Storage argument (Association  
 attribute), 133  
 Storage parameters, 126  
 Storage Schema Definition  
 Language (SSDL), 248  
 stored procedures  
 example, 183–184  
 LINQ to Entities and, 283–284  
 with multiple results (listing), 146

- O/R Designer and, 235–238
- stored procedure declaration (listing), 143
- user-defined functions and, 142–148
- stored procedures, modeling
  - CUD stored procedures, 262–266
  - non-CUD stored procedures, 259–262
  - overview, 259
- string manipulation extension
  - method (listing), 168
- String type methods (.NET Framework), 162
- subexpressions, 40
- SubmitChanges method, 172, 179–180, 568
- SubmitChanges, overriding, 180–183
- SubSonic, 564
- SUM aggregation operation, 155
- Sum operator, 78–79, 161
- synchronization
  - of entities, 178–179
  - of tasks, 538–539
- SyncLINQ implementation, 572–573
- syntactic sugar, 453
- syntax, query. *See* query syntax
- system engineers, LINQ for, 571
- System.Collections and System.Collections.Generic namespaces, 538
- System.Collections.Concurrent namespace, 538
- System.Data.DataSet, 343
- System.Data.DataSetExtensions assembly, 343
- System.Data.Linq assembly, 120
- System.Data.Linq
  - Mapping.AutoSync enumeration, 178–179
- System.Data.Linq.SqlClient namespace, 161
- System.Data namespace, 354
- System.Diagnostics.Process property, 460
- System.IComparable interface, 59
- System.Json namespace, 571
- System.Linq.Expressions namespace, 418
- System.Linq namespace, 20, 26, 53
- System.Messaging.dll assembly, 324
- System.Runtime.Serialization namespace, 197
- System.Threading.Tasks namespace, 518
- System.Threading.Tasks.Task class, 521

- System.Xml.Linq.Extensions class, 385
- System.Xml.Linq framework, 374
- System.Xml.Linq namespace, 364
- System.Xml.Schema.Extensions class, 404
- System.Xml.XPath.Extensions class, 407
- System.Xml.XPath, support for, 407–409

## T

- T4 templates, 271–272
- Table<T> generic class, 120
- Table<T> methods, 192
- table-valued UDF (listing), 149
- tag replacement using XElement
  - ReplaceWith method (listing), 383
- Take keyword, 41
- Take operator, 93, 506
- Take While keyword, 41
- TakeWhile operator, 94, 162
- Task Parallel Library (TPL)
  - CancellationToken to cancel a task (listing), 528
  - child task, creating (listing), 526–527
  - concurrency
    - considerations, 535–536
  - concurrent collections, 538
  - ContinueWhenAll method (listing), 525
  - ContinueWith method, use of (listing), 524–525
  - controlling task execution, 523
  - handling exceptions from child tasks (listing), 530–531
  - handling exceptions from tasks (listing), 529
  - nested task, creating (listing), 526
  - overview, 517–518
  - Parallel.For and Parallel.ForEach methods, 518–520
  - Parallel.Invoke method, 520–521
  - race conditions, 536–537
  - safe programming
    - in multithreaded environments, 535
  - Task class, 521–522
  - TaskScheduler class, 539
  - tasks for asynchronous operations, 531–535
  - task synchronization, 538–539
  - Task<TResult> class, 522–523
- Task.WaitAll method, 521
- Telerik OpenAccess, 564
- TEntity type, 588
- ThenBy/ThenByDescending operators, 59–61
- thread safety, concurrency and, 607
- ThreadStatic attribute, 606
- threads used by PLINQ, 540–543
- Time Stamp data property, 225
- TInner type joins, 67
- ToArray/ToList operators, 103–104
- ToBeDeleted entity state, 178
- ToBeInserted entity state, 178
- ToBeUpdated entity state, 178
- ToDictionary extension
  - method., 104–105
- ToList method, 8
- ToLookup operator, 106–107
- Torgersen, Mads, 417
- ToString method, 122, 204
- ToString override, 501
- ToTraceString method (ObjectQuery<T> type), 292–293
- ToUpper method, 142
- TOuter type joins, 67
- tracking vs. no tracking (entities), 299
- transactions
  - database, 189–190
  - handling in n-tier architectures, 606
  - managing, 322–327
- Translate<T> method (ObjectContext type), 294–295
- try ... catch blocks
  - query expressions and, 47
- TryGetObjectByKey method, 317–319
- T-SQL commands, 161
- two-tier solutions, LINQ to SQL in, 579–580
- type checking, 19
- typed DataSet, querying with LINQ, 352–353
- type declarations with simple relationships (listing), 9
- type declarations with two-way relationships (listing), 10
- typed nodes, validation of, 404–407
- Type property for data members of entity, 225
- type systems, transparency across, 20

