

Paul Kimmel

Foreword by Darryl Hogan
Architect Evangelist, Microsoft Corporation

LINQ

UNLEASHED

for C#

SAMS

www.it-ebooks.info

Paul Kimmel

LINQ

UNLEASHED

for C#



800 East 96th Street, Indianapolis, Indiana 46240 USA

LINQ Unleashed for C#

Copyright © 2009 by Pearson Education, Inc.

All rights reserved. No part of this book shall be reproduced, stored in a retrieval system, or transmitted by any means, electronic, mechanical, photocopying, recording, or otherwise, without written permission from the publisher. No patent liability is assumed with respect to the use of the information contained herein. Although every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions. Nor is any liability assumed for damages resulting from the use of the information contained herein.

ISBN-13: 978-0-672-32983-8

ISBN-10: 0-672-32983-2

Library of Congress Cataloging-in-Publication Data

Kimmel, Paul.

LINQ unleashed for C# / Paul Kimmel. — 1st ed.

p. cm.

ISBN 978-0-672-32983-8

1. C# (Computer program language) 2. Microsoft LINQ. I. Title.

QA76.73.C154K5635 2009

005.13'3—dc22

2008030703

Printed in the United States of America

First Printing August 2008

Trademarks

All terms mentioned in this book that are known to be trademarks or service marks have been appropriately capitalized. Sams Publishing cannot attest to the accuracy of this information. Use of a term in this book should not be regarded as affecting the validity of any trademark or service mark.

Warning and Disclaimer

Every effort has been made to make this book as complete and as accurate as possible, but no warranty or fitness is implied. The information provided is on an "as is" basis. The authors and the publisher shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this book.

Bulk Sales

Sams Publishing offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales. For more information, please contact

U.S. Corporate and Government Sales

1-800-382-3419

corpsales@pearsontechgroup.com

For sales outside of the U.S., please contact

International Sales

international@pearson.com

Editor-in-Chief

Karen Gettman

Executive Editor

Neil Rowe

Development Editor

Mark Renfrow

Managing Editor

Kristy Hart

Project Editor

Betsy Harris

Copy Editor

Karen Annett

Indexers

Lisa Stumpf

Publishing Works

Proofreader

Linda Seifer

Technical Editor

Joe Kunk

Publishing

Coordinator

Cindy Teeters

Cover Designer

Gary Adair

Compositor

Jake McFarland

Contents at a Glance

Introduction	1
Part I Getting Ready for LINQ	
1 Programming with Anonymous Types	5
2 Using Compound Type Initialization	29
3 Defining Extension and Partial Methods	61
4 yield return: Using .NET's State Machine Generator	85
5 Understanding Lambda Expressions and Closures	97
6 Using Standard Query Operators	121
Part II LINQ for Objects	
7 Sorting and Grouping Queries	137
8 Using Aggregate Operations	151
9 Performing Set Operations	167
10 Mastering Select and SelectMany	185
11 Joining Query Results	211
12 Querying Outlook and Active Directory	239
Part III LINQ for Data	
13 Querying Relational Data with LINQ	265
14 Creating Better Entities and Mapping Inheritance and Aggregation	289
15 Joining Database Tables with LINQ Queries	309
16 Updating Anonymous Relational Data	349
17 Introducing ADO.NET 3.0 and the Entity Framework	383
Part IV LINQ for XML	
18 Extracting Data from XML	415
19 Comparing LINQ to XML with Other XML Technologies	437
20 Constructing XML from Non-XML Data	453
21 Emitting XML with the XmlWriter	463
22 Combining XML with Other Data Models	469
23 LINQ to XSD Supports Typed XML Programming	485
Index	499

Table of Contents

Introduction	1
Conventions Used in This Book	2
Part I Getting Ready for LINQ	
1 Programming with Anonymous Types	5
Understanding Anonymous Types	6
Programming with Anonymous Types.....	7
Defining Simple Anonymous Types.....	7
Using Array Initializer Syntax.....	7
Creating Composite Anonymous Types	9
Using Anonymous Type Indexes in For Statements	12
Anonymous Types and Using Statements	14
Returning Anonymous Types from Functions.....	17
Databinding Anonymous Types	18
Testing Anonymous Type Equality	23
Using Anonymous Types with LINQ Queries	24
Introducing Generic Anonymous Methods.....	25
Using Anonymous Generic Methods.....	26
Implementing Nested Recursion	27
Summary	28
2 Using Compound Type Initialization	29
Initializing Objects with Named Types	30
Implementing Classes for Compound Initialization Through Named Types	32
Understanding Auto-Implemented Properties	34
Initializing Anonymous Types	34
Initializing Collections	36
Finishing the Hypergraph	39
Implementing the Hypergraph Controls Using the Observer Pattern	47
Using Conversion Operators	51
ToArray	51
OfType	54
Cast	54
AsEnumerable	55

ToList	56
ToDictionary	57
ToLookup	58
Summary	60
3 Defining Extension and Partial Methods	61
Extension Methods and Rules of the Road	61
Defining Extension Methods	64
Implementing Extension Methods	64
Overloading Extension Methods	67
Defining Generic Extension Methods	69
How Extension Methods Support LINQ	73
Implementing a “Talking” String Extension Method	78
Defining Partial Methods	79
Summary	84
4 yield return: Using .NET’s State Machine Generator	85
Understanding How yield return Works	86
Using yield return and yield break	88
Profiling Code	93
Using yield break	95
Summary	95
5 Understanding Lambda Expressions and Closures	97
Understanding the Evolution from Function Pointers to Lambda Expressions	98
Writing Basic Lambda Expressions	101
Automatic Properties	102
Reading Lambda Expressions	103
Lambda Expressions Captured as Generic Actions	104
Searching Strings	106
Lambda Expressions Captured as Generic Predicates	108
Binding Control Events to Lambda Expressions	109
Dynamic Programming with Lambda Expressions	110
Using Select<T> with Lambda Expressions	110
Using Where<T> with Lambda Expressions	112
Using OrderBy<T> with Lambda Expressions	113
Compiling Lambda Expressions as Code or Data	114
Lambda Expressions and Closures	117
Currying	119
Summary	120

6 Using Standard Query Operators	121
Understanding How LINQ Is Implemented	121
Constructing a LINQ Query	122
Filtering Information	122
Using Quantifiers	124
Partitioning with Skip and Take	126
Using Generation Operations	127
DefaultIfEmpty	127
Empty	127
Range	127
Repeat	128
Equality Testing	129
Obtaining Specific Elements from a Sequence	131
Appending Sequences with Concat	132
Summary	133
Part II LINQ for Objects	
7 Sorting and Grouping Queries	137
Sorting Information	137
Sorting in Ascending and Descending Order	138
Sort in Descending Order Using the Extension Method Directly	140
Performing Secondary Sorts	141
Reversing the Order of Items	144
Grouping Information	145
Summary	150
8 Using Aggregate Operations	151
Aggregating	151
Averaging Collection Values	154
Counting Elements	157
Finding Minimum and Maximum Elements	157
Summing Query Results	162
Median: Defining a Custom Aggregation Operation	163
Summary	165
9 Performing Set Operations	167
Finding Distinct Elements	167
Finding Distinct Objects Using Object Fields	169
Defining Exclusive Sets with Intersect and Except	177
Creating Composite Resultsets with Union	182
Summary	184

10 Mastering Select and SelectMany	185
Exploring Select	185
Selecting with Function Call Effects	186
Manipulating Select Predicates	190
Returning Custom Business Objects from a Data Access Layer	190
Using Select Indexes to Shuffle (or Unsort) an Array	194
Forming the Basis of a Card Game Like Blackjack	195
Projecting New Types from Calculated Values	200
Importing DLLs	200
Using GDI+ with Windows API (or External DLL) Methods	201
Using Select to I-Cap Words	201
Projecting New Types from Multiple Sources	203
Creating a New Sequence from Multiple Sequences with SelectMany	205
Using SelectMany with Indexes	207
Summary	209
11 Joining Query Results	211
Using Multiple From Clauses	211
Defining Inner Joins	213
Using Custom, or Nonequijoins	214
Defining a Nonequal Custom Join	215
Defining a Custom Join with Multiple Predicates	219
Defining Custom Joins with a Temporary Range Variable	220
Implementing Group Join and Left Outer Join	224
Defining a Group Join	224
Implementing a Left Outer Join	226
Implementing a Cross Join	228
Defining Joins Based on Composite Keys	237
Summary	237
12 Querying Outlook and Active Directory	239
LINQ to Outlook	239
Querying Active Directory with Straight C# Code	243
LINQ to Active Directory	245
Creating an IQueryable LINQ Provider	245
Implementing the IQueryableProvider	246
Defining Active Directory as the Data Source	248
Converting a LINQ Query to an Active Directory Query	252
Implementing Helper Attributes	257
Defining Active Directory Schema Entities	259
Querying Active Directory with LINQ	260
Summary	262

Part III LINQ for Data

13 Querying Relational Data with LINQ	265
Defining Table Objects	266
Mapping Classes to Tables	269
Viewing the Query Text Generated by LINQ	273
Connecting to Relational Data with DataContext Objects	275
Querying DataSets	277
Selecting Data from a DataTable	278
Querying the DataTable with a Where Clause	280
Using Partitioning Methods	282
Sorting Against DataTables	282
Defining a Join with DataSets	282
SqlMetal: Using the Entity Class Generator Tool	285
Using the LINQ to SQL Class Designer	285
Summary	287
14 Creating Better Entities and Mapping Inheritance and Aggregation	289
Defining Better Entities with Nullable Types	289
Mapping Inheritance Hierarchies for LINQ to SQL	294
Creating Inheritance Mappings with the LINQ to SQL Designer	298
Customizing Classes Created with the LINQ to SQL Designer	299
Adding EntitySet Classes as Properties	300
Creating Databases with LINQ to SQL	305
Summary	308
15 Joining Database Tables with LINQ Queries	309
Defining Joins with LINQ to DataSet	310
Coding Equijoins	310
Coding Nonequijoins	312
Defining a Left Join and a Word about Right Joins	313
Considering Right Joins	315
Defining Joins with LINQ to SQL	317
Coding Equijoins	317
Implementing the Group Join	321
Implementing a Left Join	331
Querying Views with LINQ	340
Building a View in SQL Server	340
Querying a View with LINQ to SQL	342
Databinding with LINQ to SQL	345
Summary	347

16 Updating Anonymous Relational Data	349
Adding and Removing Data	349
Inserting Data with LINQ to SQL	349
Deleting Data with LINQ to SQL	352
Updating Data with LINQ to SQL	354
Using Stored Procedures	355
Calling User-Defined Functions	363
Using Transactions	366
Understanding Conflict Resolution	368
Indicating the Conflict Handling Mode for SubmitChanges	369
Catching and Resolving Concurrency Conflicts	371
N-Tier Applications and LINQ to SQL	376
Summary	382
17 Introducing ADO.NET 3.0 and the Entity Framework	383
Understanding the General Nature of the Problem and the Solution	384
Understanding Problems with the Relational Database	
Model as It Pertains to C# Programmers	384
Understanding How the Entity Framework Is Designed to Help	385
Grokking the Nature of the Solution	385
Finding Additional Resources	386
Wikipedia	387
Entity SQL Blog	387
Downloading and Installing the Entity Framework	387
Downloading Samples	388
Go Live Estimate	388
Building a Sample Application Using Vanilla ADO.NET Programming	389
Defining a Database for Storing Stock Quotes	389
Adding a Stored Procedure for Inserting Quotes	390
Adding a Foreign Key	393
Reference: The Complete Sample Database Script	394
Writing Code to Obtain the Stock Quotes and Update the Database	397
Programming with the Entity Framework	401
Creating the Entity Data Model	401
Adding an Association	402
Querying the Entity Data Model with Entity SQL	402
Querying the Entity Data Model with LINQ to Entities	405
Doing It All with LINQ	406
Summary	411

Part IV LINQ for XML

18 Extracting Data from XML	415
Loading XML Documents	415
Querying XML Documents	416
Using XDocument	416
Using XElement	420
Managing Attributes	420
Loading XML from a String	424
Handling Missing Data	425
Using Query Expressions with XML Data	426
Using Namespaces	427
Nesting Queries	428
Filtering with Where Clauses	429
Finding Elements Based on Context	430
Sorting XML Queries	431
Calculating Intermediate Values with Let	432
Annotating Nodes	433
Summary	435
19 Comparing LINQ to XML with Other XML Technologies	437
Comparing LINQ to XML with XPath	438
Using Namespaces	439
Finding Children	441
Finding Siblings	442
Filtering Elements	442
Comparing LINQ to XML Transformations with XSLT	443
Transforming XML Data Using Functional Construction	450
Summary	452
20 Constructing XML from Non-XML Data	453
Constructing XML from CSV Files	454
Generating Text Files from XML	456
Using XML and Embedded LINQ Expressions (in VB)	458
Summary	462
21 Emitting XML with the XmlWriter	463
Exploring the XmlWriter, Quickly	464
Using XmlTextWriter to Write an XML File	465
Summary	467

22	Combining XML with Other Data Models	469
	Creating XML from SQL Data	469
	Defining the Object-Relational Map	470
	Constructing the XML Document from the SQL Data	473
	Using the XComment Node Type	475
	Displaying the XML Document in a TreeView	475
	Updating SQL Data from XML	478
	Summary	483
23	LINQ to XSD Supports Typed XML Programming	485
	Understanding the Basic Design Goals of LINQ to XSD	486
	Programming with LINQ to XSD	487
	Downloading and Installing the LINQ to XSD Preview	487
	Creating a LINQ to XSD Preview Console Application	487
	Defining the XML Context	488
	Defining the XML Schema File	490
	Adding a Regular Expression to a Schema File	491
	Querying with LINQ to XML for Objects	496
	Summary	498
	Index	499

Foreword

Data affects just about every aspect of our lives. Everything we do is analyzed, scrutinized, and delivered back to us in the form of coupons and other marketing materials. When you write an application, you can be sure that data in one form or another will be part of the solution. As software developers, the ease with which we can store, retrieve, and analyze data is crucial to our ability to develop compelling applications. Add to that the fact that data can come in a number of different shapes and formats, and it quickly comes to light that there is tremendous value in a consistent framework for accessing many types of data.

Several different data access approaches have been developed for Windows developers over the years. ADO and OLEDB and subsequently ADO.NET gave us universal access to relational databases. MSXML and ADO.NET made it possible to inspect and manipulate XML documents. Each of these technologies had their benefits and drawbacks, but one common thread ran through each of them: They failed to deliver data access capabilities in a way that felt natural to developers.

LINQ now makes data access a first-class programming concept in .NET, making it possible for developers to express queries in a way that makes sense to developers. What makes LINQ unique is that it enables programmers to create type-safe data access code complete with Intellisense support and compile time syntax checking.

Paul Kimmel has done an excellent job of presenting LINQ in a concise and complete manner. Not only has he made LINQ approachable, but he has also masterfully explained concepts such as Anonymous Types and Lambda Expressions that help make LINQ a reality. The sample code throughout the book demonstrates the application of the technology in a clear and meaningful way. This is a great “Saturday morning with a pot of coffee” kind of book. I hope you’ll dive in and get as much out of this book as I did.

Darryl Hogan
Architect Evangelist, Microsoft

About the Author

Paul Kimmel is a four-time Microsoft MVP, the author of over a dozen books on object-oriented programming and UML, including three books on Microsoft .NET, a columnist for codeguru.com, developer.com, informit.com, devsource.com, and devx.com, a cofounder of the Greater Lansing Area .NET Users Group (glugnet.org, East Lansing and Flint), a full-time software developer, and sometimes pilot. Paul still lives and works in the greater Lansing, Michigan, area (and hasn't given up on the economy). After 15 years of independent consulting, Paul now works for EDS as an application architect.

Dedication

This book is dedicated to the men and women of the United States armed forces, especially those brave souls serving in conflict zones, away from hearth and kin. To all military police men and women—Lookout Firefly it's Night-train! Mohawk, 38 out.

Acknowledgments

Sometimes I read other people's acknowledgements but a long list of names is boring unless it's your name. If the casual reader is reading this, then you are helping me thank the myriad of people that make a book like this possible. If you are one of the people listed, then know that your able assistance is greatly appreciated.

Thanks to Neil Rowe, my acquisitions editor; Curt Johnson, marketing manager at Pearson; and Joan Murray, editor, also at Pearson. I wrote slower but finishing a book is thrilling.

Thanks to Joe Kunk for not complaining about how much work technical editing is and actually testing the examples. Joe also does a lot of heavy lifting at Glugnet East Lansing and Flint (our .NET User Groups). Thanks to the rest of the glugnet board for covering for me in my absence: Vivek Joshi, Alireza Namvar, Eric Vogel, Jeff McWherter, Vijay Jagdale, Jason Harris, Aaron Lilywhite, and Rich Hamilton.

Thanks to Bart De Smet from Microsoft for assistance with the Active Directory IQueryable provider—LINQ to AD is cool. Thanks to Daryl Hogan for the foreword.

I'd also like to mention Brian Dawson, Brad Jones, Tyler Durden, Jackson Wayfare, Blizzard for War Craft and Rockstar games for GTA IV, my kids Alex and Noah, Dena and Joe Swanson (for wine and comic relief), Ed Swanson for cigars, and all of the great folks at Pearson behind the scenes that turn my six-fingered manuscript into a shiny new book.

We Want to Hear from You!

As the reader of this book, *you* are our most important critic and commentator. We value your opinion and want to know what we're doing right, what we could do better, what areas you'd like to see us publish in, and any other words of wisdom you're willing to pass our way.

You can email or write me directly to let me know what you did or didn't like about this book—as well as what we can do to make our books stronger.

Please note that I cannot help you with technical problems related to the topic of this book, and that due to the high volume of mail I receive, I might not be able to reply to every message.

When you write, please be sure to include this book's title and author, as well as your name and phone or email address. I will carefully review your comments and share them with the author and editors who worked on the book.

E-mail: feedback@samspublishing.com

Mail:
Neil Rowe
Executive Editor
Sams Publishing
800 East 96th Street
Indianapolis, IN 46240 USA

Reader Services

Visit our website and register this book at www.informit.com/register for convenient access to any updates, downloads, or errata that might be available for this book.

This page intentionally left blank

Introduction

By the time you are holding this book in your hands, I will have 30 years in since the first time I wrote some code. That code was ROM-BASIC on a TRS-80 in Washington grammar school in Owosso, Michigan, and I was in the fifth grade. Making the “tank” slide back and forth shooting blips across the screen was neat. Changing the code to change blip speeds and numbers of targets was exhilarating. Three decades later and I get more excited each passing year. There are great technologies on the horizon like Microsoft Surface, Popfly, and LINQ. This book is about LINQ, or Language INtegrated Query.

LINQ is a SQL-like language for C#. When I first saw it, I didn't like it. My first impression was that someone had glommed on a bastardization of C# and it was ugly like SQL can get. I didn't like it because I didn't understand it. However, I gave LINQ a second chance (as I want you to do) and discovered that LINQ is thoroughly integrated, tremendously powerful, and almost as much fun as a Tesla Roadster or doing hammerheads in an Extra 300L.

The query capabilities of LINQ are extended to objects, SQL, DataSets, XML, XSD, entities, and can be extended to other providers like Active Directory or SharePoint. This means that you can write queries—that are similar in syntax—against objects, data, XML, XSD, entities, or Active Directory (with a little work) much like you would a SQL query in a database. And, LINQ is actually engineered artfully and brilliantly on top of generics as well as some new features in .NET 3.5, such as extension methods, anonymous types, and Lambda Expressions. Another very important characteristic of LINQ is that it clearly demonstrates Microsoft's willingness to innovate and take the best of existing technologies

like Lambda Calculus— invented in the 1930s—and if it's good or great, incorporate these elements into the tools and languages we love.

LINQ and its underpinnings are powerful *and* challenging, and in this book you will get what you need to know to completely understand all that makes LINQ work and begin using it immediately. You will learn about anonymous methods, extension methods, Lambda Expressions, state machines, how generics and the CodeDOM play a big role in powerful tools like LINQ, and writing LINQ queries and why you will want to do it in the bigger, grander scheme of things. You will also learn how to save a ton of time and effort by not hard-coding those elements that you will no longer need or want to hard-code, and you will have a better grasp of how LINQ fits into n-tier architectures without breaking guidelines that have helped you succeed to date.

Brought to you by a four-time Microsoft MVP and columnist for over a decade, *LINQ Unleashed for C#* will teach you everything you need to know about LINQ and .NET 3.5 features and how to be more productive and have more fun than ever before.

Conventions Used in This Book

The following typographic conventions are used in this book:

Code lines, commands, statements, variables, and text you see onscreen appear in a monospace typeface.

Occasionally in listings bold is used to draw attention to the snippet of code being discussed.

Placeholders in syntax descriptions appear in an *italic monospace* typeface. You replace the placeholder with the actual filename, parameter, or whatever element it represents.

Italics highlight technical terms when they're being defined.

A code-continuation icon is used before a line of code that is really a continuation of the preceding line. Sometimes a line of code is too long to fit as a single line on the page. If you see ➔ before a line of code, remember that it's part of the line immediately above it.

The book also contains Notes, Tips, and Cautions to help you spot important or useful information more quickly.

PART I

Getting Ready for LINQ

IN THIS PART

CHAPTER 1	Programming with Anonymous Types	5
CHAPTER 2	Using Compound Type Initialization	29
CHAPTER 3	Defining Extension and Partial Methods	61
CHAPTER 4	<code>yield return</code> : Using .NET's State Machine Generator	85
CHAPTER 5	Understanding Lambda Expressions and Closures	97
CHAPTER 6	Using Standard Query Operators	121

This page intentionally left blank

CHAPTER 1

Programming with Anonymous Types

"Begin at the beginning and go on till you come to the end; then stop."

—Lewis Carroll, from *Alice's Adventures in Wonderland*

Finding a beginning is always a little subjective in computer books. This is because so many things depend on so many other things. Often, the best we can do is put a stake in the ground and start from that point. Anonymous types are our stake.

Anonymous types use the keyword `var`. `Var` is an interesting choice because it is still used in Pascal and Delphi today, but `var` in Delphi is like `ByRef` in Visual Basic (VB) or `ref` in C#. The `var` introduced with .NET 3.5 indicates an anonymous type. Now, our VB friends are going to think, *"Well, we have had variants for years; big deal."* But `var` is not a dumbing down and clogging up of C#. Anonymous types are something new and necessary.

Before looking at anonymous types, let's put a target on our end goal. Our end goal is to master LINQ (integrated queries) in C# for objects, Extensible Markup Language (XML), and data. We want to do this because it's cool, it's fun, and, more important, it is very powerful and expressive. To get there, we have to start somewhere and anonymous types are our starting point.

Anonymous types quite simply mean that you don't specify the type. You write `var` and C# figures out what type is defined by the right side, and C# emits (writes the code), indicating the type. From that point on, the type is strongly defined, checked by the compiler (not at runtime), and exists as a complete type in your code. Remember, you

IN THIS CHAPTER

- ▶ Understanding Anonymous Types
- ▶ Programming with Anonymous Types
- ▶ Databinding Anonymous Types
- ▶ Testing Anonymous Type Equality
- ▶ Using Anonymous Types with LINQ Queries
- ▶ Introducing Generic Anonymous Methods

didn't write the type definition; C# did. This is important because in a query language, you are asking for and getting *ad hoc* types that are defined by the context, the query result. In short, your query's result might return a previously undefined type.

An important concept here is that you don't write code to define the ad hoc types—C# does—so, you save time by not writing code. You save design time, coding time, and debug time. Microsoft pays that cost. Anonymous types are the vessel that permit you to use these ad hoc types. By the time you are done with this chapter, you will have mastered the left side of the operator and a critical part of LINQ.

In addition, to balance the book, the chapters are laced with useful or related concepts that are generally helpful. This chapter includes a discussion on generic anonymous methods.

Understanding Anonymous Types

Anonymous types defined with `var` are not VB variants. The `var` keyword signals the compiler to emit a strong type based on the value of the operator on the right side. Anonymous types can be used to initialize simple types like integers and strings but detract modestly from clarity and add little value. Where `var` adds punch is by initializing composite types on the fly, such as those returned from LINQ queries. When such an anonymous type is defined, the compiler emits an immutable—read-only properties—class referred to as a projection.

Anonymous types support IntelliSense, but the class should not be referred to in code, just the members.

The following list includes some basic rules for using anonymous types:

- ▶ Anonymous types must always have an initial assignment and it can't be null because the type is inferred and fixed to the initializer.
- ▶ Anonymous types can be used with simple or complex types but add little value to simple type definitions.
- ▶ Composite anonymous types require member declarators; for example, `var joe = new {Name="Joe" [, declaratory=value, ...]}`. (In the example, `Name` is the member declaratory.)
- ▶ Anonymous types support IntelliSense.
- ▶ Anonymous types cannot be used for a class field.
- ▶ Anonymous types can be used as initializers in `for` loops.
- ▶ The `new` keyword can be used and has to be used for array initializers.
- ▶ Anonymous types can be used with arrays.
- ▶ Anonymous types are all derived from the `Object` type.
- ▶ Anonymous types can be returned from methods but must be cast to `object`, which defeats the purpose of strong typing.

- ▶ Anonymous types can be initialized to include methods, but these might only be of interest to linguists.

The single greatest value and the necessity of anonymous types is they support creating single-use elements and composite types returned by LINQ queries without the need for the programmer to fully define these types in static code. That is, the designers can focus significantly on primary domain types, and the programmers can still create single-use anonymous types ad hoc, letting the compiler write the class definition.

Finally, because anonymous types are immutable—think no property setters—two separately defined anonymous types with the same field values are considered equal.

Programming with Anonymous Types

This chapter continues by exploring all of the ways you can use anonymous types, paving the way up to anonymous types returned by LINQ queries, stopping at the full explanation of the LINQ query here. You can simply think of the query as a first look at queries with the focus being on the anonymous type itself and what you can do with those types.

Defining Simple Anonymous Types

A simple anonymous type begins with the var keyword, the assignment operator (=), and a non-null initial value. The anonymous type is assigned to the name on the left side of the assignment operator, and the type emitted by the compiler to Microsoft Intermediate Language (MSIL) is determined by the right side of the operator. For instance:

```
var title = "LINQ Unleashed for C#";
```

uses the anonymous type syntax and assigns the string value to “LINQ Unleashed for C#”. This code is identical in the MSIL to the following:

```
string title = "LINQ Unleashed for C#";
```

This emitted code equality can be seen by looking at the Intermediate Language (IL) with the Intermediate Language Disassembler (ILDASM) utility (see Figure 1.1).

The support for declaring simple anonymous types exists more for completeness and symmetry than utility. In departmental language wars, purists are likely to rail against such use as it adds ambiguity to code. The truth is the type of the data is obvious in such simple use examples and it hardly matters.

Using Array Initializer Syntax

You can use anonymous type syntax for initializing arrays, too. The requirements are that the new keyword must be used. For example, the code in Listing 1.1 shows a simple console application that initializes an anonymous array of Fibonacci numbers. (The anonymous type and array initialization statement are highlighted in bold font.)

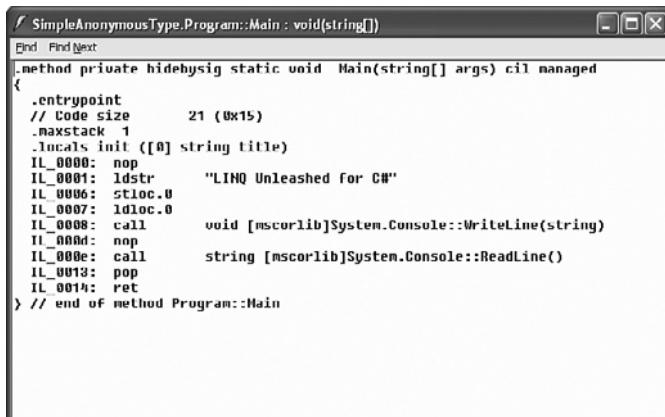


FIGURE 1.1 Looking at the .locals init statement and the `Console::Write(string)` statement in the MSIL, it is clear that `title` is emitted as a string.

LISTING 1.1 An Anonymous Type Initialized with an Array of Integers

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ArrayInitializer
{
    class Program
    {
        static void Main(string[] args)
        {
            // array initializer
            var fibonacci = new int[]{ 1, 1, 2, 3, 5, 8, 13, 21 };
            Console.WriteLine(fibonacci[0]);
            Console.ReadLine();
        }
    }
}

```

The first eight numbers in the Fibonacci system are defined on the line that begins `var fibonacci`. Fibonacci numbers start with the number 1 and the sequence is resolved by adding the prior two numbers. (For more information on Fibonacci numbers, check out Wikipedia; Wikipedia is wicked cool at providing detailed facts about such esoterica.)

Even in the example shown in Listing 1.1, you are less likely to get involved in language ambiguity wars if you use the actual type `int[]` instead of the anonymous type syntax for arrays.

Creating Composite Anonymous Types

Anonymous types really start to shine when they are used to define composite types, that is, classes without the “typed” class definition. Think of this use of anonymous types as defining an inline class without all of the typing. Listing 1.2 shows an anonymous type representing a lightweight person class.

LISTING 1.2 An Anonymous Type Containing Two Fields and Two Properties Without All of the Class Plumbing Typed By the Programmer

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ImmutableAnonymousTypes
{
    class Program
    {
        static void Main(string[] args)
        {
            var dena = new {First="Dena", Last="Swanson"};
            //dena.First = "Christine"; // error - immutable
            Console.WriteLine(dena);
            Console.ReadLine();
        }
    }
}
```

The anonymous type defined on the line starting with `var dena` emits a class, referred to as a projection, in the MSIL (see Figure 1.2). Although the projection’s name—the class name—cannot be referred to in code, the member elements—defined by the member declarators `First` and `Last`—can be used in code and IntelliSense works for all the elements of the projection (see Figure 1.3).

Another nice feature added to anonymous types is the overloaded `ToString` method. If you look at the MSIL or the output from Listing 1.2, you will see that the field names and field values, neatly formatted, are returned from the emitted `ToString` method. This is useful for debugging.

Adding Behaviors to Anonymous Composite Types

If you try to add a behavior to an anonymous type at initialization—for instance, by using an anonymous delegate—the compiler reports an error. However, it is possible with a little bending and twisting to add behaviors to anonymous types. The next section shows you how.

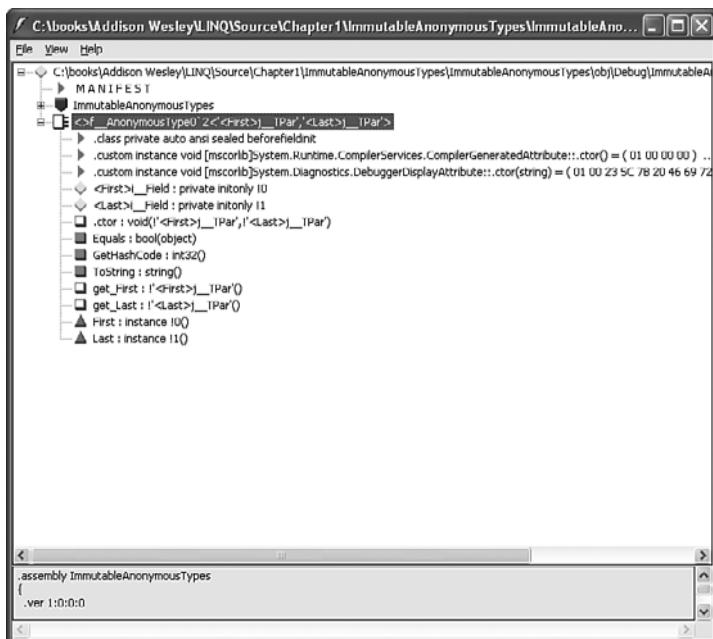


FIGURE 1.2 Anonymous types save a lot of programming time when it comes to composite types, as shown by the elements emitted to MSIL.

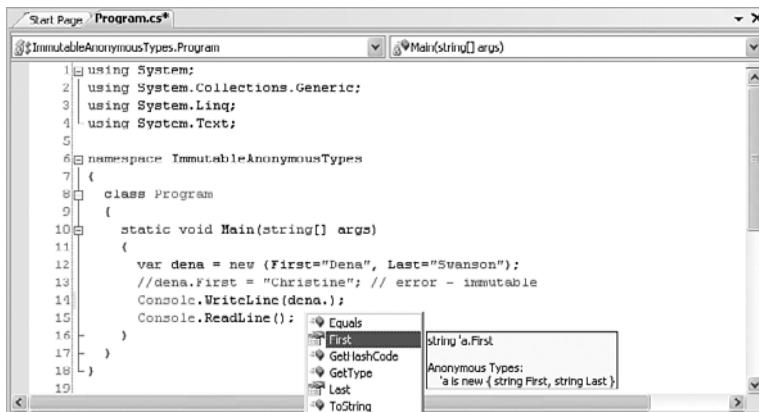


FIGURE 1.3 IntelliSense works quite well with anonymous types.

Adding Methods to Anonymous Types

To really understand language possibilities, it's helpful to bend and twist a language to make it do things it might not have been intended to do directly. One of these things is adding behaviors (aka methods). Although it might be harder to find a practical use for anonymous type-behaviors, Listing 1.4 shows you how to add a behavior to and use that behavior with an anonymous type. (The generic delegate **Func** in bold in the listing is used to initial the anonymous type's method.)

LISTING 1.4 Adding a Behavior to an Anonymous Type

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Reflection;

namespace AnonymousTypeWithMethod
{
    class Program
    {
        static void Main(string[] args)
        {
            // adding method possibility
            Func<string, string, string> Concat1 =
                delegate(string first, string last)
            {
                return last + ", " + first;
            };

            // whacky method but works
            Func<Type, Object, string> Concat2 =
                delegate(Type t, Object o)
            {
                PropertyInfo[] info = t.GetProperties();
                return (string)info[1].GetValue(o, null) +
                    ", " + (string)info[0].GetValue(o, null);
            };

            var dena = new {First="Dena", Last="Swanson", Concat=Concat1};
            //var dena = new {First="Dena", Last="Swanson", Concat=Concat2};
            Console.WriteLine(dena.Concat(dena.First, dena.Last));
            //Console.WriteLine(dena.Concat(dena.GetType(), dena));
            Console.ReadLine();
        }
    }
}
```

The technique consists of defining an anonymous delegate and assigning that anonymous delegate to the generic Func class. In the example, Concat was defined as an anonymous delegate that accepts two strings, concatenates them, and returns a string. You can assign that delegate to a variable defined as an instance of Func that has the three string parameter types. Finally, you assign the variable Concat to a member declarator in the anonymous type definition (referring to var dena = new {First="Dena", Last="Swanson", Concat=Concat}; now).



After the plumbing is in place, you can use IntelliSense to see that the behavior—Concat—is, in fact, part of the anonymous type *dena*, and you can invoke it in the usual manner.

Using Anonymous Type Indexes in For Statements

The var keyword can be used to initialize the index of a `for` loop or the recipient object of a `foreach` loop. The former is a simple anonymous type and the latter becomes a useful construct when the container to iterate over is something more than a sample collection. Listing 1.5 shows a `for` statement, and Listing 1.6 shows the `foreach` statement, both using the `var` construct.

LISTING 1.5 Demonstrating How to Iterate Over an Array of Integers—Using the Fibonacci Numbers from Listing 1.1—and the `var` Keyword to Initialize the Index

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace AnonymousForLoop
{
    class Program
    {
        static void Main(string[] args)
        {
            var fibonacci = new int[]{ 1, 1, 2, 3, 5, 8, 13, 21 };
            for( var i=0; i<fibonacci.Length; i++)
                Console.WriteLine(fibonacci[i]);
            Console.ReadLine();
        }
    }
}
```

LISTING 1.6 Demonstrating Basically the Same Code but Using the More Convenient `foreach` Construct

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace AnonymousForEachLoop
{
    class Program
    {
        static void Main(string[] args)
```

LISTING 1.6 Continued

```
{  
    var fibonacci = new int[]{ 1, 1, 2, 3, 5, 8, 13, 21 };  
    foreach( var fibo in fibonacci)  
        Console.WriteLine(fibo);  
    Console.ReadLine();  
}  
}  
}
```

The only requirement that must be met for an object to be the iterand in a `foreach` statement is that it must functionally represent an object that implements `IEnumerable` or `IEnumerable<T>`—the generic equivalent. Incidentally, this is also the same requirement for bindability, as in binding to a `GridView`.

TIP

At any time, you can branch in `for` or `foreach` statements with the `break` or `continue` keywords or the `goto`, `return`, or `throw` statements.

An all-too-common use of the `for` construct is to copy a subset of elements from one collection of objects to a new collection, for example, copying all the customers in the 48843 ZIP code to a `customersToCallOn` collection. In C# 2.0, the `yield return` and `yield break` key phrases actually played this role. For example, `yield return` signaled the compiler to emit a *state machine* in MSIL—in essence, it emitted the `copy` collection for you.

In .NET 3.5, the ability to query collections, datasets, and XML to essentially ask questions about data or copy some elements is one of those things that LINQ does very well. Listing 1.7 shows code that uses a LINQ statement to return just the numbers in the Fibonacci short sequence that are divisible by 3. (For now, don't worry about understanding all of the elements of the query.)

LISTING 1.7 A `foreach` Statement Whose Iterand Is Derived from a LINQ Query

```
using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Text;  
  
namespace AnonymousForEachLoopFromExpression  
{  
    class Program  
    {
```

LISTING 1.7 Continued

```
static void Main(string[] args)
{
    var fibonacci = new int[]{ 1, 1, 2, 3, 5, 8, 13, 21, 33, 54, 87 };
    // uses LINQ query
    foreach( var fibo in from f in fibonacci where f%3==0 select f )
        Console.WriteLine(fibo);
    Console.ReadLine();
}
```

The LINQ query—used as the iterand in the `foreach` statement—makes up this part of the Listing 1.7:

```
from f in fibonacci where f % 3 == 0 select f
```

For now, it is enough to know that this query meets the requirement that it returns an *enumerable* result, in fact, `IEnumerable<T>` where *T* is an *int* type.

If this is your first experience with LINQ, the query might look strange. The capability and power and this book will quickly make them familiar and desirable friends. For now, it is enough to know that queries meet the requirement of an enumerable resultset and can be used in a `foreach` statement.

Anonymous Types and Using Statements

The `using` statement is shorthand notation for `try...finally`. With `try...finally` and `using`, the purpose is to ensure resources are cleaned up before the `using` block exits or the `finally` block is run. This is accomplished by calling `Dispose`, which implies that items created in `using` statements implement `IDisposable`. Employ `using` when the created types implement `IDisposable`—like `SqlConnections`—and use `try...finally` when you need to do some kind of cleanup work, but do not necessarily need to invoke `Dispose` (see Listing 1.8).

LISTING 1.8 Using Statement and var Work Because SqlConnection Implements IDisposable

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data;
using System.Data.SqlClient;

namespace AnonymousUsingStatement
```

LISTING 1.8 Continued

```
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            string connectionString =  
                "Data Source=BUTLER;Initial Catalog=AdventureWorks2000;" +  
                "Integrated Security=True";  
            using( var connection = new SqlConnection(connectionString))  
            {  
                connection.Open();  
                Console.WriteLine(connection.State);  
                Console.ReadLine();  
  
            }  
        }  
    }  
}
```

The help documentation will verify that `SqlConnection` is derived from `DBConnection`, which, in turn, implements `IDisposable`. You can use a tool like Anakrino or Reflector—free decompilers and disassemblers—to see that `Dispose` in `DBConnection` invokes the `Close` method on a connection.

To really understand how things are implemented, you can use ILDASM—or one of the previously mentioned decompilers—and look at the MSIL that is emitted. If you look at the code in Listing 1.8's IL, you can clearly see the substitution of `using` for a properly configured `try...finally` block. (The `try` element—after `SqlConnection` creation—and the `finally` block invoking `Dispose` are shown in bold font in Listing 1.9.)

LISTING 1.9 The MSIL for the `var` and `using` Statement in Listing 1.8

```
.method private hidebysig static void Main(string[] args) cil managed  
{  
    .entrypoint  
    // Code size       66 (0x42)  
    .maxstack  2  
    .locals init ([0] string connectionString,  
                 [1] class [System.Data]System.Data.SqlClient.SqlConnection connection,  
                 [2] bool CS$4$0000)  
    IL_0000:  nop  
    IL_0001:  ldstr      "Data Source=BUTLER;Initial Catalog=AdventureWorks2"  
              + "000;Integrated Security=True"  
    IL_0006:  stloc.0
```



LISTING 1.9 Continued

```

IL_0007: ldloc.0
IL_0008: newobj     instance void
           &[System.Data]System.Data.SqlClient.SqlConnection::ctor(string)
IL_000d: stloc.1
.try
{
    IL_000e: nop
    IL_000f: ldloc.1
    IL_0010: callvirt instance void
[System.Data]System.Data.Common.DbConnection::Open()
    IL_0015: nop
    IL_0016: ldloc.1
    IL_0017: callvirt     instance valuetype[System.Data]System.Data.ConnectionState
[System.Data]System.Data.Common.DbConnection::get_State()
    IL_001c: box         [System.Data]System.Data.ConnectionState
    IL_0021: call         void [mscorlib]System.Console::WriteLine(object)
    IL_0026: nop
    IL_0027: call         string [mscorlib]System.Console::ReadLine()
    IL_002c: pop
    IL_002d: nop
    IL_002e: leave.s    IL_0040
} // end .try
finally
{
    IL_0030: ldloc.1
    IL_0031: ldnnull
    IL_0032: ceq
    IL_0034: stloc.2
    IL_0035: ldloc.2
    IL_0036: brtrue.s   IL_003f
    IL_0038: ldloc.1
    IL_0039: callvirt     instance void [mscorlib]System.IDisposable::Dispose()
    IL_003e: nop
    IL_003f: endfinally
} // end handler
IL_0040: nop
IL_0041: ret
} // end of method Program::Main

```

You don't have to master IL to use .NET effectively, but you can learn from it and writing .NET emitters—code that emits IL directly—is supported in the .NET Framework. As shown in the MSIL, you can infer, for example, that the proper way to use `try...finally` is to create the protected object, try to use it, and, finally, clean it up. If you read a little further—in the `finally` block starting with IL 0030—you can see that the compiler also

put a check in to ensure that the protected object, the `SqlConnection`, is compared with null before `Dispose` is called. This code is demonstrated in IL 0030, IL 0031, IL 0032, and the branch statement on IL 0036.

Returning Anonymous Types from Functions

Anonymous types can be returned from functions because the garbage collector (GC) cleans up any objects, but outside of the defining scope, the anonymous type is an instance of an object. Unfortunately, returning an object defeats the value of the IntelliSense system and the strongly typed nature of anonymous types. Although you could use reflection to rediscover the capabilities of the anonymous type, again you are taking a feature intended to make life more convenient and making it somewhat inconvenient again. Listing 1.10 puts these elements together, but as a practical matter, it is best to design solutions to use anonymous types within the defining scope. (Ironically, using objects within the defining scope was a style issue used in C++ to reduce the probability of memory leaks. Those familiar with C++ won't find this slight quirk of anonymous types any more inconvenient.)

LISTING 1.10 Returning an Anonymous Type from a Method Defeats the Strongly Typed Utility of Anonymous Types

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Reflection;

namespace ReturnAnonymousTypeFromMethod
{
    class Program
    {
        static void Main(string[] args)
        {
            var anon = GetAnonymous();
            Type t = anon.GetType();
            Console.WriteLine(t.GetProperty("Stock").GetValue(anon, null));
            Console.ReadLine();
        }

        public static object GetAnonymous()
        {
            var stock = new { Stock="MSFT", Price="32.45" };
            return stock;
        }
    }
}
```

Although it is intellectually satisfying to play with the reflection subsystem, writing code like that in Listing 1.10 is a slow and painful means to an end. (In addition, the code in Listing 1.10, as written, is fraught with the potentiality for bugs due to null values being returned from `GetType`, `GetProperty`, and `GetValue`.)

Databinding Anonymous Types

Some interesting startups got blown up when the stock market bubble burst, such as PointCast. PointCast searched the web—based on criteria the user provided—and displayed stock prices on a ticker and news in a browsable environment. One of the possible kinds of data was streaming stock prices. (Thankfully, the 1990s day-trading craze is over, but the ability to get such data is still interesting.)

This section looks at how you can combine cool technologies, such as anonymous types, AJAX, `HttpWebRequests`, `HttpWebResponse`s, and queries to Yahoo!'s stock-quoting capability, and assemble a web stock ticker. Aside from the code, a demonstration of data-binding anonymous types, and a brief description of what role the various technology elements are playing, this book doesn't elaborate in detail on features like AJAX (because of space and topic constraints). (For more information on web programming, see Stephen Walther's *ASP.NET 3.5 Unleashed*.)

The sample (in Listing 1.11) is actually very easy to complete, but uses some very cool technology and plumbing underneath. In the solution, a website project was created. The application contains a single .aspx web page. On that page, a `ScriptManager`, `UpdatePanel1` (both AJAX controls), a `DataList`, `Label`, and AJAX Timer are added. The design-time view of the page is shown in Figure 1.4 and the runtime view is shown in Figure 1.5. (Listing 1.12 shows the settings for the Web controls.)

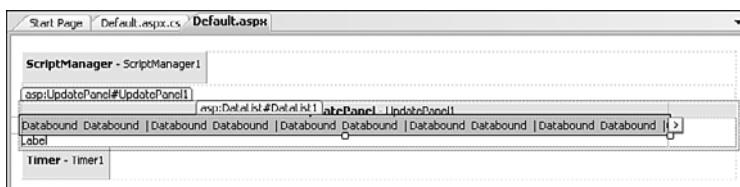


FIGURE 1.4 Just five controls and you have an asynchronous AJAX page.



FIGURE 1.5 A very simple design but the code is actually updating the stock prices every 10 seconds with that postback page flicker.

Because of anonymous types, the code to actually query the stock process from Yahoo! is very short (see Listing 1.11).

LISTING 1.11 This Code Uses `HttpWebRequest` and `HttpWebResponse` to Request Stock Quotes from Yahoo!