## U

- Unchanged entity state, 178
- Unidirectional serialization setting (entities), 197

Union operator, 72–75  
 Unique association property, 228  
 unique object identity, 129–130  
 unit of work, identifying boundaries of, 606–607  
 Untracked entity state, 178  
 untyped DataSet data, accessing, 353  
 updatable views, 125  
 UpdateCheck argument (Column attribute), 188  
 Update Check data property, 225  
 UpdateCustomer method, 340  
 Update Model From Database feature (EDM Designer), 243  
 Update operations  
   intercepting, 184  
   mapping to, 237–238  
 updates, database, 179–180  
 update statements, customizing, 183–184  
 updating  
   entities, 302–303  
   event handlers, 620  
 user-defined functions (UDFs)  
   LINQ to Entities, 281–282  
   O/R Designer and, 235–238  
   using inside LINQ queries, 148–150  
 user interface (UI) layer, 577

## V

Validate method, 404  
 validation of typed nodes, 404–407  
 value type vs. reference type  
   semantic, 75–76  
 VARCHAR(MAX) type, 160  
 var keyword, 7  
 VerificationException type, 160  
 views, O/R Designer and, 238  
 VisitBinaryComparison  
   implementation, 507  
 visiting expression trees  
   DisplayVisitor specialization of ExpressionVisitor class (listing), 445  
   Expression tree visit based on a lambda expression approach (listing), 449–451  
   Expression visitor algorithm implemented through a lambda expression (listing), 447–449  
   ExpressionVisitor class, Visit method in (listing), 440–442  
   overview, 439–440  
   testing DisplayVisitor class (listing), 446–447

VisitBinary method in  
   ExpressionVisitor class (listing), 443–444  
 VisitConstant and VisitParameter methods in ExpressionVisitor class (listing), 443  
 VisitLambda method in  
   ExpressionVisitor class (listing), 444–445  
 VisitMethodCall and  
   VisitInvocation methods in ExpressionVisitor class (listing), 444–445  
 Visual Basic  
   aggregate operators in, 86–88  
   code defining customer types, 51–52  
   join statements, 40–41  
   keywords, 41  
   query expression in (listing), 24–27  
   query expression to group developers by programming language (listing), 34  
   query expression with join between data sources (listing), 31–32  
   query expression with orderby clause with multiple ordering conditions (listing), 36  
   reading XML file using LINQ to XML and Visual Basic syntax (listing), 16  
   source code (LINQ to SQL), 207–209  
   Visual Basic XML literals and global XML namespaces (listing), 376  
   Visual Basic XML literals used to declare XML content with default XML namespace (listing), 373  
   Visual Basic XML literals used to declare XML namespace with custom prefix (listing), 375  
   Visual Basic XML literal used to transform XML, 403  
   XML for orders, creating using Visual Basic XML literals, 16  
   XML literals, 362  
 Visual Studio, DBML files and, 218

## W

Warren, Matt, 166, 440  
 WCF Data Services, 647  
 WCF Data Services in Silverlight, 650

WCF RIA Services, 652  
 Web Application Toolkit for Bing Search, 571  
 websites, for downloading  
   AdxStudio xRM SDK, 567  
   BLToolkit, 563  
   DataObjects.NET, 563  
   DryadLINQ, 572  
   Entity Designer Database Generation Power Pack, 114, 245  
   Genom-e, 564  
   Indexed LINQ, 573  
   Json.NET 3.5 library, 571  
   LinqConnect, 564  
   LINQExtender, 573  
   LINQKit library, 463  
   LINQ over C#, 573  
   LINQPad, 573  
   LINQ to Expressions, 573  
   LINQ to Geo, 573  
   LLBLGen Pro runtime, 564  
   MetaLinq, 573  
   Microsoft DevLabs project, 560  
   NHibernate, 564  
   PLINQO, 564  
   SubSonic, 564  
   Telerik OpenAccess, 564  
   Web Application Toolkit for Bing Search, 571  
 websites, for further information  
   “ADO.NET Architecture”, 273  
   ADO.NET third-party providers, 596  
   canonical and database functions, 281  
   “Code Generation in LINQ to SQL”, 207  
   CopyToDataTable<T> custom method, 345  
   custom partitioning in PLINQ queries, 543  
   “Dataontext class”, 142  
   “Data Types and Functions (LINQ to SQL)”, 161  
   Data Mapper, 586  
   data selection with  
     EntityDataSource, 628  
   “Dealing with Linq’s Immutable Expression Trees”, 447  
   expression trees, 435  
   external XML mapping, 123  
   “How to Create a New .edmx File”, 242  
   “Introducing System.Transactions in the .NET Framework 2.0”, 322