```
using System;
using System.Data;
using System.Diagnostics;
using System.Configuration;
using System.Collections;
using System.Linq;
using System.Web;
using System.Web.Security;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;
using System.Web.UI.HtmlControls;
using System.Xml.Linq;
using System.Web.Services ;
using System.Net;
using System.IO;
using System.Text;

namespace DataBindingAnonymousTypes
{
    public partial class _Default : System.Web.UI.Page
    {
        protected void Page_Load(object sender, EventArgs e)
        {
            Update();
        }

        private void Update()
        {
            var quote1 = new {Stock="DELL", Quote=GetQuote("DELL")};
            var quote2 = new {Stock="MSFT", Quote=GetQuote("MSFT")};
            var quote3 = new {Stock="GOOG", Quote=GetQuote("GOOG")};

            var quotes = new object[]{ quote1, quote2, quote3 };
            DataList1.DataSource = quotes;
            DataList1.DataBind();
            Label3.Text = DateTime.Now.ToString("MM/dd/yyyy");
        }
    }

    protected void Timer1_Tick(object sender, EventArgs e)
    {
```

LISTING 1.11 Continued

```
//Update();
}

public string GetQuote(string stock)
{
    try
    {
        return InnerGetQuote(stock);
    }
    catch(Exception ex)
    {
        Debug.WriteLine(ex.Message);
        return "N/A";
    }
}

private string InnerGetQuote(string stock)
{
    string url = @"http://quote.yahoo.com/d/quotes.csv?s={0}&f=pc";
    var request = HttpWebRequest.Create(string.Format(url, stock));

    using(var response = request.GetResponse())
    {
        using(var reader = new StreamReader(response.GetResponseStream(),
            Encoding.ASCII))
        {
            return reader.ReadToEnd();
        }
    }
}
```

The method `InnerGetQuote` has a properly formatted uniform resource locator (URL) query for the Yahoo! stock-quoting feature. Next, an `HttpWebRequest` sends the URL query to Yahoo! Then, the `HttpWebResponse`—returned by `request.GetResponse`—is requested and a `StreamReader` reads the response. Easy, right?

All of this code is run by the `Update` method. `Update` creates anonymous types containing a `Stock` and `Quote` field (which are populated by the `GetQuote` and `InnerGetQuote` methods). An anonymous array of these quote objects is created and all of this is bound to the `DataList`. The `DataList` itself has template controls that are data bound to the `Stock` and `Quote` fields of the anonymous type. Figure 1.6 shows the template design of the `DataList`. The very easy binding statement is shown in Figure 1.7.

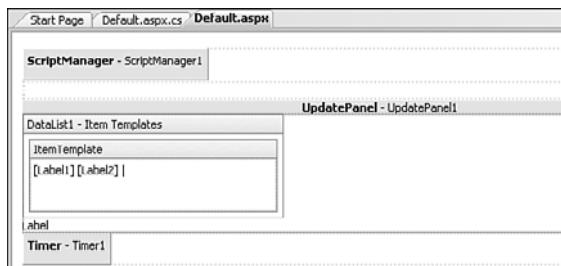


FIGURE 1.6 The template view of the DataList is two Label controls and the | character.

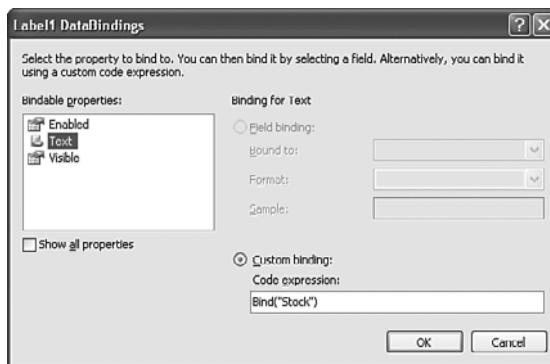


FIGURE 1.7 The binding statements for bound template controls have been very short (as shown) since Visual Studio 2005.

All of the special features, such as template editing and managing bindings, are accessible through the DataList Tasks button, which is shown to the right of the DataList in Figure 1.4. You can also edit elements such as binding statements directly in the ASP designer. Listing 1.12 shows the ASP/HTML for the web page.

LISTING 1.12 The ASP That Creates the Page Shown in Figure 1.4 (Design Time) and Figure 1.5 (Runtime)

```
<%@ Page Language="C#" AutoEventWireup="true" CodeBehind="Default.aspx.cs"
  Inherits="DataBindingAnonymousTypes._Default" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
  <title>Untitled Page</title>
</head>
```

LISTING 1.12 Continued

```

<body>
    <form id="form1" runat="server">
        <asp:ScriptManager ID="ScriptManager1" runat="server">
        </asp:ScriptManager>
        <div>

        </div>
        <asp:UpdatePanel ID="UpdatePanel1" runat="server" EnableViewState="False">
            <ContentTemplate>
                <asp:DataList ID="DataList1" runat="server" RepeatDirection="Horizontal">
                    <itemtemplate>
                        <asp:Label ID="Label1" runat="server" Text='<%# Bind("Stock") %>'>
                            ↪</asp:Label>
                            &nbsp;<asp:Label ID="Label2" runat="server" Text='<%# Bind("Quote") %>'>
                            ↪</asp:Label>
                            &nbsp;;
                    </itemtemplate>
                </asp:DataList>
                <asp:Label ID="Label3" runat="server" Text="Label"></asp:Label>
            </ContentTemplate>
            <triggers>
                <asp:asyncpostbacktrigger ControlID="Timer1" EventName="Tick" />
            </triggers>
        </asp:UpdatePanel>
        <asp:Timer ID="Timer1" runat="server" Interval="10000" ontick="Timer1_Tick">
        </asp:Timer>
    </form>
</body>
</html>

```

The really neat thing about this application (besides getting stock quotes) is that the postbacks happen transparently with AJAX. The way AJAX works is that an asynchronous postback happens and all of the code runs except the part that renders the new page data. Instead, text is sent back and JavaScript updates small areas of the page.

The underlying technology for AJAX is an XMLHttpRequest, and this technology in its raw form has been around for a while. But, the raw form required wiring up callbacks and spinning your own JavaScript. You can still handcraft AJAX code of course, but now there are web controls, such as the UpdatePanel and Timer, that take care of the AJAX plumbing for you.

The elements that initiate the AJAX behavior are called triggers. Triggers can really be any postback event. Listing 1.12 uses the AJAX Timer's Tick event. (And, if you want this to actually look like a ticker, play with some styles and add some color.)

Testing Anonymous Type Equality

Anonymous type equality is defined very deliberately. If any two or more anonymous types have the same order, number, and member declaratory type and name, the same anonymous type class is defined. In this instance, it is permissible to use the referential equality operator on these types. If any of the order, number, and member declarator type and name is different, a different anonymous type definition is defined for each. And, of course, testing referential integrity produces a compiler error.

NOTE

It is possible to use reflection to get type information about anonymous types, and you might want to do this, occasionally, for anonymous types returned from methods.

However, the actual name of the anonymous type can vary between compilations, so devising a way to use the class name probably has no reliable uses.

If you want to test member equality, use the `Equals` method (defined by all objects). Anonymous types with the same order, type, and name, type, and value of member declarators also produce the same hash; the hash is the basis for the equality test. Listing 1.13 provides some samples of anonymous types followed by equality tests and comments indicating those that produce the same anonymous types and those that have member-wise equality.

LISTING 1.13 Various Anonymous Types with Annotations

```
var audioBook = new {Artist="Bob Dylan",
    Song="I Shall Be Released"}; // anonymous type 1
var songBook1 = new {Artist="Bob Dylan",
    Song="I Shall Be Released"}; // also anonymous type 1
var songBook2 = new {Singer="Bob Dylan",
    Song="I Shall Be Released"}; // anonymous type 2
var audioBook1 = new {Song="I Shall Be Released",
    Artist="Bob Dylan"}; // anonymous type 3

audioBook.Equals(songBook1);           // true everything the same
audioBook.Equals(songBook2);           // first member declarators different
songBook1.Equals(songBook2);           // member declarator-names differ
audioBook1.Equals(audioBook);          // member declarators in different orders
```

The anonymous types `audioBook` and `songBook1` produce the same anonymous type. These are the only two that produce the same hash and, as a result, the `Equals` method returns true. The other anonymous types are similar, but either the member declarators are different—`songBook1` uses the member declarator `Artist` and `songBook2` uses `Singer`—or the order of the declarators are different—referring to `audioBook` and `audioBook1`.

Using Anonymous Types with LINQ Queries

The most significant attribute of anonymous types in conjunction with LINQ is that they support *hierarchical data shaping* without writing all of the plumbing code or resorting to SQL. Data shaping is roughly transforming data from one composition to another. LINQ lets you do this with natural queries, and anonymous types give you a place to store the results of these queries.

This whole book is about LINQ, so Listing 1.14 shows a couple of LINQ examples without getting too far ahead in upcoming chapter material. Again, each example also has a brief description.

LISTING 1.14 A Couple of Simple LINQ Queries to Play With Demonstrating Future Topics Such as Sorting and Projections

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace AnonymousTypeWithQuery
{
    class Program
    {
        static void Main(string[] args)
        {
            var numbers = new int[] { 1, 2, 3, 4, 5, 6, 7 };
            var all = from n in numbers orderby n descending select n;

            foreach(var n in all)
                Console.WriteLine(n);

            var songs = new string[]{"Let it be", "I shall be released"};
            var newType = from song in songs select new {Title=song};

            foreach(var s in newType)
                Console.WriteLine(s.Title);

            Console.ReadLine();
        }
    }
}
```

The first query—from `n` in `numbers` orderby `n` descending select `n`—sorts the integers 1 to 7 in reverse order and stuffs the results in the anonymous type `all`. The second query—from `song` in `songs` select new `{Title=song}`—shapes the array of strings in

songs to an enumerable collection of anonymous objects with a property `Title`. (The second example takes an array of strings and shapes it into an array of objects with a well-named property.)

Introducing Generic Anonymous Methods

For newer programmers, word reuse can be confusing. For example, *anonymous* methods are unrelated to anonymous types except to the extent that it means the type of the method is unnamed. Anonymous methods are covered in this section because they are valuable and worth covering, but, for the most part, this section switches topics.

Anonymous methods behave like regular methods except that they are unnamed. They were introduced as an alternative to defining delegates that did very simple tasks, where full-blown methods amounted to more than just extra typing. Anonymous methods also evolved further into *Lambda Expressions*, which are even shorter (terse) methods. Chapter 5, “Understanding Lambda Expressions and Closures,” delves deeper into the evolution of methods. For now, this section takes an introductory look at anonymous *generic* methods.

An anonymous method is like a regular method but uses the `delegate` keyword, and doesn’t require a name, parameters, or return type. Listing 1.15 shows a regular method (used as a delegate for the `CancelKeyPress` event, `Ctrl+C` in a console application) and an anonymous delegate that performs the same role.

LISTING 1.15 A Regular Method and Anonymous Method Handling the `CancelKeyPress` Event in a Console Application

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace AnonymousMethod
{
    class Program
    {
        static void Main(string[] args)
        {
            // ctrl+c
            Console.CancelKeyPress += new ConsoleCancelEventHandler
                (Console_CancelKeyPress);

            // anonymous cancel delegate
            Console.CancelKeyPress +=
                delegate
                {
                    Console.WriteLine("Anonymous Cancel pressed");
                };
        }
    }
}
```

LISTING 1.15 Continued

```
Console.ReadLine();  
  
}  
  
static void Console_CancelKeyPress(object sender, ConsoleCancelEventArgs e)  
{  
    Console.WriteLine("Cancel pressed");  
}  
}  
}
```

TIP

To quickly stub out an event-handling method, type the **object.eventname**, the **+ =** operator, and press the Tab key twice.

The regular method (used as a delegate) is named `ConsoleCancelEventHandler`. Although the double-Tab trick generates these stubbed delegates for you, they are overkill for one-line event handlers. The second statement that begins with the `Console.CancelKeyPress +=` delegate demonstrates an anonymous method (delegate) that is equivalent to the longer form of the method. Notice that because the parameters in the delegate aren't used, they are omitted from the anonymous delegate. You have the option of using the parameter types and names if they are needed in the delegate.

Using Anonymous Generic Methods

Delegates are really just methods that are used (mostly) as event handlers. Generic methods are those that have parameterized types. (Think replaceable data types.) Therefore, anonymous generic delegates are anonymous methods that are associated with replaceable parameterized types. A very useful type is `Func<T>` (and `Func<T, T1, ... Tn>`), demonstrated in Listing 1.16. This generic delegate (defined in the `System` namespace) can be assigned to delegates and anonymous delegates with varying return types and parameters, which makes it a very flexible delegate holder.

LISTING 1.16 Demonstrating How to Use `System.Func` to Define an Essentially Nested Implementation of the Factorial Function

```
using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Text;
```

LISTING 1.16 Continued

```
namespace AnonymousGenericDelegate
{
    class Program
    {
        static void Main(string[] args)
        {
            System.Func<long, long> Factorial =
                delegate(long n)
            {
                if(n==1) return 1;
                long result=1;
                for(int i=2; i<=n; i++)
                    result *= i;
                return result;
            };
            Console.WriteLine(Factorial(6));
            Console.ReadLine();
        }
    }
}
```

For all intents and purposes, `Factorial` is a nested function. Listing 1.16 used `Func<long, long>`, where the first long parameter represents the return type and the second is the parameter. Notice that the listing also used a named parameter for the anonymous delegate.

Implementing Nested Recursion

Now, you can have a little fun bending and twisting the `Factorial` function to use recursion. The challenge is that the named delegate is not named until after the delegate definition—the name being `Factorial`. Hence, you can't use the name in the anonymous delegate itself, but you can make it work.

There is a class called `StackFrame`. `StackFrame` permits getting methods (and information from the call stack) and you can use this class and reflection to invoke the anonymous delegate recursively. (This code is obviously esoteric—referred to this as programmer *esoterism*—but it is fun and demonstrates a lot of features of the framework in a little bit of space, as shown in Listing 1.17.)

LISTING 1.17 Nested, Recursive Anonymous Generic Methods—as a Routine Practice

```
using System;
using System.Diagnostics;
using System.Collections.Generic;
using System.Linq;
```

LISTING 1.17 Continued

```
using System.Text;
using System.Reflection;

namespace AnonymousGenericRecursiveDelegate
{
    class Program
    {
        static void Main(string[] args)
        {
            Func<long, long> Factorial =
                delegate(long n)
                {
                    return n > 1 ?
                        n * (long)(new StackTrace()
                            .GetFrame(0).GetMethod().Invoke(null, new object[]{n-1}))
                        : n;
                };

            Console.WriteLine(Factorial(6));
            Console.ReadLine();
        }
    }
}
```

Again, writing code like the `Factorial` delegate in Listing 1.17 is only fun for the writer, but elements of it do have utility. For example, anonymous delegates like the `Factorial` can be useful for one-time, simple event handling. Assigning behaviors to the `Func<T>` delegate type effectively makes nested functions and reusable delegates that can be passed as arguments, a very dynamic way to program. Getting the `StackFrame` can be a great way to create a utility that tracks function calls during debugging—like writing the `StackTrace` to the Debug window in a way that is useful to you—and reflection has many uses.

Reflection can be useful for dynamically loaded assemblies, as demonstrated by NUnit and Visual Studio's unit testing.

Summary

This chapter examined anonymous types in detail. Anonymous types are strong types where the compiler does the work of figuring out the actual type and writing the class implementation, if the anonymous type is a composite type.

As you see anonymous types used throughout the book for query results, remember anonymous types are immutable, the same type is code generated if the member declaration—field name—type, number, and order are identical.

CHAPTER 2

Using Compound Type Initialization

IN THIS CHAPTER

- ▶ Initializing Objects with Named Types
- ▶ Initializing Anonymous Types
- ▶ Initializing Collections
- ▶ Using Conversion Operators

“Diplomacy is the art of saying ‘Nice doggie’ until you can find a rock.”

—Will Rogers

It is valid to wonder how much work the compiler writer should train the compilers to do for us. The answer might have to be as much as possible that doesn't immediately add to more confusion. (And, saving work is worth a little confusion.)

Some programmers might bridle at the notion of Microsoft (or any compiler writer) taking some work and control out of the hands of programmers. However, if you consider present-day software development, many layers of code are sitting on top of the hardware. These days, programmers generally do not write much assembly code. The Windows application programming interface (API) assembled a lot of disparate function calls into functions that manage many of the rudimentary tasks of Windows programming, and the .NET Framework organizes all of that code into a highly organized object hierarchy. Further, components, controls, and services sit on top of all of this. Yet, programmers still have more code than ever to write on tighter schedules with increasing complexity. And, arguably, software lags hardware; that is, our computers are capable of much more than our programs are asking of them.

All taken together, it means that compiler writers have to do as much work for us as possible and as is useful. This means if you can skip writing things because the compiler can determine what is needed then, the compiler writer

should add this behavior. Compilers doing things for us is especially innocuous if we can choose not to use compiler shortcuts.

This chapter covers compound type initialization. In part, compound initialization eliminates the need to write every flavor of constructor you might need because the .NET 3.5 compiler can infer constructor behavior based on named-type initializing and this code is arguably as clear as verbose, hand-crafted constructors. Initialization is essential for LINQ queries and useful for collections, so those features are discussed, too. In addition, as promised, the little extra rolled in is coverage of conversion operators.

Because *Hello, World!* applications can get a little dull after a while, this chapter includes the complete implementation of a toy like the Spirograph called Hypergraph. Hypergraph draws hypotrochoidal and epitrochoidal figures—or pretty shapes based on mathematical curves—using compound initialization, LINQ queries, and conversion operators.

Initializing Objects with Named Types

Object-initializer syntax permits creating objects and assigning values to fields in a short-hand notation without requiring that specific constructors be defined by the programmer to get the assignment done. Listing 2.1 shows object construction with a default constructor and property assignment. Listing 2.2 shows construction of an object using a constructor defined for the purpose, and Listing 2.3 shows object initialization using the new capability of .NET 3.5. (All of the samples are a derivative of the Hypergraph solution available for download.)

LISTING 2.1 Construction of Two Objects and Property Initialization Using a Verbose Style of Coding

```
private void Form2_Paint(object sender, PaintEventArgs e)
{
    ColoredPoint p1 = new ColoredPoint();
    p1.MyColor = Color.Red;
    p1.MyPoint = new Point(50,50);

    ColoredPoint p2 = new ColoredPoint();
    p2.MyColor = Color.Red;
    p2.MyPoint = new Point(75, 75);
    e.Graphics.DrawLine(Pens.Red, p1.MyPoint, p2.MyPoint);
}
```

LISTING 2.2 Construction of a Pen Object 5 Pixels Wide Using a Constructor with Defined Parameters

```
private void Form2_Paint(object sender, PaintEventArgs e)
{
    ColoredPoint p1 = new ColoredPoint();
```

LISTING 2.2 Continued

```
p1.MyColor = Color.Red;
p1.MyPoint = new Point(50,50);

ColoredPoint p2 = new ColoredPoint();
p2.MyColor = Color.Red;
p2.MyPoint = new Point(75, 75);

Pen pen = new Pen(Color.FromArgb(255, 0, 0), 5);
e.Graphics.DrawLine(pen, p1.MyPoint, p2.MyPoint);
}
```

2

LISTING 2.3 The Paint Event Handler Using Object Initialization with Named Types—the Name of the Properties—Instead of a Constructor and Parameters

```
private void Form2_Paint(object sender, PaintEventArgs e)
{
    ColoredPoint p1 = new ColoredPoint{MyColor=Color.Red, MyPoint=new Point(50,50)};
    ColoredPoint p2 = new ColoredPoint{MyColor=Color.Red, MyPoint=new Point(75, 75)};
    Pen pen = new Pen(Color.FromArgb(255, 0, 0), 5);
    e.Graphics.DrawLine(pen, p1.MyPoint, p2.MyPoint);
}
```

The code in Listing 2.3 is, effectively, emitted as if it had been written like the code in Listing 2.1. Also, the code in Listing 2.3 looks very similar to how `ColoredPoint` objects are constructed with constructors with parameters. The difference is that you do not need to write the verbose form in Listing 2.1, nor do you need to write the constructor, but the intent of this code is as clear as if you had written a two-parameter constructor or the long version (like Listing 2.1).

Listing 2.3 eliminates the long form of construction and writing the constructor—with all the benefits of having done so. Because a lot of what we do in .NET is write constructors and initialize objects, object initialization with named types saves us from writing a lot of dull code.

The basic behavior of using *initialization with named types* is to use ordinary object declaration, or anonymous type declaration and the `new` keyword and type on the left side of the operator. However, instead of using parentheses and parameter values, you can use property names, the assignment operator, and the value. The general syntax of named type initialization is as follows:

```
Class variable = new Class{Property1=value1, etc};
```

See Listing 2.3 for an example. When you use this form of object initialization, you can selectively choose to provide initializers for only those properties you are interested in.

Implementing Classes for Compound Initialization Through Named Types

Using named parameters is not completely new to .NET. Named types have been used to initialize attributes since .NET 1.0. Initializing objects by naming public properties rather than parameters to a constructor looks natural and is intuitive. This feature is nice syntactic sugar for everyday types but is critical for anonymous types returned from LINQ queries.

With queries, named initialization lets us pick and choose the properties in the source and map them to new property names in the target of the query, a projection. (We'll return to this subject in a minute.) To demonstrate named typed initialization, take a look at Listing 2.4, which shows the `ColoredPoint` class for the Hypergraph and notice how you can pick and choose which properties to initialize when you construct `ColoredPoint` objects. (Listing 2.4 contains the complete `ColoredPoint` class. `ColoredPoint` is a wrapper class for a `Point` and a `Color`.)

LISTING 2.4 The `ColoredPoint` Class for the Hypergraph Is Used to Represent Colored Points

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Drawing;

namespace Hypergraph
{
    public class ColoredPoint
    {
        // Auto-implemented property
        public Color MyColor { get; set; }

        // can't use because we need to modify X and Y properties
        private PointF myPoint;
        public PointF MyPoint
        {
            get
            {
                return myPoint;
            }
            set
            {
                myPoint = value;
            }
        }

        public float X
```

LISTING 2.4 Continued

```
{  
    get  
    {  
        return myPoint.X;  
    }  
    set  
    {  
        myPoint.X = value;  
    }  
}  
  
public float Y  
{  
    get  
    {  
        return myPoint.Y;  
    }  
    set  
    {  
        myPoint.Y = value;  
    }  
}  
  
/// <summary>  
/// Initializes a new instance of the ColoredPoint class.  
/// </summary>  
public ColoredPoint()  
{  
}  
  
public override string ToString()  
{  
    return MyPoint.ToString();  
}  
  
internal ColoredPoint Clone()  
{  
    return new ColoredPoint{MyColor=this.MyColor, MyPoint=this.MyPoint};  
}  
}
```

The `ColoredPoint` class supports storing a `Point` and a `Color`. (The `Color` exists to support multicolored mathematical shapes but is not used.) More important, note that there is no

constructor defined that accepts a `Color` or `Point`. They aren't needed. You can use named-typed initialization and just not write that code.

Named-type initialization is demonstrated in the `Clone` method. `Clone` names the properties to initialize and makes a copy of the `ColoredPoint` object. (In practice, you could use the predefined `MemberWiseClone` object for a shallow copy instead of writing your own `Clone` procedure.)

There is another interesting feature in Listing 2.4, the `MyColor` property. `MyColor` uses auto-implemented properties, which are covered in the next section.

Understanding Auto-Implemented Properties

`MyColor` has an empty getter and setter. At first, it looks just like a property definition in an interface. However, when the .NET 3.5 compiler sees this code, it implements the property automatically, hence the *nom de guerre*. If you don't need to access a field directly and you are just getting and setting an underlying field property, which is pretty common, you can let the .NET compiler implement a private backing field for you. When you use auto-implemented properties, the private backing field is only accessible through the getter and setter.

NOTE

Writing do-nothing properties is a boring chore but considered a best practice. Instead of spinning your wheels writing these do-nothing property getters and setters, let the compiler do it. However, you need to write some properties, and CodeRush from DevExpress takes a lot of the sting out of writing code in general.

Attributes are not allowed on auto-implemented properties, so roll your own if you need an attribute. Also, you can implement read-only auto-implemented properties by prefixing the `set;` keyword with the `private` modifier.

```
public type name{ get; private set;}
```

Finally, notice that Listing 2.4's `ColoredPoint` class does not use the auto-implemented feature for the `MyPoint` property. This is because the constituent properties of the `Point` class, `X` and `Y`, were surfaced and the field `myPoint` was referred to. Even if you used an auto-implemented property for `MyPoint` and referred to the property `MyPoint.X` to surface the constituent property `X`, it's a compiler error. If you do more than set a field, implement the property yourself.

Initializing Anonymous Types

Anonymous types can only be initialized with object initializer syntax. That's because there is no formally defined type until the code is compiled. Object initialization can also be used to change the property names in the anonymous type.

TIP

Support for renaming properties in anonymous types might prove a useful technique for renaming poorly named properties from generated or legacy code and for pretty printing in reports and grids.

This ability to query existing types, rename (or project) a new name on the anonymous type's properties, is referred to as *shaping* and the resultant anonymous type is referred to as a *projection*. The code in Listing 2.5 uses a query to project the generic `List<Person>` and projects it to an anonymous type referred to by `names` and changes the `Name` property in the `Person` class to `FirstName`.

LISTING 2.5 Projecting a New Type from Person and Renaming a Property in the Initializer Clause

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace AnonymousTypeInitializer
{
    class Program
    {
        static void Main(string[] args)
        {
            List<Person> people = new List<Person>
            {
                new Person{ID=1, Name="Paul", Email="pkimmel@softconcepts.com"},
                new Person{ID=2, Name="Joe", Email="jswan@gmail.com"}
            };

            var names = from p in people select new {FirstName=p.Name};
            foreach(var name in names)
                Console.WriteLine(name.FirstName);
            Console.ReadLine();

        }
    }

    public class Person
    {
        private int iD;
        public int ID
        {
            get { return iD; }
        }
    }
}
```

LISTING 2.5 Continued

```

    set { iD = value; }
}

private string name;
public string Name
{
    get { return name; }
    set { name = value; }
}

private string email;
public string Email
{
    get { return email; }
    set { email = value; }
}
}
}
}

```

Listing 2.5 defines a `Person` class. The statement beginning with `var names` uses a LINQ query to select all of the `Name` values. (Don't worry about the query too much right now.) The initializer `{FirstName=p.Name}` means to initialize the resultset as an anonymous type with a property named `FirstName`. The `foreach` statement shows how we now refer to the `Name` values through the projected `FirstName` property.

If you leave the `name` part out of the initialization clause, the projected type uses the same name as the initializing type. In Listing 2.5, this means objects in the anonymous types in `names` would have a `Name` property.

Initializing Collections

Initialization syntax is extended to collections, too. In previous versions of .NET, when you wanted to create Lists of objects, you first created the list, then each object, adding the objects one at a time to the collection (List or whatever). Listing 2.5's initialization of the `people` List demonstrates how to initialize the `List<Person>` and all of the `Person` objects (also using initialization syntax), adding all of the `Person` objects to the List in one fell swoop.

TIP

Properties of value types cannot be initialized using object initializer syntax.

If you use a tool like Lutz Roeder's .NET Reflector (see Figure 2.1), you can quickly determine that the compiler has converted the collection initialization code from Listing 2.5 into the long form of List construction, object construction, and calls to the List's `Add`

method. Over the course of an application collection initialization, you will save many lines of code.

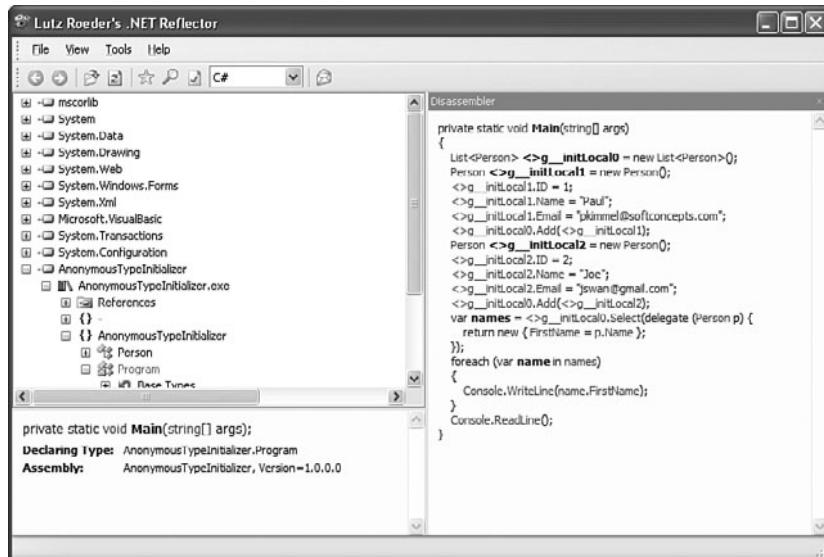


FIGURE 2.1 Reflector's disassembler clearly shows that the collection initialization code is converted by the compiler into the verbose form of construction and a call to Add.

You can combine object initialization and LINQ queries and implement the `ColoredPointList`. `ColoredPointList` (Listing 2.6) inherits from `List<T>` and uses a query and a conversion operator to convert the typed list of `ColoredPoint` objects to an array of `PointF` objects. This approach was used because it is very easy to store objects in Lists, but GDI+ uses arrays of points to draw arcs, which is what the Hypergraph images (see Figure 2.2) are composed of.

LISTING 2.6 The `ColoredPointList` Uses Queries and Conversion Operators to Make Storing the Points Easy and Converting Them to a Form Necessary for GDI+'s Drawing Methods

```

using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Drawing;

namespace Hypergraph
{
    public class ColoredPointList : List<ColoredPoint>
    {

```

LISTING 2.6 Continued

```
/// <summary>
/// Use conversion operator ToArray
/// </summary>
/// <returns></returns>
public PointF[] GetPoints()
{
    return (from p in this select p.MyPoint).ToArray<PointF>();
}

public ColoredPointList Clone()
{
    ColoredPointList list = new ColoredPointList();
    list.AddRange((from o in this select o.Clone()));
    return list;
}
}
```

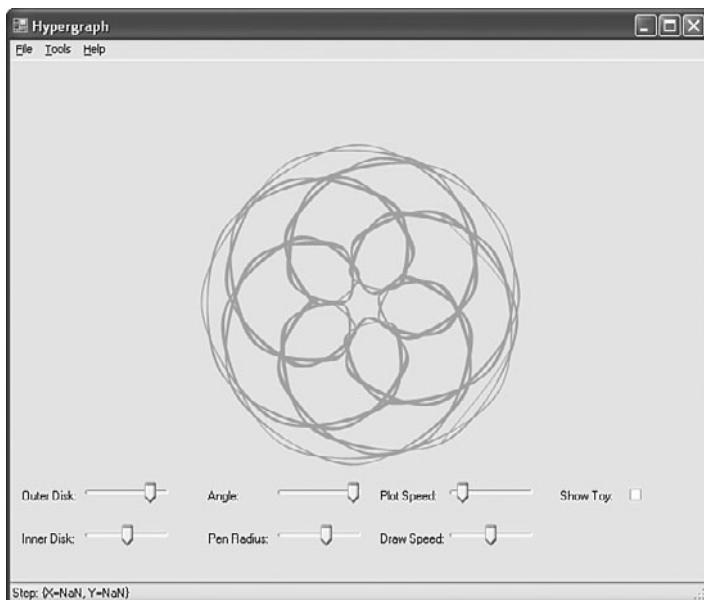


FIGURE 2.2 The Hypergraph toy with one of the rendered images.

The `GetPoints` method uses a LINQ query that selects all of the `Point` objects from all of the `ColoredPoints`. The query actually returns an instance of `System.Linq.Enumerable.SelectIterator<T>` of `ColoredPoints` and when you access this iterator, you get the types described in the `select` clause. `ToArray<T>` is a conversion

operator (via an extension method) that converts the elements of the query to an array. Extension methods are discussed in more detail in Chapter 3, “Defining Extension and Partial Methods.”

Finishing the Hypergraph

To help you explore the Hypergraph, Listing 2.7 includes the code for the Hypergraph itself and Listing 2.8 contains the IHypergraph interface. The interface is used as a means of associating Hypergraph with the slider controls (on a UserControl), shown at the bottom of Figure 2.2.

LISTING 2.7 The Complete Listing of the Hypergraph Class

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Text;
using System.Drawing;
using System.Drawing.Imaging;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

namespace Hypergraph
{
    public class Hypergraph : IHypergraph
    {
        private static Random random = new Random(DateTime.Now.Millisecond);
        private static float NextRandom(float scalar)
        {
            return (float)random.NextDouble() * scalar;
        }

        private static int NextRandom(int scalar)
        {
            return random.Next(scalar);
        }

        /// <summary>
        /// Didn't use auto-implemented fields here by choice
        /// </summary>
        private float angle = NextRandom(5f);
        public float Angle
        {
            get { return angle; }
            set { angle = value; }
        }
    }
}
```

LISTING 2.7 Continued

```
private float speed = NextRandom(1f);
public float Speed
{
    get { return speed; }
    set { speed = value; }
}

private float centerX = NextRandom(200f);
public float CenterX
{
    get { return centerX; }
    set { centerX = value; }
}

private float centerY = NextRandom(200f);
public float CenterY
{
    get { return centerY; }
    set { centerY = value; }
}

private float innerRingRadius = NextRandom(81f);
public float InnerRingRadius
{
    get { return innerRingRadius; }
    set { innerRingRadius = value; }
}

private float outerRingRadius = NextRandom(100f);
public float OuterRingRadius
{
    get { return outerRingRadius; }
    set { outerRingRadius = value; }
}

private float angleOfPen = NextRandom(3.5f);
public float AngleOfPen
{
    get { return angleOfPen; }
    set { angleOfPen = value; }
}

private float penRadialDistanceFromCenter = NextRandom(70f);
public float PenRadialDistanceFromCenter
{
```

LISTING 2.7 Continued

```
get { return penRadialDistanceFromCenter; }
set { penRadialDistanceFromCenter = value; }
}

/// <summary>
/// Can't use auto-implemented type here because we have behaviors
/// </summary>
private Color penColor = Color.FromArgb(NextRandom(255), NextRandom(255),
    NextRandom(255));
public Color PenColor
{
    get { return penColor; }
    set
    {
        penColor = value;
        MyPen = new Pen(value);
    }
}

/// <summary>
/// Auto-implemented field only accessible here
/// </summary>
public Pen MyPen{ get; set; }

/// <summary>
/// No auto-implemented fields here because construction of object field
/// is easier without it
/// </summary>
private ColoredPointList list = new ColoredPointList();
public ColoredPointList List
{
    get { return list; }
}

public void Clear()
{
    Broadcaster.Broadcast("Clear");
    list.Clear();
}

public PointF[] Points
{
    get { return list.GetPoints(); }
}
```

LISTING 2.7 Continued

```
public ColoredPoint PlotNextPoint()
{
    angle += speed;
    angleOfPen = angle - angle * outerRingRadius / innerRingRadius;

    // object initializer with named types
    ColoredPoint point = new ColoredPoint{MyColor=penColor,
        X=GetX(), Y=GetY()};

    list.Add(point);
    const string mask = "Step: {0}";
    Broadcaster.Broadcast(mask, point.ToString());
    return point;
}

private float GetCsx()
{
    return (float)(centerX + Math.Cos(angle)
        * (outerRingRadius - innerRingRadius));
}

private float GetCsy()
{
    return (float)(centerY + Math.Sin(angle)
        * (outerRingRadius - innerRingRadius));
}

private float GetX()
{
    return (float)(GetCsx() + Math.Cos(angleOfPen)
        * penRadialDistanceFromCenter);
}

private float GetY()
{
    return (float)(GetCsy() + Math.Sin(angleOfPen) *
        penRadialDistanceFromCenter);
}

public RectangleF OuterRingBoundsRect()
{
    return new RectangleF(centerX-outerRingRadius,
        centerY - outerRingRadius,
        2 * outerRingRadius, 2 * OuterRingRadius);
}
```

LISTING 2.7 Continued

```
public RectangleF GetAxis()
{
    return new RectangleF(GetX(), GetY(),
        innerRingRadius, innerRingRadius);
}

public RectangleF InnerRingBoundsRect()
{
    return new RectangleF(GetCx() - innerRingRadius,
        GetCy() - innerRingRadius, 2 * innerRingRadius,
        2 * innerRingRadius);
}

/// <summary>
/// Initializes a new instance of the Hypergraph class.
/// </summary>
/// <param name="angle"></param>
/// <param name="speed"></param>
/// <param name="centerX"></param>
/// <param name="centerY"></param>
/// <param name="innerRingRadius"></param>
/// <param name="outerRingRadius"></param>
/// <param name="angleOfPen"></param>
/// <param name="penRadialDistanceFromCenter"></param>
/// <param name="penColor"></param>
/// <param name="list"></param>
public Hypergraph(float angle, float speed,
    float centerX, float centerY, float innerRingRadius,
    float outerRingRadius, float angleOfPen,
    float penRadialDistanceFromCenter, Color penColor,
    ColoredPointList list)
{
    this.angle = angle;
    this.speed = speed;
    this.centerX = centerX;
    this.centerY = centerY;
    this.innerRingRadius = innerRingRadius;
    this.outerRingRadius = outerRingRadius;
    this.angleOfPen = angleOfPen;
    this.penRadialDistanceFromCenter = penRadialDistanceFromCenter;
    this.penColor = penColor;
    this.list = list;
}

/// <summary>
```

LISTING 2.7 Continued

```
/// Initializes a new instance of the Hypergraph class.  
/// </summary>  
public Hypergraph(Hypergraph o)  
{  
    this.angle = o.angle;  
    this.speed = o.speed;  
    this.centerX = o.centerX;  
    this.centerY = o.centerY;  
    this.innerRingRadius = o.innerRingRadius;  
    this.outerRingRadius = o.outerRingRadius;  
    this.angleOfPen = o.angleOfPen;  
    this.penRadialDistanceFromCenter = o.penRadialDistanceFromCenter;  
    this.penColor = o.penColor;  
    this.list = o.List.Clone();  
}  
  
/// <summary>  
/// Initializes a new instance of the Hypergraph class.  
/// </summary>  
public Hypergraph()  
{  
    MyPen = new Pen(Color.FromArgb(NextRandom(255), NextRandom(255),  
        NextRandom(255)));  
}  
  
public Hypergraph Clone()  
{  
    return new Hypergraph(this);  
}  
  
private GraphicsPath path = new GraphicsPath();  
public GraphicsPath Path  
{  
    get  
    {  
        path.Reset();  
        if(List.Count > 2) path.AddPolygon(Points);  
        path.CloseAll();  
        return path;  
    }  
}  
  
public void SaveToyToFile(string filename)  
{  
    Broadcaster.Broadcast("Saving to {0}", filename);  
    Hypergraph copy = this.Clone();
```

LISTING 2.7 Continued

```
foreach (var p in copy.List)
{
    p.MyPoint = new PointF(p.MyPoint.X - copy.CenterX +
        (copy.OuterRingRadius + copy.InnerRingRadius) / 2,
    p.MyPoint.Y - copy.CenterY +
        (copy.OuterRingRadius + copy.InnerRingRadius) / 2);
}

GraphicsPath path = copy.Path;
RectangleF bounds = path.GetBounds();
Bitmap bitmap = new Bitmap((int)bounds.Width, (int)bounds.Height);
Graphics graphics = Graphics.FromImage(bitmap);
graphics.DrawCurve(MyPen, copy.Points);
bitmap.Save(filename, ImageFormat.Jpeg);
Broadcaster.Broadcast("Saved");
}

public void Draw(Graphics g)
{
    if(Points.Length < 2) return;
    Broadcaster.Broadcast("Drawing");
    g.SmoothingMode = SmoothingMode.AntiAlias;
    g.DrawCurve(MyPen, this.Points);
    if (toyVisible) ShowToy(g);
}

private void ShowToy(Graphics g)
{
    g.DrawEllipse(Pens.Silver, OuterRingBoundsRect());
    g.DrawEllipse(Pens.Silver, InnerRingBoundsRect());
    RectangleF rect = GetAxis();
    g.FillEllipse(new SolidBrush(PenColor), rect.X, rect.Y, 6, 6);
}

internal void CenterImage(int width, int height)
{
    Clear();
    CenterX = width / 2;
    CenterY = height / 2;
    Broadcaster.Broadcast("Centered image at {0} x {1}", CenterX, CenterY);
}

#region IHypergraph Members

private bool toyVisible;
```

LISTING 2.7 Continued

```

public bool ToyVisible
{
    get
    {
        return toyVisible;
    }
    set
    {
        toyVisible = value;
    }
}

#endregion

internal void Plot(int p)
{
    for(var i=0; i<p; i++)
        this.PlotNextPoint();
}
}
}

```

LISTING 2.8 The IHypergraph Interface Is Used to Support the Observer Pattern Between the UserControl with Sliders and the Hypergraph Class

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Drawing;
using System.Text;

namespace Hypergraph
{
    public interface IHypergraph
    {
        float InnerRingRadius { get; set; }
        float OuterRingRadius { get; set; }
        float AngleOfPen { get; set; }
        float PenRadialDistanceFromCenter { get; set; }
        float Speed { get; set; }
        bool ToyVisible { get; set; }
        void Clear();
        Color PenColor { get; set; }
    }
}

```

LISTING 2.8 Continued

```
public interface IHypergraphObserver
{
    IHypergraph Subject {set;}
}
```

2

Implementing the Hypergraph Controls Using the Observer Pattern

Listing 2.9 shows the implementation the HypergraphController (see Figure 2.3). This `UserControl` contains all of the sliders that permit manipulating the Hypergraph instance. To make this interaction orderly, an interface and the Observer pattern is used. Observer exists simply to make it easy to manipulate the state of any object that implements `IHypergraph`. This is done through binding to delegates. Listing 2.10 shows the implementation of the subject and observer interfaces.



FIGURE 2.3 The HypergraphController.

LISTING 2.9 The Implementation of the HypergraphController

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Drawing;
using System.Data;
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace Hypergraph
{
    public partial class HypergraphController : UserControl, IHypergraphObserver
    {
        public HypergraphController()
        {
            InitializeComponent();
        }

        #region IsubjectObserver Members
        private IHypergraph subject;
        public IHypergraph Subject

```

LISTING 2.9 Continued

```
{  
    set  
    {  
        subject = value;  
        SubjectChanged();  
    }  
}  
  
private void SubjectChanged()  
{  
    if(subject == null ) return;  
    TrackBarOuterDisk.Value = (int)subject.OuterRingRadius;  
    TrackBarInnerDisk.Value = (int)subject.InnerRingRadius;  
    TrackBarAngle.Value = (int)subject.AngleOfPen;  
    TrackBarPenRadius.Value = (int)subject.PenRadialDistanceFromCenter;  
}  
  
#endregion  
  
private void TrackBarOuterDisk_Scroll(object sender, EventArgs e)  
{  
    Changed();  
}  
  
private void TrackBarInnerDisk_Scroll(object sender, EventArgs e)  
{  
    Changed();  
}  
  
private void TrackBarDrawSpeed_Scroll(object sender, EventArgs e)  
{  
    if(timer != null)  
        timer.Interval = TrackBarDrawSpeed.Value;  
}  
  
private Timer timer;  
public Timer Timer  
{  
    get  
    {  
        return timer;  
    }  
    set  
    {  
        timer = value;  
    }  
}
```

LISTING 2.9 Continued

```
if(timer != null)
    TrackBarDrawSpeed.Value = timer.Interval;
}

private void TrackBarAngle_Scroll(object sender, EventArgs e)
{
    Changed();
}

private void TrackBarPenRadius_Scroll(object sender, EventArgs e)
{
    Changed();
}

private void CheckBoxShowToy_CheckedChanged(object sender, EventArgs e)
{
    if(subject!=null)
        subject.ToyVisible = CheckBoxShowToy.Checked;
}

private void TrackBarPlotSpeed_Scroll(object sender, EventArgs e)
{
    Changed();
}

private void Changed()
{
    if(subject == null) return;
    subject.Clear();
    subject.OuterRingRadius = TrackBarOuterDisk.Value;
    toolTip1.SetToolTip(TrackBarInnerDisk, TrackBarInnerDisk.Value.ToString());
    subject.InnerRingRadius = TrackBarInnerDisk.Value;
    toolTip1.SetToolTip(TrackBarInnerDisk, TrackBarInnerDisk.Value.ToString());
    subject.AngleOfPen = TrackBarAngle.Value;
    toolTip1.SetToolTip(TrackBarAngle, TrackBarAngle.Value.ToString());
    subject.PenRadialDistanceFromCenter = TrackBarPenRadius.Value;
    toolTip1.SetToolTip(TrackBarPenRadius, TrackBarPenRadius.Value.ToString());
    subject.Speed = TrackBarDrawSpeed.Value;
    toolTip1.SetToolTip(TrackBarDrawSpeed, TrackBarDrawSpeed.Value.ToString());
}

internal void Random()
{
```

LISTING 2.9 Continued

```

Random random = new Random(DateTime.Now.Millisecond);
TrackBarAngle.Value = random.Next(TrackBarAngle.Minimum,
    TrackBarAngle.Maximum);
TrackBarInnerDisk.Value = random.Next(TrackBarInnerDisk.Minimum,
    TrackBarInnerDisk.Maximum);
TrackBarOuterDisk.Value = random.Next(TrackBarOuterDisk.Minimum,
    TrackBarOuterDisk.Maximum);
TrackBarPenRadius.Value = random.Next(TrackBarPenRadius.Minimum,
    TrackBarPenRadius.Maximum);
TrackBarPlotSpeed.Value = random.Next(TrackBarPlotSpeed.Minimum,
    TrackBarPlotSpeed.Maximum);
if(subject != null)
    subject.PenColor = Color.FromArgb(random.Next(255), random.Next(255),
        random.Next(255));
Changed();
}
}
}

```

LISTING 2.10 The Implementation of the Subject and Observer, Which Exist to Make It Easy to Associate a Subject with the Object That Observes It

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Drawing;
using System.Text;

namespace Hypergraph
{
    public interface IHypergraph
    {
        float InnerRingRadius { get; set; }
        float OuterRingRadius { get; set; }
        float AngleOfPen { get; set; }
        float PenRadialDistanceFromCenter { get; set; }
        float Speed { get; set; }
        bool ToyVisible { get; set; }
        void Clear();
        Color PenColor { get; set; }
    }

    public interface IHypergraphObserver
    {

```

LISTING 2.10 Continued

```
IHypergraph Subject {set;}  
}  
}
```

2

Using Conversion Operators

A very common task is converting data from one form to another. Historically, this entailed creating the target object and copying data from the source argument and initializing new instances of the target object. This is necessary but tedious (and time-consuming) work and converting code is prevalent.

Prevalent, tedious, and time-consuming work cries out for convenience tools. The new version of .NET has answered this cry. The following sections explore the conversion operators `ToArray`, `OfType`, `Cast`, `AsEnumerable`, `ToList`, `ToDictionary`, and `ToLookup`.

ToArray

Arrays are fast but not very convenient. It is cumbersome to write array management code when collections like `List<T>` make managing data easier. However, many legacy APIs, existing code, and some algorithms are designed to use or are locked into arrays. A perfect example is the Hypergraph toy. It is much easier to store points in a collection, but GDI+ methods like `DrawCurve` are designed to use arrays. To get the ease of use of collections but use an array when drawing the arcs, you can call `ToArray` on the `List` of `ColoredPoints`.

The elided class fragments in Listing 2.11 shows the complete `GetPoints` method, which uses a LINQ query and `ToArray` and the `SaveToyToFile` method. The code uses the `GetPoints` method to convert the list of points to an array and `SaveToyToFile` uses a bitmap and the points to save the image to an external file. For completeness, the Broadcaster—which uses the Observer form of publish subscribe, or what is sometimes called the *broadcast-listener*—pattern is provided in Listing 2.12.

LISTING 2.11 The Combination of a LINQ Query, the `ToArray` Conversion Method, and GDI+ to Save an Image to a File

```
public class ColoredPointList : List<ColoredPoint>  
{  
  
    /// <summary>  
    /// Use conversion operator ToArray  
    /// </summary>  
    /// <returns></returns>  
    public PointF[] GetPoints()  
    {
```

LISTING 2.11 Continued

```
var points = from p in this select p.MyPoint;
return points.ToArray<PointF>();
//return (from p in this select p.MyPoint).ToArray<PointF>();
//return (from p in this select p.MyPoint).ToArray<PointF[]>();

//PointF[] points = new PointF[this.Count];
//for(int i=0; i<this.Count; i++)
//  points[i] = this[i].MyPoint;

//return points;
}

...
}

Public class Hypergraph
{
public PointF[] Points
{
    get { return list.GetPoints(); }
}
public void SaveToyToFile(string filename)
{
    Broadcaster.Broadcast("Saving to {0}", filename);
    Hypergraph copy = this.Clone();
    foreach (var p in copy.List)
    {
        p.MyPoint = new PointF(p.MyPoint.X - copy.CenterX +
            (copy.OuterRingRadius + copy.InnerRingRadius) / 2,
            p.MyPoint.Y - copy.CenterY +
            (copy.OuterRingRadius + copy.InnerRingRadius) / 2);
    }

    GraphicsPath path = copy.Path;
    RectangleF bounds = path.GetBounds();
    Bitmap bitmap = new Bitmap((int)bounds.Width, (int)bounds.Height);
    Graphics graphics = Graphics.FromImage(bitmap);
    graphics.DrawCurve(MyPen, copy.Points);
    bitmap.Save(filename, ImageFormat.Jpeg);
    Broadcaster.Broadcast("Saved");
}
...
}
```

LISTING 2.12 The Broadcast-Listener Behavior Pattern, an Implementation of the Observer Pattern Is Used to Move Messages from Anywhere in the System to the Presentation Layer

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Hypergraph
{
    public interface IListener
    {
        void Listen(string message);
        bool Listening { get; }
    }

    public class Broadcaster
    {
        private static List<IListener> listeners = new List<IListener>();
        public static void Add(IListener listener)
        {
            listeners.Add(listener);
        }

        public static void Broadcast(string message)
        {
            foreach(var listener in listeners)
                listener.Listen(message);
        }

        public static void Broadcast(string format, params object[] o)
        {
            Broadcast(string.Format(format, o));
        }
    }
}
```

For example, if any class implements `IListener`, then registered Listeners—`IListeners` added to the Broadcasters internal list—will receive status messages from anywhere in the system that broadcasts a message. In the Hypergraph example, the main `Form` implements `IListener` and uses the string messages to update the `Form`'s `StatusBar`.