- “LINQ: Building an IQueryable Provider-Part IX”, 166
  - “LINQ to SQL, Aggregates, Entities, and Quantum Mechanics”, 162
  - LINQ queries, applying to
    - EntityDataSource, 632
  - LINQ to Active Directory, 571
  - LINQ to Amazon, 570
  - LINQ to Flickr, 570
  - LINQ to Google
    - implementation, 570
  - LINQ to LDAP, 571
  - LINQ to Streams
    - implementation, 572
  - LINQ to WMI, 571
  - “MessageQueue Class”, 323
  - Microsoft Developer Network, 3
  - .NET Framework 4 TaskScheduler
    - implementations, 539
  - “Object States and Change Tracking”, 138
  - ObjectShredder<T> source
    - code, 345
  - Open Data (OData) Protocol, 565
  - O/R Designer for LINQ to SQL, 113
  - ORM tools supporting LINQ, 114
  - Parallel Computing Developer Center, 518
  - recursive lambda expressions, 417
  - SQLMetal command-line
    - options, 213
  - “Supported and Unsupported LINQ Methods (LINQ to Entities)”, 279
  - T4 templates, 271
  - “Unit of Work”, 180
  - “Volatile Resource Managers in .NET Bring Transactions to the Common Type”, 325
  - WCF Data Services in Silverlight, 650
  - WCF RIA Services, 652
  - “Working with Entity Keys”, 317
  - where clauses
    - basics, 32
    - defining filter with, 29
  - Where extension method, 26
  - WhereKey operator, 475
  - Where method, 26–27
  - Where operator, 53–54, 470–472, 482, 506
  - Windows Forms, using LINQ
    - with, 652–655
  - Windows Management Instrumentation (WMI)
    - repository, 5
  - Windows Presentation Foundation (WPF), 581
  - WithMergeOptions method, 545
  - WMI, LINQ to, 571
  - WPF (Windows Presentation Foundation), 637
  - writes, database, 192–193
- ## X
- XAML (Extensible Application Markup Language)
    - window with complex data binding, 646
    - excerpt to define Button element (listing), 641
    - window with simple data binding, 640
  - XAttributes, 385–386
  - XComment class, 377
  - XContainer class, 365
  - XDeclaration class, 377
  - XDocument class, 364–365
  - XDocumentType class, 377
  - XElement attributes, 385
  - XElement class, 365–369
  - XML (Extensible Markup Language)
    - development frameworks supporting, 359
    - document illustrating searching with LINQ to XML (listing), 388
    - external mapping, abstracting LINQ to SQL with, 581–584
    - external XML mapping files, 201–202, 210, 212
    - LINQ query expressions, using over XML nodes, 395–401
    - manipulation in LINQ, 14–16
    - querying efficiently to build entities, 397–401
    - reading/traversing/modifying, 382–384
    - transforming with LINQ to XML, 401–404
  - XML for orders, creating using Visual Basic XML literals (listing), 16
  - XML Infoset, 359, 363
  - XML literals, 362
  - XML names, manual escaping of, 369
  - XmlReader class, 409
  - XmlReader/XMLWriter, 14, 378
  - XmlSerializer class, 334
  - XML tree modification events
    - handling (listing), 380
  - XML, querying
    - InDocumentOrder extension method, 393
    - XAttributes, 385–386
    - XNode selection methods, 392–393
    - XPath Axes, extension methods similar to, 388–392
    - XName and XNamespace classes, 372–377
    - XNode class, 370–371, 382
    - XNode selection methods, 392–393
    - XObject annotations, 410
    - XObject class, 379–382
    - XPath
      - support for, 407–409
      - XPathEvaluate extension method (listing), 408
      - XPath/XQuery, 5
    - XProcessingInstruction class, 377
    - XQuery, selecting nodes with, 15–16
    - XSD (XML Schema Definition), 359, 404–407
    - XSLT (Extensible Stylesheet Language for Transformations), 359
    - XStreamingElement class, 377–379
    - XText class, 377
- ## Z
- “zero or one to many” multiplicity, 308
  - Zip operator, 76–77



## Paolo Pialorsi



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

## Marco Russo



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







# What do you think of this book?

We want to hear from you!

To participate in a brief online survey, please visit:

[microsoft.com/learning/booksurvey](https://microsoft.com/learning/booksurvey)

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

Thank you in advance for your input!

**Microsoft**  
Press

## Stay in touch!

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

[microsoft.com/learning/books/newsletter](https://microsoft.com/learning/books/newsletter)