OfType

The conversion operator `OfType<T>` returns an `IEnumerable<T>` collection of only the types defined by the parameter `T`. For example, initializing a list of objects where some are integers, you can quickly extract just the integers, as demonstrated in Listing 2.13. Listing 2.13 returns an `IEnumerable<int>` with 1 and 4 in the resultset.

LISTING 2.13 The `OfType<T>` Conversion Operator Quickly Extracts Only Elements Whose Type Matches the Template Parameter `T`'s Type

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace OfTypeDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            var numbers = new object[]{1, "two", null, "3", 4};
            foreach(var anInt in numbers.OfType<int>())
                Console.WriteLine(anInt);
            Console.ReadLine();
        }
    }
}
```

Cast

To receive an exception if elements of the source type cannot be converted to the target type, use the `Cast <T>` conversion operator. If you use numbers from Listing 2.13, you'll receive an `InvalidOperationException` when the `Cast` operator hits the string "two" (see Listing 2.14).

LISTING 2.14 Use `Cast<T>` When You Want to Receive an `InvalidOperationException` If Elements of the Source Do Not Match the `typeof T`

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace CastDemo
```

LISTING 2.14 Continued

```
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            var numbers = new object[]{1, "two", null, "3", 4};  
            foreach(var anInt in numbers.Cast<int>())  
                Console.WriteLine(anInt);  
            Console.ReadLine();  
  
        }  
    }  
}
```

2

AsEnumerable

Many types implement `IEnumerable`. Some of these types implement public members that are identical to `IEnumerable`. For instance, if you have a type `MyList` that implements a `Where` method (as does `IEnumerable<T>`), invoking `Where` on `MyList` would call `MyList`'s implementation of `Where`. By calling the `AsEnumerable` method first and then calling `Where`, you invoke `IEnumerable`'s `Where` method instead of `MyList`'s (see Listing 2.15).

LISTING 2.15 AsEnumerable Forces an Object That Implements `IEnumerable` to Use the Behaviors of the `IEnumerable` Interface

```
using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Text;  
  
namespace AsEnumerable  
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            const string Bacon = "For it is the solecism of a Prince " +  
                "to think to control the end yet not endure the mean.";  
  
            MyList<string> strings = new MyList<string>();  
            strings.AddRange(Bacon.Split());  
            IEnumerable<string> inMyList = strings.Where(str=>str=="solecism");
```

LISTING 2.15 Continued

```

foreach(string s in inMyList)
    Console.WriteLine(s);

Console.ReadLine();

IEnumerable<string> notInMyList =
    strings.AsEnumerable().Where(str=>str=="Prince");

foreach(string s in notInMyList)
    Console.WriteLine(s);
Console.ReadLine();

}

}

public class MyList<T> : List<T>
{
    public IEnumerable<T> Where(Func<T, bool> predicate)
    {
        Console.WriteLine("MyList");
        return Enumerable.Where(this, predicate);
    }
}
}

```

In Listing 2.15, the first `Where` call invokes `MyList`'s `Where`. After the call to `AsEnumerable()`, the next call is to `List<T>`'s `Where` as part of `List<T>`'s implementation of `IEnumerable`.

ToList

The `ToList` conversion operator forces an immediate query evaluation and stores the results in a `List<T>`. Listing 2.16 shows an anonymous type of college football teams, a LINQ query whose results are converted to a `List<T>`, and some additional teams are added.

LISTING 2.16 Converting Query Results to a `List<T>` Lets Us Add Additional Elements to the Resultset

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace ToListDemo
{

```

LISTING 2.16 Continued

```
class Program
{
    static void Main(string[] args)
    {
        var teams = new string[]{"Spartans", "Fighting Irish", "Wolverines"};
        List<string> footballTeams = (from name in teams select name).
            ➔ToList<string>();
        footballTeams.Add("Hoosiers");
        footballTeams.Add("Fighting Illini");
        footballTeams.Add("Badgers");
        Console.WriteLine(footballTeams.Count);
        Console.ReadLine();
    }
}
```

2

ToDictionary

A dictionary is a collection of name and value pairs. `ToDictionary` converts an `IEnumerable<T>` object—such as is returned from a LINQ query—into an `IDictionary<Key, Value>` object. The `ToDictionary` method used in Listing 2.17 uses a selector `Func<Game, string>` to set the keys. The selector used is the Lambda Expression `Key=>Key.Opponent`, which essentially makes the Opponent name the key and the Game object itself the value part of the pair. Key is an instance of Game and Opponent is the string part of the Func generic. (For more information on Lambda Expressions, see Chapter 5, “Understanding Lambda Expressions and Closures.”)

LISTING 2.17 Converting a `List<T>` to an `IDictionary<K,V>`, a Dictionary of Name and Value Pairs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ToDictionaryDemo
{
    class Program
    {
        public class Game
        {
            public string Opponent{get; set; }
            public string Score{get; set; }
```

LISTING 2.17 Continued

```

    }

    static void Main(string[] args)
    {
        var spartans = new List<Game>{
            new Game{Opponent="UAB", Score="55-18"},
            new Game{Opponent="Bowling Green", Score="28-17"},
            new Game{Opponent="Pittsburgh", Score="17-13"},
            new Game{Opponent="Notre Dame", Score="17-14"}};

        //var games = from g in spartans select g;

        IDictionary<string, Game> stats = spartans
            .ToDictionary(Key=>Key.Opponent);
        Console.WriteLine("Spartans vs. {0} {1}", stats["Notre Dame"].Opponent,
            stats["Notre Dame"].Score);
        Console.ReadLine();
    }
}
}

```

ToLookup

`ToLookup` converts an `IEnumerable<T>` to a `Lookup<Key, Element>` type. `Lookup` is like a dictionary, but where a `Dictionary` uses a single key value, `Lookup` maps keys to a collection of values. `Lookups` have no public constructor and are immutable. You cannot add or remove elements or keys after they are created.

Listing 2.18 defines a `Product` class containing a `Code` and `Description`—in a real application, you might use `SKU`, or stock-keeping unit. Next, we request a `Lookup<Key, Element>` from the `List` of `Products`—`List<Product>`. The key for the `Lookup` is defined by the Lambda Expression `c=>c.Substring(0,3)`, which keys the `Lookup` on the first three characters of the `Product.Code`. Finally, the `Lookup` values are displayed in groups by the key using the `IGrouping` interface, and the last bit of code shows how to get just one group by the key.

LISTING 2.18 Using the `Lookup<Key, Element>` Type and the `IGrouping` Interface

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ToLookupDemo
{

```

LISTING 2.18 Continued

```
class Program
{
    public class Product
    {
        public string Code{ get; set; }
        public string Description{ get; set; }
    }

    static void Main(string[] args)
    {
        List<Product> products = new List<Product>
        {
            new Product{Code="SPM1", Description="Spam"},
            new Product{Code="SPM2", Description="Mechanically Separated Chicken"},
            new Product{Code="LME1", Description="Bologna"},
            new Product{Code="SUP1", Description="Tomato"},
            new Product{Code="SUP2", Description="Chicken Noodle"}};

        Lookup<string, string> lookup = (Lookup<string,string>)
            products.ToLookup(
                c=>c.Code.Substring(0,3), c=>c.Code + " " + c.Description);

        foreach(IGrouping<string, string> group in lookup)
        {
            Console.WriteLine(group.Key);
            foreach(string s in group)
                Console.WriteLine(" {0}", s);
        }

        Console.ReadLine();

        IEnumerable<string> spmGroup = lookup["SPM"];
        foreach(var str in spmGroup)
            Console.WriteLine(str);

        Console.ReadLine();
    }
}
```



Summary

Compound type initialization makes it easier to initialize arrays and lists and is an essential capability for initializing anonymous types. Conversion operators take some of the tedium out of converting between convenient types and other types. More important, this chapter demonstrates the *classic onion layering* of complexity in .NET that makes it possible to do a lot of work with relatively few lines of code.

This chapter also introduced or demonstrated and provided additional examples of Lambda Expressions, LINQ queries, conversion operators, the Observer pattern, and provided some examples that used GDI+. Lambda Expressions are discussed in detail in Chapter 5, “Understanding Lambda Expressions and Closures,” and LINQ queries are explored in Chapter 6, “Using Standard Query Operators.”

CHAPTER 3

Defining Extension and Partial Methods

“Do you see a man skilled in his work? He will serve before kings; he will not serve before obscure men.”

—Proverbs 22:29

Conceptually, think of extension methods as an implementation of the *Decorator Structural pattern*. If you check <http://www.dofactory.com>, you will see that the actual decorator is intended to add behavior dynamically using inheritance and composition. The ToolTip control is a closer technical match to the decorator, but extension methods are logically close enough.

Extension methods are designed to support adding behaviors that look like they belong to an existing class when you can't add behaviors. For example, you can't inherit and add behaviors for *sealed* classes, and you can't inherit and add behaviors to intrinsic types like `int` and `string`.

This chapter looks at extension methods, including how to implement them and how they support Language INtegrated Query (LINQ), and partial methods, including where and how they are used.

Extension Methods and Rules of the Road

Historically, extension methods have been used to add new behaviors when possible and wrapper classes have been used when inheritance couldn't be used. Extension methods clean up the clumsy syntax of wrapper methods and permit you to extend sealed classes and intrinsic types.

IN THIS CHAPTER

- ▶ Extension Methods and Rules of the Road
- ▶ Defining Extension Methods
- ▶ How Extension Methods Support LINQ
- ▶ Implementing a “Talking” String Extension Method
- ▶ Defining Partial Methods

In addition, extension methods can help you avoid deep inheritance trees. Deep inheritance trees get hard to manage, and the extension method provides another way to add a feature without full-blown inheritance.

Listing 3.1 demonstrates how you can add an extension method called `Dump` to an object state dumper for debugging purposes. `Dump` uses `Reflection` to display the internal state of all the properties of a class.

LISTING 3.1 An Extension Method That Displays the State of Any Object, Useful for Debugging Purposes

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Reflection;
using System.Diagnostics;

namespace ExtensionMethodDump
{
    class Program
    {
        static void Main(string[] args)
        {
            var song = new {Artist="Jussi Bjorling", Song="Aida"};
            song.Dump();
            Console.ReadLine();
        }
    }

    public static class Dumper
    {
        public static void Dump(this Object o)
        {
            PropertyInfo[] properties = o.GetType().GetProperties();
            foreach(PropertyInfo p in properties)
            {
                try
                {
                    Debug.WriteLine(string.Format("Name: {0}, Value: {1}", p.Name,
                        p.GetValue(o, null)));
                }
                catch
                {

```

LISTING 3.1 Continued

```
        Debug.WriteLine(string.Format("Name: {0}, Value: {1}", p.Name,
            "unk."));
    }
}
}
}
}
```

In Listing 3.1, an anonymous type is defined containing an `Artist` and `Song`, Jussi Björling interpretation of “Aida.” Because `Dump` operates upon type `object` and every type inherits from `object`, the anonymous type has access to `Dump`. Calling `Dump` on the anonymous type referred to by `song` displays the artist and song to the Debug window.

TIP

To make `Dump` really practical, add a method that accepts an `IList` and check for array properties.

As with any feature, some rules of the road define when and how a feature can be used. The following list describes how extension methods can be used and provides useful information about them:

- ▶ Use extension methods to extend sealed types without inheritance.
- ▶ Use extension methods to keep deep inheritance hierarchies from getting out of hand.
- ▶ Extension methods are static methods in static classes (see Listing 3.1).
- ▶ The first argument of an extension method must use the `this` modifier preceding the argument type; `this` refers to the type being extended.
- ▶ Extension properties, events, and operators are not supported (but might be in the future).
- ▶ Extension methods are less discoverable and more limited than instance methods.
- ▶ Extension methods are invoked using instance syntax.
- ▶ Extension methods have a lower precedence than regular methods; therefore, if a class has a similarly named method, the instance method will be called.
- ▶ Extension methods are supported by IntelliSense.
- ▶ Generic extension methods are supported, which is, in fact, how many of the LINQ keywords are implemented in the `System.Linq` namespace.
- ▶ Extension methods can be invoked on literals; assuming you had an extension method `public static void SayIt(this string what)`, you could invoke `SayIt` with `"Hello World".SayIt()`.

- ▶ Extension methods can be used on sealed classes and intrinsic types like `int` because extension methods automatically support boxing and unboxing; that is, when value types are used like objects, the .NET Framework wraps a class around the value type.
- ▶ Extension methods are not really members, so you can only access public members of the object being extended.
- ▶ An extension method implicitly uses the `ExtensionAttribute`; the `ExtensionAttribute` is used explicitly in VB .NET.

The term you will hear associated with extension methods is *duct typing*. Pun intended.

Defining Extension Methods

Extension methods follow a consistent pattern. Define a public static class. In that class, define a public static method. The first argument of the method must use the `this` modifier. The first argument, modified by `this`, represents the class of the object being extended. After the first argument, add additional arguments. Static methods can be overloaded by parameters and can be generic methods. This section explores extension methods, overloaded extension methods, and generic extension methods.

Implementing Extension Methods

Listing 3.1 demonstrated a basic extension method, and the previous section covered the basic guidelines for extension methods. It's now time to look at extension methods that have return types and extension methods with multiple parameters.

Listing 3.2 contains a variation on the `Dump` extension method to include a return type. In Listing 3.2, a `StringBuilder` is used to compile the properties and return them as a string. The `StringBuilder` is used here because strings are immutable in .NET, so the `StringBuilder` is the way to go when building text output. Listing 3.3 demonstrates how to use additional parameters in extension methods. In Listing 3.3, the second parameter to `Dump` is a `TextWriter`; this approach permits passing in `Console.Out` as the `TextWriter`, sending the content to the console.

LISTING 3.2 Defining an Extension Method with a Return Type

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Reflection;
using System.Diagnostics;
```

LISTING 3.2 Continued

```
namespace ExtensionWithReturn
{
    class Program
    {
        static void Main(string[] args)
        {
            var songs = new{
                Artist="Green Day", Song="Wake Me Up When September Ends"};

            Console.WriteLine(songs.Dump());
            Console.ReadLine();
        }
    }

public static class Dumper
{
    public static string Dump(this Object o)
    {
        PropertyInfo[] properties = o.GetType().GetProperties();
        StringBuilder builder = new StringBuilder();
        foreach(PropertyInfo p in properties)
        {
            try
            {
                builder.AppendFormat(string.Format("Name: {0}, Value: {1}", p.Name,
                    p.GetValue(o, null)));
            }
            catch
            {
                builder.AppendFormat(string.Format("Name: {0}, Value: {1}", p.Name,
                    "unk."));
            }
            builder.AppendLine();
        }
        return builder.ToString();
    }
}
```



LISTING 3.3 Adding Additional Parameters to the Extension Method; As Demonstrated Here, You Can Pass in `Console.Out` as the `TextWriter` Parameter

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Reflection;
using System.Diagnostics;
using System.IO;

namespace ExtensionWithAdditionalParameters
{
    class Program
    {
        static void Main(string[] args)
        {
            var song = new{
                Artist="Avril Lavigne", Song="My Happy Ending"
            };

            song.Dump(Console.Out);
            Console.ReadLine();
        }
    }

    public static class Dumper
    {
        public static void Dump(this Object o, TextWriter writer)
        {
            PropertyInfo[] properties = o.GetType().GetProperties();
            foreach(PropertyInfo p in properties)
            {
                try
                {
                    writer.WriteLine(string.Format("Name: {0}, Value: {1}", p.Name,
                        p.GetValue(o, null)));
                }
            }
        }
    }
}
```

LISTING 3.3 Continued

```
        }
    catch
    {
        writer.WriteLine(string.Format("Name: {0}, Value: {1}", p.Name,
            "unk."));
    }
}
}
```



Overloading Extension Methods

You also have the option of overloading extension methods. Listing 3.4 has two extension methods named `Dump`. The first accepts the extended type object and dumps the single-object properties, and the second `Dump` extends an `IList`, iterating over each item in a collection and calling `Dump` on that item.

LISTING 3.4 `Dump` Is Overloaded to Extend Object and `IList`; the Framework and Compiler Determine Which Version to Call

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Reflection;
using System.Diagnostics;

namespace OverloadedExtensionDump
{
    class Program
    {
        static void Main(string[] args)
        {
            var songs = new[]
            {
                new {Artist="Jussi Bjorling", Song="Aida"},
                new {Artist="Sheryl Crow", Song="Steve McQueen"}
            };

            songs.Dump();
            Console.ReadLine();
        }
    }
}
```

LISTING 3.4 Continued

```
        }
    }

public static class Dumper
{
    public static void Dump(this Object o)
    {
        PropertyInfo[] properties = o.GetType().GetProperties();
        foreach(PropertyInfo p in properties)
        {
            try
            {
                Debug.WriteLine(string.Format("Name: {0}, Value: {1}", p.Name,
                    p.GetValue(o, null)));
            }
            catch
            {
                Debug.WriteLine(string.Format("Name: {0}, Value: {1}", p.Name,
                    "unk."));
            }
        }
    }

    public static void Dump(this IList list)
    {
        foreach(object o in list)
            o.Dump();
    }
}
```

Before object-oriented programming and overloading, you had to write code like `DumpObject` and `DumpList`. The challenge is that such code becomes cumbersome very quickly. With overloading and an understanding that `object` is the root type for all types and that many kinds of collections implement `IList`, you can support just writing `Dump` in its various forms and let the compiler track which specific version to use.

Defining Generic Extension Methods

Extension methods start to get really powerful when combined with generics. An ever-present thorn in the side of developers is writing all of the plumbing that initializes null but necessary entity objects—open a connection, call the stored procedure, read every record, read every field checking for null, and initialize each object, adding it to the collection. This code is terribly dull after the ten-thousandth time. Listing 3.5 combines a generic extension method for a base class called the EntityClass and a single generic SafeRead<T> method that reads any kind of field and if it is not null, SafeRead returns that value. If the field is null, a default value is provided.

TIP

Microsoft is working on a better relational-to-entity mapping solution. Refer to Chapter 17, “Introducing ADO.NET 3.0 and the Entity Framework,” for more on this subject.

Listing 3.5 solves an ongoing pickle of a problem. In pure object-oriented programming, objects should be self-initializing. However, to make entity objects self-initializing, you would have to carry ADO.NET around with your entities. Using an extension method lets entities look self-initializing, but you can put the plumbing outside of the entity class itself.

LISTING 3.5 The Generic Extension Method SafeRead<T> Lets Your Entities Behave Like a Self-Initializing Database Class (or Entities) While Keeping the Plumbing Separate

```
using System;using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data;
using System.Data.SqlClient;
using System.Reflection;
using System.Diagnostics;
using System.IO;

namespace GenericExtensionMethods
{
    class Program
    {
        static void Main(string[] args)
        {
            string connectionString = "Data Source=CASPAR;Initial Catalog=
=>Northwind;Integrated Security=True";
            // create the list of orders
            List<Order> orders = new List<Order>();
```

LISTING 3.5 Continued

```
// open and protect the connection
using(SqlConnection connection = new SqlConnection(connectionString))
{
    // prepare to select all orders
    SqlCommand command = new SqlCommand("SELECT * FROM ORDERS", connection);
    command.CommandType = CommandType.Text;
    connection.Open();
    SqlDataReader reader = command.ExecuteReader();

    // read all orders
    while(reader.Read())
    {
        Order order = new Order();
        // the extension method does the legwork but it looks like order
        // is self-initializing
        order.Read(reader);
        orders.Add(order);
    }
}

// use the dumper to send everything to the console
orders.Dump(Console.Out);
Console.ReadLine();
}

public static class ReaderHelper
{
    public static void Read(this Order order, IDataReader reader)
    {
        order.OrderID = order.SafeRead(reader, "OrderID", -1);
        order.CustomerID = order.SafeRead(reader, "CustomerID", "");
    }

    /// <summary>
    /// One reader checks for all the null possibilities
    /// </summary>
    /// <typeparam name="T"></typeparam>
    /// <param name="entity"></param>
    /// <param name="reader"></param>
    /// <param name="fieldName"></param>
    /// <param name="defaultValue"></param>
```

LISTING 3.5 Continued

```
/// <returns></returns>
public static T SafeRead<T>(this EntityClass entity,
    IDataReader reader, string fieldName, T defaultValue)
{
    try
    {
        object o = reader[fieldName];
        if(o == null || o == System.DBNull.Value)
            return defaultValue;

        return (T)Convert.ChangeType(o, defaultValue.GetType());
    }
    catch
    {
        return defaultValue;
    }
}

public static class Dumper
{
    public static void Dump(this Object o, TextWriter writer)
    {
        PropertyInfo[] properties = o.GetType().GetProperties();
        foreach(PropertyInfo p in properties)
        {
            try
            {
                writer.WriteLine(string.Format("Name: {0}, Value: {1}", p.Name,
                    p.GetValue(o, null)));
            }
            catch
            {
                writer.WriteLine(string.Format("Name: {0}, Value: {1}", p.Name,
                    "unk."));
            }
        }
    }
}

public static void Dump<T>(this IList<T> list, TextWriter writer)
{
    foreach(object o in list)
        o.Dump(writer);
```

LISTING 3.5 Continued

```
        }
    }

public class EntityClass{}
public class Order : EntityClass
{
    /// <summary>
    /// Initializes a new instance of the Order class.
    /// </summary>
    public Order()
    {
    }

private int orderID;
public int OrderID
{
    get
    {
        return orderID;
    }
    set
    {
        orderID = value;
    }
}

private string customerID;
public string CustomerID
{
    get
    {
        return customerID;
    }
    set
    {
        customerID = value;
    }
}
```

In Listing 3.5, the `Main` method sets up a pretty typical database read. The difference is that it uses a couple of extension methods that make entity classes look self-initializing while putting that plumbing in the `ReaderHelper` extension class.

`ReaderHelper` has an extension method for `Order` objects that contains the information about the `Orders` table (in the Northwind database). Information about the `Orders` table is only needed when you are actually interacting with the database, so you don't really need to carry that around with the `Order` objects. `SafeRead<T>` is an extension class that manages reading the actual field value while performing all of that painful checking for null.

Finally, the `Dumper` class is recycled and defines a simple entity based on part of the `Northwind.Orders` table.



How Extension Methods Support LINQ

As you have seen, LINQ looks a lot like Structured Query Language (SQL). In fact, however, LINQ is fully integrated into the .NET Framework. This is done partly through the `System.Linq` namespace. The `System.Linq` namespace is defined in the `System.Core.dll` assembly and contains some very complicated-looking extension methods (in the `Queryable` class), such as

```
public static IQueryable<TSource> Where<TSource>(
    this IQueryable<TSource> source, Expression<Func<TSource, bool>> predicate)
```

and

```
public static IQueryable<TResult> Select<TSource, TResult>(
    this IQueryable<TSource> source, Expression<Func<TSource, TResult>> selector)
```

These methods look complicated—and for those who put off learning generics, probably do seem complicated—but what can be gleaned from the method headers is that these are generic extension methods that extend `IQueryable`. And, generics are used a lot here. (These method headers are like generics gone wild, but it is pretty clear that LINQ might have evolved to make it much easier to actually use all of these capabilities.)

If you take the `Where` method and blot out the parameterized arguments, you have the following:

```
public static IQueryable Where(this IQueryable source, Expression predicate)
```

This is a little easier to read. `Where` is an extension method that extends `IQueryable`, returns `IQueryable`, and takes an `Expression` that acts as the determining predicate—that is, what is in the resultset. If you add in all of the parameterized arguments, the code just means it's flexible enough to work with whatever kinds of data the `IQueryable` object contains.

In short, this means that LINQ's underpinnings are based on extension methods (and generics), so that when you write a LINQ statement, the compiler is actually emitting the call to the more complex-looking generic methods. Listing 3.6 contains a simple LINQ query that returns even numbers from an array. Listing 3.7 shows the Microsoft Intermediate Language (MSIL or IL)—which was retrieved using Intermediate Language Disassembler (ILDASM)—emitted that contains the more verbose calls to the extension methods in `System.Linq`.

LISTING 3.6 A Simple-Looking Query That Emits Calls to Extension Methods Like `System.Linq.Enumerable.Where` in the `System.Core.dll` Assembly

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace QueryWithWhere
{
    class Program
    {
        static void Main(string[] args)
        {
            var numbers = new int[]{1966, 1967, 1968, 1969, 1970};
            var evens = from num in numbers where num % 2 == 0 select num;

            foreach(var result in evens)
                Console.WriteLine(result);

            Console.ReadLine();
        }
    }
}
```

LISTING 3.7 The MSIL from ILDASM That Reflects How the Much Easier (in Comparison) LINQ Query Is to Write Than Direct Calls to These Methods

```
.method private hidebysig static void Main(string[] args) cil managed
{
    .entrypoint
    // Code size      121 (0x79)
    .maxstack 4
    .locals init ([0] int32[] numbers,
```

LISTING 3.7 Continued

```
[1] class [mscorlib]System.Collections.Generic.IEnumerable`1<int32> evens,
[2] int32 result,
[3] class [mscorlib]System.Collections.Generic.IEnumerator`1<int32>
➥CS$5$0000,
[4] bool CS$4$0001)
IL_0000:  nop
IL_0001:  ldc.i4.5
IL_0002:  newarr     [mscorlib]System.Int32
IL_0007:  dup
IL_0008:  ldtoken    field valuetype '<PrivateImplementationDetails>
➥{13096E17-8E04-40AD-AD54-10AC18376A40}'/'__StaticArrayInitTypeSize=20'
➥'<PrivateImplementationDetails>
➥{13096E17-8E04-40AD-AD54-10AC18376A40}':::$method0x6000001-1'
IL_000d:  call         void [mscorlib]
➥System.Runtime.CompilerServices.RuntimeHelpers::InitializeArray(class
➥[mscorlib]System.Array,
valuetype [mscorlib]System.RuntimeFieldHandle)
IL_0012:  stloc.0
IL_0013:  ldloc.0
IL_0014:  ldsfld     class [System.Core]System.Func`2<int32,bool>
QueryWithWhere.Program::'<>9__CachedAnonymousMethodDelegate1'
IL_0019:  brtrue.s   IL_002e
IL_001b:  ldnull
IL_001c:  ldftn      bool QueryWithWhere.Program::'<Main>b__0'(int32)
IL_0022:  newobj      instance void class
➥[System.Core]System.Func`2<int32,bool>::ctor(object,native int)
IL_0027:  stsfld      class [System.Core]System.Func`2<int32,bool>
➥QueryWithWhere.Program::'<>9__CachedAnonymousMethodDelegate1'
IL_002c:  br.s        IL_002e
IL_002e:  ldsfld     class [System.Core]System.Func`2<int32,bool>
➥QueryWithWhere.Program::'<>9__CachedAnonymousMethodDelegate1'
IL_0033:  call         class [mscorlib]System.Collections.Generic. IEnumerable`1<!0>
➥[System.Core]System.Linq.Enumerable::Where<int32>(class [mscorlib]System.
Collections.Generic.IEnumerable`1<!0>,
class [System.Core]System.Func`2<!0,bool>)
IL_0038:  stloc.1
IL_0039:  nop
IL_003a:  ldloc.1
IL_003b:  callvirt    instance class
➥[mscorlib]System.Collections.Generic.IEnumerator`1<!0> class
➥[mscorlib]System.Collections.Generic.IEnumerable`1<int32>::GetEnumerator()
IL_0040:  stloc.3
.try
```

LISTING 3.7 Continued

```

{
    IL_0041: br.s      IL_0051
    IL_0043: ldloc.3
    IL_0044: callvirt  instance !0 class
    ↪[mscorlib]System.Collections.Generic.IEnumerator`1<int32>::get_Current()
    IL_0049: stloc.2
    IL_004a: ldloc.2
    IL_004b: call      void [mscorlib]System.Console::WriteLine(int32)
    IL_0050: nop
    IL_0051: ldloc.3
    IL_0052: callvirt  instance bool
    ↪[mscorlib]System.Collections.IEnumerator::MoveNext()
    IL_0057: stloc.s   CS$4$0001
    IL_0059: ldloc.s   CS$4$0001
    IL_005b: brtrue.s  IL_0043
    IL_005d: leave.s   IL_0071
} // end .try
finally
{
    IL_005f: ldloc.3
    IL_0060: ldnnull
    IL_0061: ceq
    IL_0063: stloc.s   CS$4$0001
    IL_0065: ldloc.s   CS$4$0001
    IL_0067: brtrue.s  IL_0070
    IL_0069: ldloc.3
    IL_006a: callvirt  instance void [mscorlib]System.IDisposable::Dispose()
    IL_006f: nop
    IL_0070: endfinally
} // end handler
IL_0071: nop
IL_0072: call      string [mscorlib]System.Console::ReadLine()
IL_0077: pop
IL_0078: ret
} // end of method Program::Main

```

The use of the `Where` method is shown in bold. Preparation of the `Func<T, TResult>` delegates as arguments to the expression predicate are also discernible. Although it is unlikely that it is or will be necessary to use the long-winded method calls over LINQ, Listing 3.8 is an expanded version of Listing 3.6, which shows what the code might look like without LINQ.

LISTING 3.8 The Long Form of Listing 3.6 Using the Verbose and Expanded Version of Some of the Elements, such as the Func Delegate and the Where Extension Method

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace QueryWithWhereLongForm
{
    class Program
    {
        static void Main(string[] args)
        {
            var numbers = new int[]{1966, 1967, 1968, 1969, 1970};
            Func<int, bool> getEvents = delegate(int num)
            {
                return num % 2 == 0;
            };

            IEnumerable<int> evens = numbers.Where<int>(getEvents);

            IEnumerator<int> enumerator = evens.GetEnumerator();
            while(enumerator.MoveNext())
                Console.WriteLine(enumerator.Current);

            Console.ReadLine();
        }
    }
}
```

You could also use some variations for Listing 3.8. You could have not used anonymous types. You could substitute the anonymous delegate for a Lambda Expression, but you might agree that the LINQ query is the more concise and clear form of the solution (in Listing 3.6).

Implementing a “Talking” String Extension Method

At TechEd 2007, David West and Ivar Jacobson (one of the three amigos of the Rational Unified Process [RUP] fame and the inventor of use cases) said no one reads books. Buy them, yes. Read them, no.

Reading and writing is both informative and enjoyable. Sure, Google and Wikipedia and many more websites can be useful for solving programming problems, but these sites are not always accessible or portable—it’s tough to Google in the tub—so nothing beats a book for portability, tactile sensation, and smell. (New books smell as good as new cars.) That said, reasons to write are to enjoy the learning process, have fun with the code, and sell books to readers that will enjoy them. (All book sales go to my yacht, *Chillin’ the Most*, fund, and you’ll get invited on a three-hour cruise.) That’s what this section is about—programming is fun.

To complete Listing 3.9, a reference to the Microsoft Speech Object Library 5.0 (see Figure 3.1) is added to a console application. After the COM Interop assembly—which is generated automatically—is added to the solution, you can add a using statement for the `SpeechLib`. Finally, an extension method `Say` for strings is added in the `MakeItTalk` class, and the `SpVoiceClass` is used to set the voice and rate of speech and send the text to the speakers.

LISTING 3.9 Adding a Reference to the `SpeechLib` (the Microsoft Speech Object Library 5.0 in `sapi.dll`) and Making Your Strings Talk

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using SpeechLib;

namespace TalkingStringClass
{
    class Program
    {
        static void Main(string[] args)
        {
            "Microsoft Surface is Cool!".Say();
            "Press Enter to Quit".Say();
            Console.ReadLine();
        }
    }

    public static class MakeItTalk
    {
```

LISTING 3.9 Continued

```

public static void Say(this string s)
{
    SpVoice voice = new SpVoiceClass();
    // required attributes - no attributes are required
    // optional attributes - nonrequired
    ISpeechObjectTokens tokens = voice.GetVoices("", "");
    voice.Rate = 2;
    voice.Volume = 100;
    voice.Voice = tokens.Item(0);
    voice.Speak(s, SpeechVoiceSpeakFlags.SVSFDefault);
}
}
}
}

```

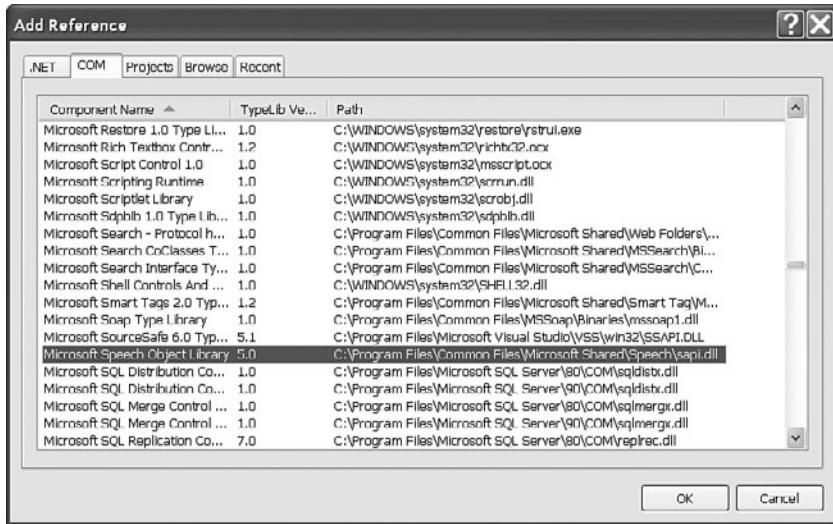


FIGURE 3.1 Adding references to COM libraries.

You can use this code in VB or code very similar to it in JavaScript for your web clients. Refer to the article, "Talking Web Clients with JavaScript and the Speech API" at developer.com: <http://www.developer.com/lang/jscript/article.php/3688966>.

Defining Partial Methods

One of the things authors always have to do is figure out how to use something. One of the things we sometimes have to do is figure out why the heck anyone would use that something. Partial methods was one of those *what the heck do you do with it* moments, but,

then, Bill Wagner of *Effective C#* (Addison-Wesley Professional, 2004) fame told me something during our user group meeting (Greater Lansing Area .NET Users Group, or Glugnet, which I think sounds like a beer-drinking club).

Bill said, “Partial methods are placeholders for generated code.”

“Oh, like SAP user exits. I get it.” That was my reply, and comparing anything with SAP is ghastly, but that’s what they are.

The problem with code-generated code has typically been that if the consumer changed the generated code and regenerated the code, the user’s changes were overwritten. One solution to this problem was to inherit from generated code and write custom changes in the child class. However, partial methods let the producer stub out the method signature with a partial method. If the consumer wants to insert some behavior at that point, the consumer provides an implementation for the partial method. If no implementation is provided, the partial method is stripped out. This is clever and useful for code generators. (The producer and consumer can be the same programmer, of course.)

The following are some basic guidelines for partial methods:

- ▶ Partial methods are declared in partial classes.
- ▶ Partial methods use the `partial` modifier.
- ▶ Partial methods don’t have a method body at the point of introduction.
- ▶ Partial methods must return `void`.
- ▶ Partial methods can be static, and can have arguments and argument modifiers, including `ref` and `params` (an array of arguments).
- ▶ Partial methods are private, but no literal access modifiers are permitted; that is, you can’t use the `private` keyword explicitly.
- ▶ Unused partial methods are stripped out by the compiler.

The example in Listing 3.10 uses a partial method.

LISTING 3.10 A Partial Method Demo

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Web.UI.WebControls;
using System.Reflection;
using System.Diagnostics;
```

```
namespace PartialMethodDemo
{
    class Program
```

LISTING 3.10 Continued

```
{  
    static void Main(string[] args)  
    {  
        CustomerList customers = new CustomerList  
        {  
            new Customer{CustomerID=1, Name="Levi, Ray and Shoup"},  
            new Customer{CustomerID=2, Name="General Dynamics Land Systems"},  
            new Customer{CustomerID=3, Name="Microsoft"}  
        };  
  
        customers.Dump();  
        Console.ReadLine();  
    }  
}  
  
  
public partial class CustomerList  
{  
    private string propertyName;  
    private SortDirection direction;  
  
    partial void SpecialSort(string propertyName, SortDirection direction)  
    {  
        this.propertyName = propertyName;  
        this.direction = direction;  
        Sort(Comparer);  
    }  
  
  
    /// <summary>  
    /// Using an integer to change the direction of the sort was a suggestion made by  
    /// Chris Chartran, my good friend from Canada  
    /// </summary>  
    /// <param name="x"></param>  
    /// <param name="y"></param>  
    /// <returns></returns>  
    private int Comparer(Customer x, Customer y)  
    {  
        try  
        {  
            PropertyInfo lhs = x.GetType().GetProperty(propertyName);  
            PropertyInfo rhs = y.GetType().GetProperty(propertyName);  
        }  
    }
```



LISTING 3.10 Continued

```
int directionChanger = direction == SortDirection.Ascending ? 1 : -1;

object o1 = lhs.GetValue(x, null);
object o2 = rhs.GetValue(y, null);

if(o1 is IComparable && o2 is IComparable)
{
    return ((IComparable)o1).CompareTo(o2) * directionChanger;
}

// no sort
return 0;
}
catch(Exception ex)
{
    Debug.WriteLine(ex.Message);
    return 0;
}
}

public partial class CustomerList : List<Customer>
{
    partial void SpecialSort(string propertyName, SortDirection direction);

    public void Dump()
    {
        SpecialSort("CustomerID", SortDirection.Descending);

        foreach(var customer in this)
            Console.WriteLine(customer.ToString());
    }
}

public class Customer
{
```

LISTING 3.10 Continued

```
private int customerID;
public int CustomerID
{
    get
    {
        return customerID;
    }
    set
    {
        customerID = value;
    }
}

private string name;
public string Name
{
    get
    {
        return name;
    }
    set
    {
        name = value;
    }
}

public override string ToString()
{
    return string.Format("CustomerID={0}, Name={1}", CustomerID, Name);
}
```

The code in Listing 3.10 is entirely for convenience. The `Main` procedure uses compound type initialization to initialize a list of customers. Then, the collection is dumped; dumping object state is something you might do while debugging and testing.

The partial class `CustomerList` represents the part that fills in the partial method `SpecialSort`. `SpecialSort` accepts a property name and a sort direction—courtesy of Chris Chartrand—and uses `Reflection` to sort on the named property. Because `SpecialSort` is defined as a partial method, you could elect not to implement it or let some other consumer implement in a manner deemed suitable for that project's needs.



The comparison is done using `Reflection` in the `Comparer` method. The `SortDirection` is used to determine whether to multiply the result by 1 or -1. Remember, `CompareTo` returns an integer where -1 means less than, 0 means equal to, and 1 means greater than. Thus, if you multiply the result by -1, you change the direction of the sort.

Finally, the `CustomerList` class inherits from `List<Customer>`. You might use this technique to add special behaviors to your typed collections. As you can see, the `Dump` method calls `SpecialSort`, which, if implemented, yields ordered output. The last part of Listing 3.10 is routine; the `Customer` class just rounds out the demo.

Summary

In the 1960s, I remember being at the State Fair in Detroit and my first ride on the Tilt-a-whirl. For a little kid, it was one of those “We aren’t in Kansas anymore, Toto” kind of moments. Cool new language features are still like that to me.

This chapter discussed extension methods. Extension methods are used heavily to support LINQ. The result is that instead of using generic-intensive code, you can use a more natural, querylike language, LINQ. You are encouraged to master generics, but LINQ is much easier to use than the multiparameter extension methods on which it is built.

To recap, extension methods permit you to add behaviors to intrinsic types, sealed classes, and keep deep inheritance trees under control. This chapter also looked at partial methods, which are used mostly in generated code to provide placeholders for consumer code.

CHAPTER 4

yield return: Using .NET's State Machine Generator

“Always bear in mind that your own resolution to succeed is more important than any one thing.”

—Abraham Lincoln

Just a few short years ago, developers had to inherit from `ReadOnlyCollectionBase` or `CollectionBase` to implement strongly typed collections. If you wanted to define custom types, instead of using `DataSets`, then this was the way to go. With the introduction of generics, you no longer have to do that. However, until recently developers still had to write by hand all of the plumbing that created subtypes from those collections. Generally, this consisted of iterating over a loop, applying some filtering criteria, and then extracting only those objects into a new collection. This plumbing is routine and dull to implement (after a few hundred times). With `yield return`, you no longer have to write your own sublists.

Using the key phrase `yield return`, the .NET Framework (since 2.0) now code generates a state machine for us. The state machine is essentially an enumerable typed collection. The benefit is that you can focus on primary domain types and rest assured that support for subsets of this type are accessible through `yield return` or LINQ queries. In many cases, you will want to use LINQ queries for subsets, but `yield return` is worth exploring here.

This chapter looks at `yield return` and `yield break`. You'll see how to use these constructs and take a peek at the code generated by these simple phrases.

IN THIS CHAPTER

- ▶ Understanding How `yield return` Works
- ▶ Using `yield return` and `yield break`

Shared Source Code Supports Debugging Framework

Microsoft is releasing the source code for the .NET 3.5 Framework. This code will be available for download and can be integrated into Visual Studio for drill-down debugging into the framework. For example, if you hit some .NET code while stepping—using Debug, Step Into—Visual Studio will download the needed .NET code and step into it automatically.

To set up this auto-download and drill-down feature in Visual Studio, select Tools, Options; expand the Debugging, Symbols feature; and add the following URL to the symbol file locations list and a local cache folder: <https://source.microsoft.com/symbols>. You can use Figure 4.1 as a visual guide. To check on availability, stay tuned to Scott Guthrie's blog at <http://weblogs.asp.net/scottgu/>.

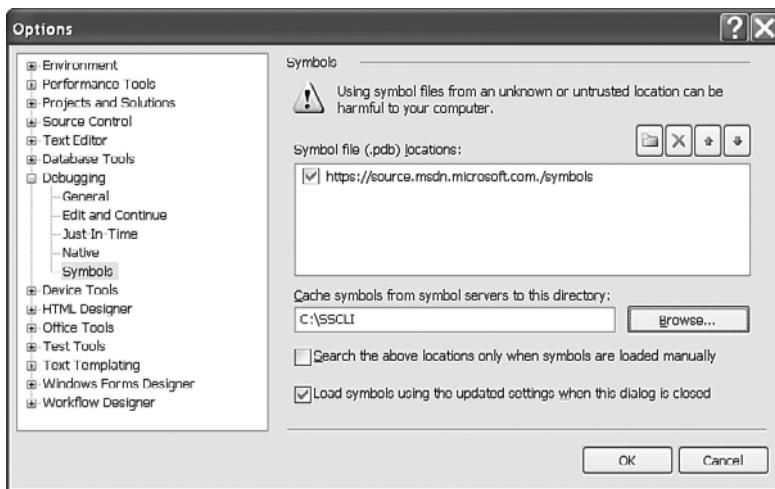


FIGURE 4.1 You can drill down into the .NET Framework code if you really want to know how things work in .NET.

Understanding How `yield return` Works

The word `yield` by itself is not a keyword. This means we can still use `yield` as a variable, which might be helpful to financial people. However, `yield return` and `yield break` are key phrases.

Together, `yield return` is a compiler cue that tells the compiler to generate a state machine. Every time a line containing `yield return` is hit, the generated code will return that value. What happens is that the compiler generates an enumerator that adds the `WhileMoveNext` return-current plumbing without stack unwinding. This means that as long as there are more items to return, the stack is not unwound and `finally` blocks are not executed. When the last item is hit—when `MoveNext` returns `false`—the stack is unwound and `finally` blocks are executed.

The effect is that the function header looks like a function that returns an `IEnumerable<T>` collection of something. The key phrase `yield return` is an implementation of the iterator pattern, and it supports treating things simply like an iterable collection even if they are not.

Listing 4.1 shows a simple usage of `yield return`. `GetEvens` is used just like a collection and the `yield return` code generates an `Enumerable` class shown using Reflector in Figure 4.2.

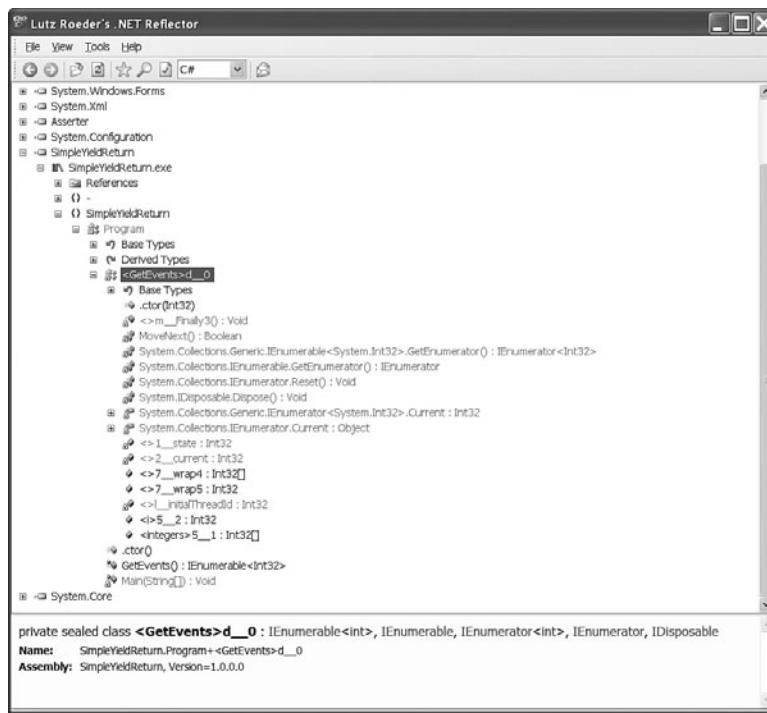


FIGURE 4.2 An `IEnumerable<T>` class is generated when you use `yield return` as shown here in Lutz Roeder's Reflector.

LISTING 4.1 A Simple Demonstration of `yield return`

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace SimpleYieldReturn
{
    class Program
    {
        static void Main(string[] args)
        {
            foreach (int i in GetEvens())
                Console.WriteLine(i);
        }

        static IEnumerable<int> GetEvens()
        {
            for (int i = 0; i < 10; i++)
                if (i % 2 == 0)
                    yield return i;
        }
    }
}

```

LISTING 4.1 Continued

```
{  
    static void Main(string[] args)  
    {  
        foreach(int i in GetEvens())  
            Console.WriteLine(i);  
        Console.ReadLine();  
    }  
  
    public static IEnumerable<int> GetEvens()  
    {  
        var integers = new[]{1, 2, 3, 4, 5, 6, 7, 8};  
        foreach(int i in integers)  
            if(i % 2 == 0)  
                yield return i;  
    }  
}
```

The `yield return` key phrase enumerator also implements `IDisposable` and a `try finally` block implicitly exists (and is code generated) around the use of the generated enumerable object.

The `yield return` phrase is governed by these basic rules:

- ▶ `Yield return` is followed by an expression where the expression has to be convertible to the type of the iterator.
- ▶ `Yield return` has to appear inside an iterator block; the iterator block can be in a method body, operator function, or accessor (property method).
- ▶ `Yield return` can't be used in unsafe blocks.
- ▶ Parameters to methods, operators, and accessors cannot be `ref` or `out`.
- ▶ `Yield return` statements cannot appear in a `catch` block or in a `try` block with one or more `catch` clauses.
- ▶ And, `yield` statements cannot appear in anonymous methods.

Using `yield return` and `yield break`

A function that has a `yield return` statement can be used for all intents and purposes, such as an `IEnumerable<T>` collection, that is, a collection of some object type `T`. This means you get the benefit of enumerability of any type of object without writing the typed collection class and plumbing for each type.

Listing 4.2 contains a `BinaryTree` class—a logical tree structure where each node contains zero, one, or two child nodes—that uses `yield return` to traverse the tree in order. In case you need a refresher, a binary tree starts with one root node. Then, nodes are added based on whether the value to be inserted is less than or greater than the root. For example, if you insert 5, 1, and 8, the tree will contain a root of 5, a left child of 1, and a right child 8. Insert 13 and 8 will have a right child of 13, and so on. To support comparing the data values of the generic `BinaryTree<T>` class, it is defined so that all types `T` have to be `IComparable`. `IComparable` ensures that you have a uniform way to examine nodes as you insert items in the list.

LISTING 4.2 An InOrder Traversal Binary Tree Using `yield return`

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Runtime.Serialization;

namespace BinaryTreeDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            BinaryTree<int> integers = new BinaryTree<int>();
            integers.AddRange(8, 5, 6, 7, 3, 4, 13, 21, 1, 17);

            // display conceptual tree
            integers.Print();
            Console.ReadLine();
            Console.Clear();

            // write in order
            foreach(var item in integers.Inorder)
                Console.WriteLine(item);
            Console.ReadLine();
        }
    }
}

//          8
//        / \
//      5   13
```

LISTING 4.2 Continued

```
//          / \     \
//          3   6     21
//          / \     \   /
//          1   4     7  17

public class Node<T> where T : IComparable<T>
{
    public T data;

    private Node<T> leftNode;
    public Node<T> LeftNode
    {
        get { return leftNode; }
        set { leftNode = value; }
    }

    private Node<T> rightNode;
    public Node<T> RightNode
    {
        get { return rightNode; }
        set { rightNode = value; }
    }

    /// <summary>
    /// Initializes a new instance of the Node class.
    /// </summary>
    /// <param name="data"></param>
    public Node(T data)
    {
        this.data = data;
    }

    public override string ToString()
    {
        return data.ToString();
    }
}

public class BinaryTree<T> where T : IComparable<T>
{
```

LISTING 4.2 Continued

```
private Node<T> root;
public Node<T> Root
{
    get { return root; }
    set { root = value; }
}

public void Add(T item)
{
    if( root == null )
    {
        root = new Node<T>(item);
        return;
    }
    Add(item, root);
}

private void Add(T item, Node<T> node)
{
    if(item.CompareTo(node.data) < 0)
    {
        if(node.LeftNode == null)
            node.LeftNode = new Node<T>(item);
        else
            Add(item, node.LeftNode);
    }
    else if( item.CompareTo(node.data) > 0)
    {
        if(node.RightNode == null)
            node.RightNode = new Node<T>(item);
        else
            Add(item, node.RightNode);
    }

    // silently discard it
}

public void AddRange(params T[] items)
{
    foreach(var item in items)
        Add(item);
}
```



LISTING 4.2 Continued

```
/// <summary>
/// error in displaying tree - 7 is overwritten 17!
/// </summary>
public void Print()
{
    Print(root, 0, Console.WindowWidth / 2);
}

public IEnumerable<T> Inorder
{
    get{ return GetInOrder(this.root); }
}

IEnumerable<T> GetInOrder(Node<T> node)
{
    if(node.LeftNode != null)
    {
        foreach(T item in GetInOrder(node.LeftNode))
        {
            yield return item;
        }
    }

    yield return node.data;

    if(node.RightNode != null)
    {
        foreach(T item in GetInOrder(node.RightNode))
        {
            yield return item;
        }
    }
}
private void Print(Node<T> item, int depth, int offset)
{
    if(item == null) return;
    Console.CursorLeft = offset;
    Console.CursorTop = depth;
    Console.Write(item.data);

    if(item.LeftNode != null)
```

LISTING 4.2 Continued

```
Print("/", depth + 1, offset - 1);
Print(item.LeftNode, depth + 2, offset - 3);

if(item.RightNode != null)
    Print("\\\", depth + 1, offset + 1);
    Print(item.RightNode, depth + 2, offset + 3);
}

private void Print(string s, int depth, int offset)
{
    Console.CursorLeft = offset;
    Console.CursorTop = depth;
    Console.Write(s);
}
}
```



The statement in `Main`, `integers.Print`, displays the logical version of the tree in the way that we conceptually think of it.

More important, `GetInOrder` checks to see if the current node's `LeftNode` is not `null`. While the `LeftNode` is not `null`, we walk to the farthest `LeftNode` using recursion and `yield return`. As soon as we run to the end of the `LeftNode` objects, we return the current node, which is the leftmost node, that node's parent, and the right node. The process of following left edges and displaying left, parent, right ensures that the items are returned in order no matter what order they appeared and were added to the binary tree.

One drawback to the recursive implementation is that each call to `GetInOrder` creates an instance of the generated `IEnumerable<T>` object, so very large trees would create a large number of these object instances.

If we reverse the position of the `LeftNode` checks and the `RightNode` checks, we would get the ordered items from greatest to smallest.

To see different logical views of the tree and verify that `GetInOrder` always returns the items from smallest to largest, try changing the order of the items in the `integers.AddRange` method.

Profiling Code

If you want to see how code like that in Listing 4.2 performs, including how many objects are created and how long various function calls take, profile the application by following these steps:

1. Select Developer, Performance Wizard.
2. Choose the application to profile and click Next.

3. Specify the Instrumentation profiling method in the Performance Wizard.
4. Click Finish.
5. In the Performance Explorer, select the profiling project and click the Launch with Profiling button.
6. Let the program run through to completion and double-click on the performance report in the Performance Explorer (see Figure 4.3).

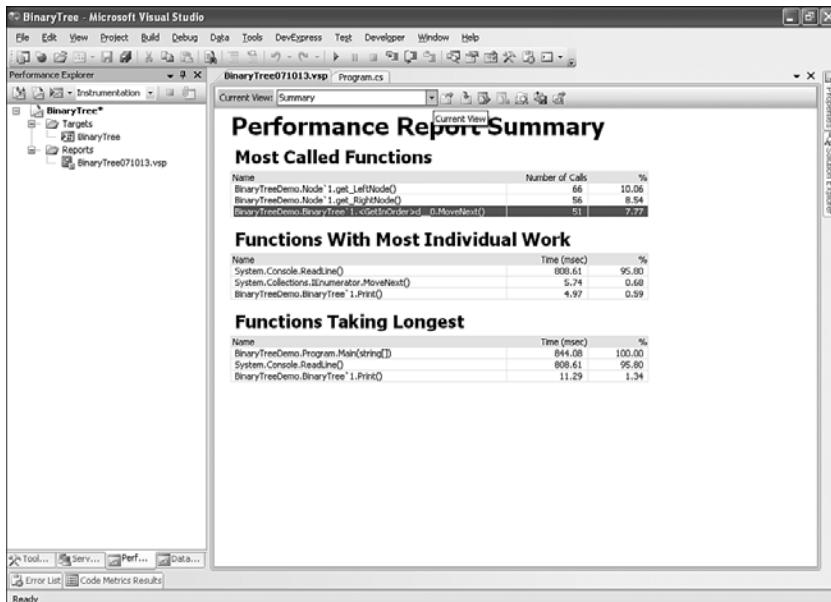


FIGURE 4.3 The view of the performance summary; select the type of view from the Current View drop-down list.

If you want to see how calls to the `IEnumerable<T>` generated class constructor were made, select Functions from the Current View. Instances of the code-generated enumerable collection should look something like `<GetInOrder>d__0..ctor(int32)`.

You can use the results from the profiling session to see exactly what your code is doing and how long things are taking.

NOTE

Profiling code is a great activity to perform in the latter part of your project when code is stabilized and nearing completion. Profiling early makes less sense because code that might be causing some bottlenecks might be removed before the end of the project. A good rule of thumb is correct and complete and then profile for performance.

Using `yield break`

If, for some reason, you want to terminate a `yield return` before the last item is reached, use the `yield break` phrase. Listing 4.3 shows the proper use of the `yield break` phrase. The code breaks when the depth of `GetInOrder` recurses more than four times, printing only the top four levels of the tree.

LISTING 4.3 Terminating a `yield return` iterator with `yield break`

```
int depth = 0;
IEnumerable<T> GetInOrder(Node<T> node)
{
    if(depth++ > 4) yield break;
    if(node.LeftNode != null)
    {
        foreach(T item in GetInOrder(node.LeftNode))
        {
            yield return item;
        }
    }

    yield return node.data;

    if(node.RightNode != null)
    {
        foreach(T item in GetInOrder(node.RightNode))
        {
            yield return item;
        }
    }
}
```



Summary

The phrases `yield return` and `yield break` were introduced in .NET 2.0. `yield return` creates a code-generated, typed enumeration, and `yield break` terminates the iteration before the last item. You can still use `yield return` and `yield break` (and use `yield` as a keyword) in .NET 3.5, but essentially this behavior is an evolutionary step on the way to LINQ queries. For the most part, where you want an `IEnumerable<T>` collection, you can shorten the code by simply using a LINQ query.

This page intentionally left blank

CHAPTER 5

Understanding Lambda Expressions and Closures

"In wisdom gathered over time I have found that every experience is a form of exploration."

—Ansel Adams

Hardware capabilities are way ahead of software capabilities, and customer demands are ahead of hardware and software. Holography, instant on and off computers, no moving parts, infinite storage, limitless speed, dependability, and 100% uptime are just a few features that would be great to have.

Businesses and people ask for and need evermore powerful software applications. For programmers, this means code has to do more. Creating code is a function of time and money. Programmers need to be able to use higher and higher levels of abstraction to write code that does more with less and that consumes less time and money. All of this implies code compression—fewer bits of code to do substantially more, hence Lambda Expressions.

Lambda Expressions are based on functional calculus—Lambda Calculus—from the 1930s, and Lambda Expressions exist in other languages. (Check www.wikipedia.org for a brief history of Lambda Calculus.)

Think of Lambda Expressions as brief inline functions whose concise nature is an ideal fit for LINQ. This chapter looks at the evolution path to Lambda Expressions, how to write and use Lambda Expressions, closures, and currying.

IN THIS CHAPTER

- ▶ Understanding the Evolution from Function Pointers to Lambda Expressions
- ▶ Writing Basic Lambda Expressions
- ▶ Dynamic Programming with Lambda Expressions
- ▶ Lambda Expressions and Closures
- ▶ Currying

Understanding the Evolution from Function Pointers to Lambda Expressions

How Product Evolution Aids in Understanding

In 2005, I was working as a contractor for Microsoft consulting. While waiting to meet my team, I was in the lobby of building 41 on the Redmond Washington campus.

Across from me was a guy about my age. Making small talk, I asked him what he was doing. He said he was one of the original 300 Microsoft employees, a C++ compiler writer, and had left in 1986. He was returning to work after roughly 20 years. (I didn't ask if he got bored or if his stock option money finally ran out.) I said, "You'd think Bill Gates would come down and welcome you back." In perfect deadpan, he said, "Bill is not a real people person."

On reflection, I now wonder how difficult it would be to work on a C++ compiler 20 years later—a lot must've changed in two decades. For me, it's always been easier to understand new language features when there are some common underpinnings and relationships to existing features. By knowing the progression or history of a feature, that feature seems much less like "magic." Lambda Expressions are part of an evolutionary progression, too.

In the 1970s Brian Kernighan and Dennis Ritchie brought us function pointers in C (see Listing 5.1). A function is referenced by an address, which is just a number, after all. Function pointers led to the notion of events—really just function pointers—and event handlers. Event handlers evolved into delegates and multicast delegates in .NET (see Listing 5.2). (Remember, fundamentally, we are still talking about function pointers under the hood but with nicer window treatments.) Delegates were compressed into anonymous delegates in .NET 2.0. Anonymous delegates lost some weight by eliminating the function name, return type, and parameter types (shown in Listing 5.3), and anonymous expressions have evolved into Lambda Expressions in Listing 5.4.

LISTING 5.1 A C++ Function Pointer Demo Where the Statement Beginning with `typedef` Defines the Function Pointer Named `FunctionPointer`

```
// FunctionPointer.cpp : main project file.
#include "stdafx.h"
using namespace System;
typedef void (*FunctionPointer)(System::String ^str);
void HelloWorld(System::String ^str)
{
    Console::WriteLine(str);
    Console::ReadLine();
}

int main(array<System::String ^> ^args)
{
    FunctionPointer fp = HelloWorld;
```

LISTING 5.1 Continued

```
fp(L"Hello World");
return 0;
}
```

LISTING 5.2 A C# Demo That Shows the Simpler Syntax of Function Pointers, Called Delegates in C#

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace CSharpFunctionPointer
{
    class Program
    {
        delegate void FunctionPointer(String str);

        static void Main(string[] args)
        {
            FunctionPointer fp = HelloWorld;
            fp("Hello World!");
        }

        static void HelloWorld(string str)
        {
            Console.WriteLine(str);
            Console.ReadLine();
        }
    }
}
```



LISTING 5.3 Identical Behavior to Listing 5.2, This Code Demonstrates an Anonymous Delegate

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace CSharpAnonymousDelegate
{
    class Program
```

LISTING 5.3 Continued

```
{  
    delegate void FunctionPointer(string str);  
  
    static void Main(string[] args)  
    {  
        FunctionPointer fp = delegate(string s)  
        {  
            Console.WriteLine(s);  
            Console.ReadLine();  
        };  
  
        fp("Hello World!");  
    }  
}
```

LISTING 5.4 A C# Example Demonstrating a Very Simple Lambda Expression Playing the Role of Delegate—`s => Console.WriteLine(s)`;

```
using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Text;  
  
namespace CSharpAnonymousDelegate  
{  
    class Program  
    {  
        delegate void FunctionPointer(string str);  
  
        static void Main(string[] args)  
        {  
            FunctionPointer fp =  
                s => Console.WriteLine(s);  
  
            fp("Hello World!");  
            Console.ReadLine();  
        }  
    }  
}
```

Using our weight-loss analogy, Lambda Expressions eliminated the `delegate` keyword, function header, parameter types, brackets, and `return` keyword (see Listing 5.4). This very

concise grammar for Lambda Expressions is not plain old esoterism (esoteric code for the sole purpose of terrorizing programmers). Lambda Expressions' concise nature makes it easy for these smallish functions to fit into new constructs that support LINQ.

A Lambda Expression is a concise delegate. The left side (of the =>) represents the arguments to the function and the right side is the function body. In Listing 5.4, the Lambda Expression is assigned to the delegate `FunctionPointer`, but .NET 3.5 defines anonymous delegates like `Action<T>` and `Func<T>`, which can be used instead of the more formal, longer delegate statement. This means that you don't have to define the delegate statement (as shown in Listing 5.4). Listing 5.5 shows the revision of Listing 5.4, replacing the `delegate` statement with the generic `Action<T>` delegate type.

LISTING 5.5 Assigning the Lambda Expression to a Predefined Generic Delegate

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace LambdaExpressionWithGenericAction
{
    class Program
    {
        static void Main(string[] args)
        {
            System.Action<string> fp =
                s => Console.WriteLine(s);

            fp("Hello World!");
            Console.ReadLine();
        }
    }
}
```



In Listing 5.5, `System.Action<T>` is used instead of a custom delegate, as demonstrated in Listing 5.4.

Writing Basic Lambda Expressions

The `System` namespace defines generic delegates `Action`, `Func`, and `Predicate`. `Action` performs an operation on the generic arguments. `Func` performs an operation on the argument or arguments and returns a value, and `Predicate<T>` is used to represent a set of criteria and determine if the argument matches the criteria. Lambda Expressions can be assigned to instances of these types or anonymous types. Or, Lambda Expressions can be used as literal expressions to methods that accept `Func`, `Action`, or `Predicate` arguments.

This section looks at examples of Lambda Expressions that match each of these generic delegate types, takes a side trip through automatic properties, and shows ways to use these capabilities within the newer features of .NET.

Automatic Properties

Automatic properties are not directly or only related to Lambda Expressions, but automatic properties are relevant to the subject of condensed code and letting the framework do more of the labor.

The basic idea behind automatic properties is that many times, programmers define properties that are just wrappers for fields and nothing else happens. Historically, the programmer writes the field declaration, and the property header, getter, and setter, returning the field or setting the field to the value keyword, respectively. Using properties instead of fields is a recommended practice, but if there are no additional behaviors, automatic properties allow the compiler to add the field, getter, and setter. All the programmer has to do is write the property header.

However, there is a caveat to using automatic properties. You will save some time not writing the whole property and field definition, but you cannot use the field variable in your code. To get around this, you just use the property. Listing 5.6 demonstrates an IronChef class that has two automatic properties: Name and Specialty.

LISTING 5.6 Automatic Properties and Named Type Initialization Save Time

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace AutomaticProperty
{
    class Program
    {
        static void Main(string[] args)
        {
            IronChef Batali = new IronChef{
                Name="Mario Batali", Specialty="Italian" };
            Console.WriteLine(Batali.Name);
            Console.ReadLine();
        }
    }

    public class IronChef
    {
        public string Name{ get; set; };
        public string Specialty{ get; set; }
    }
}
```

Listing 5.6 defines the class `IronChef` with the automatic properties `Name` and `Specialty`. Notice that you don't need to include the property method bodies. Also notice that you don't need a constructor. With named type initialization (see Chapter 2, "Using Compound Type Initialization") and automatic properties, simple classes can now be defined with many fewer lines of code and the compiler completes the definition (see the MSIL in Figure 5.1).

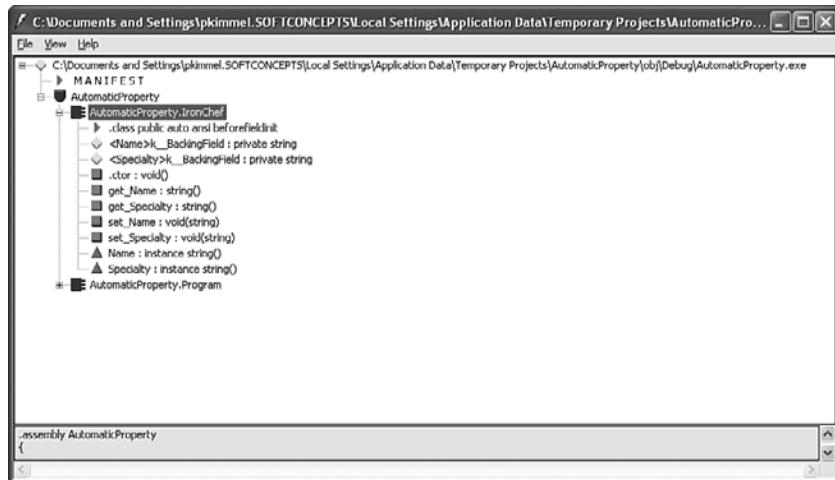


FIGURE 5.1 The snapshot from Intermediate Language Disassembler (ILDASM) shows that our `IronChef` class is provided with getters, setters, and backing fields by the compiler.

Automatic properties implicitly instruct the compiler to generate what is referred to as a backing field and these fields are annotated with the `CompilerGeneratedAttribute`.

Reading Lambda Expressions

Lambda Expressions are written with a left side, the `=>` symbol, and a right side, as in `(x,y) => x + y;`. The left side represents the input arguments and the right side is the expression to be evaluated. For example

```
(x,y) => x + y;
```

is read `x` and `y` goes to—or as I read it *gosinta*—the function `x + y`. Implicit in the expression is the return result of `x + y`. The input argument or arguments can be inferred and generated by the compiler, called implicit arguments, or expressed explicitly by you. The preceding Lambda Expression can also be rewritten as

```
(int x, int y) => x + y;
```

Listing 5.7 shows an example that uses explicit arguments, although doing so is probably just extra typing on your part. The explicit input argument statement is shown in bold.

LISTING 5.7 A Lambda Expression Demonstrating Explicit Argument Types

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ExplicitLambdaArguments
{
    class Program
    {
        static void Main(string[] args)
        {
            Func<int, int, int> add =
                (int x, int y) => x + y;
            Console.WriteLine(add(3, 4));
            Console.ReadLine();
        }
    }
}
```

In Listing 5.7, the generic `Func<T, T, TResult>` delegate type is used to represent the Lambda Expression referred to by the variable `add`.

Lambda Expressions Captured as Generic Actions

The generic `Action` delegate lets you capture behavior as an invokable object. Simply provide the parameter types for the `Action`, assign it a Lambda Expression, and you are up and running.

Listing 5.8 demonstrates using multiple parameters, including a `string` and `TextWriter`, which supports sending output to anything that is an instance of a `TextWriter`, including the `Console`. In Listing 5.9, the `Array.ForEach<T>` generic method is demonstrated, accepting an `Action<T>`, which supports iterating over an `Array<T>` of objects.

LISTING 5.8 Redirecting Output Using an `Action<T, T>` and a Lambda Expression

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Diagnostics;
using System.IO;
namespace LambdaExpressionAction
{
    class Program
```

LISTING 5.8 Continued

```
{
    static void Main(string[] args)
    {
        Action<string, TextWriter> print =
            (s, writer) => writer.WriteLine(s);
        print("Console", Console.Out);
        StreamWriter stream = File.AppendText("c:\\temp\\text.txt");
        stream.AutoFlush = true;
        print("File", stream);
        Console.ReadLine();
    }
}
```

LISTING 5.9 The Output from the Action `michiganSalesTax` Is Fed into the Action `print`; the Results Are Printed Using the `Array.ForEach` Generic Method

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ForEachWithActionLambda
{
    class Program
    {
        static void Main(string[] args)
        {
            Action<double> print =
                amount => Console.WriteLine("{0:c}", amount);
            Action<double> michiganSalesTax =
                amount => print(amount *= 1.06);

            var amounts = new double[]{
                10.36, 12.00, 134};

            Array.ForEach<double>(amounts, michiganSalesTax);
            Console.ReadLine();
        }
    }
}
```



In Listing 5.9, `print` uses a Lambda Expression to send a double to the `Console` formatted as a currency value. The Lambda Expression represented by `michiganSalesTax` adds 6% to the input double and sends the result to `print`. Near the end of Listing 5.9, the `Array.ForEach` generic method is used to iterate over each of the amounts and call `michiganSalesTax`. (It would sure be nice if those tax numbers went southward once in a while.)

Searching Strings

If you piece some of the elements from prior chapters together, it is easy to see how the mosaic of progression has evolved to LINQ queries. So far, you have learned about extension methods, anonymous types, and, now, Lambda Expressions (among other things). For instance, you know that you can use array initializer syntax and assign the results to an anonymous type, letting the compiler generate the details. Further, you know that arrays implement `IEnumerable` and you can add behaviors via extension methods. From Chapter 3, “Defining Extension and Partial Methods,” you know that `IEnumerable` has been extended to include generic methods such as `Where<T>` that accept predicates—much like `WHERE` in Structured Query Language (SQL)—and that these predicates can be expressed as `Func<TSource, TResult>` return types or literal Lambda Expressions.

Combining all of this knowledge, you can use Lambda Expressions as arguments to `Where<T>` and define short-and-sweet searches, for example, with a very few lines of code. Listing 5.10 demonstrates an anonymous type initialized by an array initializer, use of the `Where<T>` with a Lambda Expression predicate to filter the strings (recipes) that have Chicken, and the `ForEach` with a second Lambda Expression that writes each found item. This style of programming (in Listing 5.10) is referred to as functional programming.

LISTING 5.10 Combining Anonymous Types, Array Initializers, Lambda Expressions, Actions, and Extension Methods to Search and Display Strings That Contain a Keyword

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Searching
{
    class Program
    {
        static void Main(string[] args)
        {
            var recipes = new[]{
                "Crepes Florentine", "Shrimp Che Paul",
                "Beef Burgundy", "Rack of Lamb",
                "Tacos", "Rosemary Roasted Chicken",
                "Juevos Rancheros", "Buffalo Chicken Nachos"};
            var results = recipes.Where(

```

LISTING 5.10 Continued

```

    s=>s.Contains("Chicken"));

    Array.ForEach<string>(results.ToArray<string>(),
        s => Console.WriteLine(s));
    Console.ReadLine();
}

}
}
}

```

Listing 5.11 provides a variation on the search behavior by finding odd numbers in a short Fibonacci sequence. This version uses a generic `List<int>`, a Lambda Expression assigned to a `Predicate<int>` initialized by a Lambda Expression, and the `List<T>.FindAll` method—that accepts a predicate. The results are displayed by the `ObjectDumper`. `ObjectDumper` was implemented as demo code for Visual Basic (VB) but might or might not ship with .NET 3.5. (The `ObjectDumper` is installed with sample code at C:\Program Files\Microsoft Visual Studio 9.0\Samples\1033\CSharpSamples.zip.)

LISTING 5.11 A Variation on Searching Behavior Using `List<T>`'s `FindAll` and a `Predicate` Initialized with a Lambda Expression

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace FindingNumbers
{
    class Program
    {
        static void Main(string[] args)
        {
            List<int> fiboList = new List<int>();
            fiboList.AddRange(
                new[]{1,1,2,3,5,8,13,21,34,55});

            Predicate<int> match = n => n % 2 == 1;
            var odds = fiboList.FindAll(match);

            ObjectDumper.Write(odds);
            Console.ReadLine();
        }
    }
}

```



Lambda Expressions Captured as Generic Predicates

The `Predicate<T>` generic delegate accepts an argument indicated by the parameter `T` and returns a Boolean. `Predicate<T>` is used in functions like `Array.Find` and `Array.FindAll`. `Predicate<T>` can be initialized with a regular function, an anonymous delegate, or a Lambda Expression.

In the example in Listing 5.12 (based on a Windows Form project named `FindSquares`), the code draws a hundred random rectangles and then uses a Lambda Expression assigned to `Predicate<Rectangle>` and only draws the rectangles whose area is less than or equal to a target range. Originally, a hundred rectangles are created and a `Timer` paints some number of rectangles every two and a half seconds. The Lambda Expression is on the following line:

```
Predicate<Rectangle> area = rect => (rect.Width * rect.Height) <= random.Next(200)
*
random.Next(200));
```

LISTING 5.12 Demonstrating `Predicate<T>` Initialized with a Lambda Expression

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace FindSquares
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        Rectangle[] rects = new Rectangle[100];
        private void Form1_Load(object sender, EventArgs e)
        {
            Random random = new Random(DateTime.Now.Millisecond);
            int x, y, width, height;

            for(int i=0; i<100; i++)
            {
                x = random.Next(this.ClientRectangle.Width / 2);
```

LISTING 5.12 Continued

```
y = random.Next(this.ClientRectangle.Height / 2);
width = random.Next(200);
height = random.Next(200);
rects[i] = new Rectangle(x, y, width, height);
}

}

private void Form1_Paint(object sender, PaintEventArgs e)
{
    Random random = new Random(DateTime.Now.Millisecond);
    Predicate<Rectangle> area = rect =>
        (rect.Width * rect.Height) <= (random.Next(200) *
        random.Next(200));

    Rectangle[] matches = Array.FindAll(rects, area);
    e.Graphics.Clear(this.BackColor);

    foreach(var rect in matches)
        e.Graphics.DrawRectangle(Pens.Red, rect);
    e.Graphics.DrawString("Found: " + matches.Length.ToString(),
        this.Font, Brushes.Black, 10, ClientRectangle.Height - 40);
}

private void timer1_Tick(object sender, EventArgs e)
{
    Invalidate();
}
}
```



Binding Control Events to Lambda Expressions

It is worth remembering that Lambda Expressions are just very short functions, basically inline functions. Therefore, you can assign a Lambda Expression anywhere you would use a function or anonymous delegate. Remember, as is shown in Listing 5.13, the left side of the `<=` operator is the inputs and the right side represents the method body. Therefore, to match a typical `EventHandler`, you need the left side to have an object and `EventArgs` inputs and the right side to do the typical kinds of things you would do with more verbose event handlers.

LISTING 5.13 Using Lambda Expressions for Everyday Event Handlers

```
using System;
using System.Collections.Generic;
```

LISTING 5.13 Continued

```
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace LambdaEventHandlers
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
            button1.Click += (s, e) => MessageBox.Show("Click!");
        }
    }
}
```

Dynamic Programming with Lambda Expressions

You can begin exploring how Lambda Expressions support LINQ queries by seeing how Lambda Expressions are used in extension methods such as `Select<T>`, `Where<T>`, or `OrderBy<T>`. (Of course, more extension methods than these exist, as more capabilities of LINQ queries are built on top of these extension methods, but you get the general idea.)

This section explores `Select<T>`, `Where<T>`, and `OrderBy<T>`. LINQ fundamentals begin in Chapter 6, “Using Standard Query Operators,” and LINQ keywords are explored in that chapter; just remember that underneath those capabilities are extension methods and Lambda Expressions.

Using `Select<T>` with Lambda Expressions

Suppose you have an array of integers. You could simulate the `SELECT *` behavior of SQL queries by initializing the `Select` extension method with `n => n`. `n => n` means that `n` is the input and you want to return `n`. Listing 5.14 shows a very simple `Select` usage. `Select` returns an `IEnumerable<T>` instance, where `T` is the type of object returned by the Lambda Expression.

LISTING 5.14 A Simple Example That Demonstrates the Extension Method `Select<T>`

```
using System;
using System.Collections.Generic;
```

LISTING 5.14 Continued

```
using System.Linq;
using System.Text;

namespace SelectLambda
{
    class Program
    {
        static void Main(string[] args)
        {
            var numbers = new int[]{1,2,3,4,5,6};
            foreach(var result in numbers.Select(n => n))
                Console.WriteLine(result);
            Console.ReadLine();
        }
    }
}
```

In the example, the code doesn't do much more than if you just used the numbers themselves in the `foreach` statement. However, you can't—with just numbers—project a new type. With the Lambda Expression, you can project `n` to a new type with named initialization. For example, changing the Lambda Expression to `n=> Number=n` causes the extension method to project (or create) a new anonymous type with a field `Number`. By adding a second named type, you could add an additional field that indicates whether the number `n` is odd or even. Listing 5.15 shows the revision.

LISTING 5.15 Projecting a New Type Using Named Compound Initialization Resulting in a New Anonymous Class with Two Fields `Number` and `IsEven`

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace SelectLambda
{
    class Program
    {
        static void Main(string[] args)
        {
            var numbers = new int[]{1,2,3,4,5,6};
            foreach(var result in numbers.Select(n => new {Number=n, IsEven=n%2==0}))
                Console.WriteLine(result);
            Console.ReadLine();
        }
    }
}
```



LISTING 5.15 Continued

```

    }
}
}
}
```

The new anonymous type is shown in the snapshot of ILDASM (see Figure 5.2), clearly showing the projected anonymous type and the two fields Number and IsEven.



FIGURE 5.2 The snapshot from ILDASM shows that by using compound type initialization and named parameters, in effect, a new class with a Number and IsEven properties is created.

Using Where<T> with Lambda Expressions

The `Where<T>` extension method is employed in scenarios where you would use conditional logic to filter the elements returned in a resultset. Like `Select`, `Where` returns an `IEnumerable<T>`, where `T` is defined by the type of the result from the Lambda Expression. Listing 5.16 provides an example of `Where` that looks for words that contain capital `D` and lowercase `e` among the array of strings.

LISTING 5.16 Using `Where<T>` to Search for Words with `D` and `e`

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace WhereLambda
{
    class Program
```

LISTING 5.16 Continued

```
{  
    static void Main(string[] args)  
    {  
        var words = new string[]{"Drop", "Dead", "Fred"};  
        IEnumerable<string> hasDAndE =  
            words.Where(s => s.Contains('D') && s.Contains('e'));  
  
        foreach(string s in hasDAndE)  
            Console.WriteLine(s);  
        Console.ReadLine();  
    }  
}
```

Using `OrderBy<T>` with Lambda Expressions

`OrderBy<T>` is the extension method that supports sorting. `OrderBy` accepts a `Func` generic delegate. The `Func` argument can be expressed with a literal Lambda Expression. Listing 5.17 demonstrates `OrderBy`, which is used to sort the array of numbers in ascending order. (To sort in descending order, call the `OrderByDescending` method.)

LISTING 5.17 Sorting Is as Easy as Calling the `OrderBy` or `OrderByDescending` Method

```
using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Text;  
  
namespace OrderByLambda  
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            var numbers = new int[]  
                {1, 3, 5, 7, 9, 2, 4, 6, 8, 10 };  
            IEnumerable<int> ordered = numbers.OrderBy(n=>n);  
            foreach(var num in ordered)  
                Console.WriteLine(num);  
            Console.ReadLine();  
        }  
    }  
}
```



Compiling Lambda Expressions as Code or Data

Lambda Expressions are either compiled as code or data. When a Lambda Expression is assigned to a variable, field, or delegate, the compiler emits executable IL. For example, `num => num % 2 == 0;` emits IL (see Listing 5.18) that is equivalent to a function that accepts an integer, performs division, and compares the remainder with 0. (The Lambda Expression is defined in a function named `Test`, hence the emitted `<Test>b_1'(int32 num)` generated name.)

LISTING 5.18 The Lambda Expression `num => num % 2 == 0` Emits the MSIL Listed Here

```
.method private hidebysig static bool  '<Test>b_1'(int32 num) cil managed
{
    .custom instance void [mscorlib]System.Runtime.CompilerServices.
    CompilerGeneratedAttribute::ctor()  = ( 01 00 00 00 )
    // Code size      7 (0x7)
    .maxstack  8
    IL_0000:  ldarg.0
    IL_0001:  ldc.i4.2
    IL_0002:  rem
    IL_0003:  ldc.i4.0
    IL_0004:  ceq
    IL_0006:  ret
} // end of method Program:'<Test>b_1'
```

If a Lambda Expression is assigned to a variable, field, or parameter whose type is `System.Linq.Expressions.Expression<TDelegate>`, where `TDelegate` is a delegate, code that represents an expression tree is emitted (see Listing 5.19). Expression trees are used for LINQ features like LINQ for Data where LINQ queries are converted to T-SQL.

LISTING 5.19 A Lambda Expression (in a Function `Test2`) Assigned to an Instance of `Expression<TDelegate>` Emits the IL Shown

```
.method private hidebysig static void  Test2() cil managed
{
    // Code size      93 (0x5d)
    .maxstack  4
    .locals init ([0] class [System.Core]System.Linq.Expressions.ParameterExpression
    CS$0$0000,
                 [1] class [System.Core]System.Linq.Expressions.ParameterExpression[])
    CS$0$0001)
    IL_0000:  ldtoken    [mscorlib]System.Int32
    IL_0005:  call        class [mscorlib]System.Type [mscorlib]System.Type::
    GetTypeFromHandle(valuetype [mscorlib]System.RuntimeTypeHandle)
    IL_000a:  ldstr      "num"
```

LISTING 5.19 Continued

```
IL_000f: call      class [System.Core]System.Linq.Expressions.  
➥ParameterExpression [System.Core]System.Linq.Expressions.  
➥Expression::Parameter(class [mscorlib]System.Type,  
  
string)  
IL_0014: stloc.0  
IL_0015: ldloc.0  
IL_0016: ldc.i4.2  
IL_0017: box       [mscorlib]System.Int32  
IL_001c: ldtoken   [mscorlib]System.Int32  
IL_0021: call      class [mscorlib]System.Type [mscorlib]System.Type::  
➥GetTypeFromHandle(valuetype [mscorlib]System.RuntimeTypeHandle)  
IL_0026: call      class [System.Core]System.Linq.Expressions.  
➥ConstantExpression [System.Core]System.Linq.Expressions.  
➥Expression::Constant(object,  
  
class [mscorlib]System.Type)  
IL_002b: call      class [System.Core]System.Linq.Expressions.BinaryExpression  
➥[System.Core]System.Linq.Expressions.Expression::Modulo  
➥(class [System.Core]System.Linq.Expressions.Expression,  
  
class [System.Core]System.Linq.Expressions.Expression)  
IL_0030: ldc.i4.0  
IL_0031: box       [mscorlib]System.Int32  
IL_0036: ldtoken   [mscorlib]System.Int32  
IL_003b: call      class [mscorlib]System.Type  
➥[mscorlib]System.Type::GetTypeFromHandle(valuetype  
➥[mscorlib]System.RuntimeTypeHandle)  
IL_0040: call      class [System.Core]System.Linq.Expressions.  
➥ConstantExpression [System.Core]System.Linq.Expressions.  
➥Expression::Constant(object,  
  
class [mscorlib]System.Type)  
IL_0045: call      class [System.Core]System.Linq.Expressions.  
➥BinaryExpression [System.Core]System.Linq.Expressions.Expression::  
➥Equal(class [System.Core]System.Linq.Expressions.Expression,  
  
class [System.Core]System.Linq.Expressions.Expression)  
IL_004a: ldc.i4.1  
IL_004b: newarr    [System.Core]System.Linq.Expressions.ParameterExpression  
IL_0050: stloc.1  
IL_0051: ldloc.1  
IL_0052: ldc.i4.0  
IL_0053: ldloc.0  
IL_0054: stelem.ref  
IL_0055: ldloc.1
```



LISTING 5.19 Continued

```
IL_0056: call      class [System.Core]System.Linq.Expressions.  
➥Expression`1<!!0> [System.Core]System.Linq.Expressions.Expression::  
➥Lambda<class [System.Core]System.Func`2<int32,bool>>  
➥(class [System.Core]System.Linq.Expressions.Expression,  
  
class [System.Core]System.Linq.Expressions.ParameterExpression[])  
IL_005b: pop  
IL_005c: ret  
} // end of method Program::Test2
```

The expression tree is an in-memory representation of the Lambda Expression. The expression can be compiled and its underlying Lambda Expression can be invoked just like any other Lambda Expression, but, more important, Lambda Expressions can be passed around and transformed by application programming interfaces (APIs) in new ways—for example, LINQ to SQL. The code in Listing 5.20 shows the code used to generate the MSIL in Listings 5.18 and 5.19 and Test2 includes code to explore the expression tree generated.

LISTING 5.20 The Code Used to Generate the MSIL in Listings 5.18 and 5.19 and Additional Code to Explore Expression Trees

```
using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Text;  
using System.Linq.Expressions;  
  
namespace EmitLambda  
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            Test();  
            Test2();  
        }  
  
        static void Test2()  
        {  
            Expression<Func<int, bool>> exp =  
                num => num % 2 == 0;  
            // The MSIL is generated from the statement above  
            Console.WriteLine("Body: {0}", exp.Body.ToString());  
            Console.WriteLine("Node Type: {0}", exp.NodeType);  
            Console.WriteLine("Type: {0}", exp.Type);  
            Func<int, bool> lambda = exp.Compile();
```

LISTING 5.20 Continued

```
Console.WriteLine(lambda(2));
Console.ReadLine();
}

static void Test()
{
    Func<int, bool> lambda = num => num % 2 == 0;
}
}
```

The code in `Test2` after the `Expression` definition demonstrates how you can explore the method body, the kind of expression, and argument types. The output from `Test2` is shown in Figure 5.3.



FIGURE 5.3 The output from `Test2` in Listing 5.20 illustrates how APIs can explore expressions to figure out what the expression represents and how to convert it to another form, like SQL.

Lambda Expressions and Closures

When you declare a variable local to a function, that variable lives in the stack memory space of the declaring scope. So, for example, when the function returns, the local variables are cleaned up with the stack memory for that function. All of this means that if you use a local variable in a Lambda Expression, that local variable would be undefined when the function's stack is cleaned up. As a precaution, in the event a Lambda Expression is to be returned from a function, and that Lambda Expression depends on a local variable, the compiler creates a `Closure` (or wrapper class).

A closure is a generated class that includes a field for each local variable used and the Lambda Expression. The value of the local variable(s) is copied to the generated fields to effectively extend the lifetime of the local variables.

If you are familiar with the behavior of stack memory, this explanation makes sense to you. If unfamiliar, don't worry. The compiler takes care of creating the closure and all of

the plumbing is transparent. Listing 5.21 shows a Lambda Expression that uses a local variable and Figure 5.4 shows the MSIL and the generated closure.

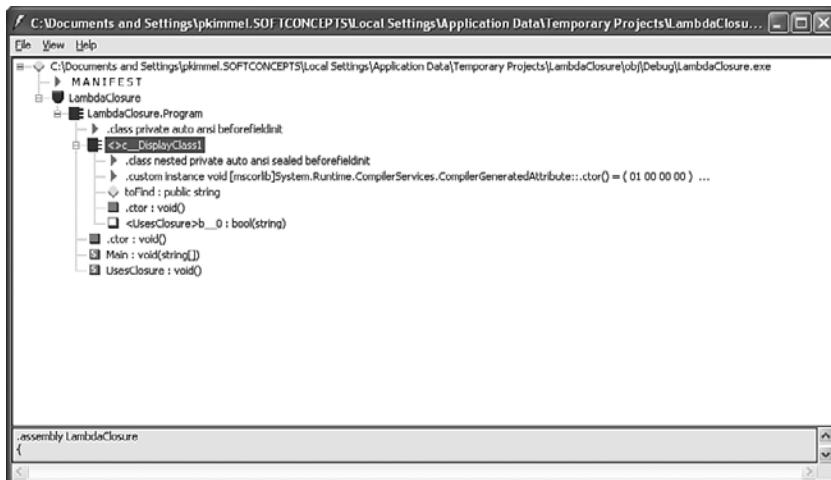


FIGURE 5.4 The closure class generated by the compiler is highlighted in the MSIL snapshot; without the local variable `toFind` (or any local), the closure is not generated.

LISTING 5.21 Using the Local Variable `toFind` Signals the Compiler to Generate a Closure, or Wrapper Class, so `toFind` Can't Become Undefined

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace LambdaClosure
{
    class Program
    {
        static void Main(string[] args)
        {
            UsesClosure();
        }

        static void UsesClosure()
        {
            string toFind = "ed";
            var words = new string[]{
                "ended", "friend", "closed", "potato"};

            var matches = words.Select(s => s.Contains(toFind));
        }
    }
}
```

LISTING 5.21 Continued

```

foreach(var str in matches)
    Console.WriteLine(str);
Console.ReadLine();
}
}
}

```

Currying

The most fun I had describing currying was to a group at the Ann Arbor Day of .NET at Washtenaw Community College in Michigan: “Currying was invented in the 1930s by the mathematician Carlos Santana.” Wikipedia can tell you about the science and math of currying and who is attributed with its invention. (You are encouraged to read the under-scoring explanation of currying.) Here, you’ll get a less-scientific explanation.

Currying is a use of Lambda Expressions in which a multistep expression is broken into several Lambda Expressions. Each Lambda Expression is a feeder to the next expression until the last expression provides the solution. This is sort of like a Turing machine. (“Turing Machines” were described by Alan Turing in the 1940s. Turing machines are basically simple computers that can solve any problem theoretically one step at a time, one piece at a time.) Refer to Listing 5.22 for an example.

LISTING 5.22 The First Half Is a Multiargument Lambda Expression and the Second Half Shows the Same Behavior Using Currying

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace LambdaCurry
{
    class Program
    {
        static void Main(string[] args)
        {
            Func<int, int, int> longLambda = (x,y) => x + y;
            Console.WriteLine(longLambda(3,4));
            Console.ReadLine();

            // currying
            Func<int, Func<int,int>> curry1 = x => y => x + y;
            Console.WriteLine(curry1(3)(4));
        }
    }
}

```

LISTING 5.22 Continued

```
        Console.ReadLine();
    }
}
}
```

In the first half, `longLambda` accepts two arguments and returns the sum. After the comment `// currying`, the same behavior is represented by a Lambda Expression that employs currying. The Lambda Expression `curry1` accepts a single `int` argument and returns a second Lambda `y => x + y`. The result is that you can call `curry1` with the first operand. This first call returns the second Lambda Expression, which you can immediately invoke—using the function call notation (`arg`) to invoke that Lambda Expression.

Lambda Expressions that employ currying are a little hard to write, and it is even harder to figure out what to do with currying Lambdas. It might be useful to employ currying Lambdas with code generators by chunking up long problems into singular Lambda Expressions that are daisy-chained together. It will be interesting to see what the programming community does with currying.

The important thing to remember is that for each step in a multiargument Lambda Expression, simply chain another `arg => expression` in the sequence; when invoking the curried expression, chain the function call operator with the input arguments (as demonstrated in Listing 5.22).

Summary

Lambda Expressions are fundamentally shorthand notation for anonymous delegates. They are inline functions. Lambda Expressions return instances of anonymous delegates, such as `Func<T>`, `Predicate<T>`, and `Action<T>`. The delegates are useful in extension methods that use functions to perform operations on arrays and other enumerable types.

Generic delegates, extension methods, and Lambda Expressions are an evolutionary step to LINQ. LINQ is a more natural like query language for .NET. LINQ queries use elements that look similar to query languages like SQL and these LINQ keywords are mapped to extension methods. The arguments to LINQ keywords are mapped to inputs to these extension methods, and the inputs are converted to the required type, usually a generic delegate.

This chapter also included discussions on projections, currying, and closures. Hopefully, these concepts will help you explore Lambda Expressions with your peers and find new and creative ways to use them.

CHAPTER 6

Using Standard Query Operators

“Most people have the will to win; few have the will to prepare to win.”

—Bobby Knight

There are a lot of keywords that make up the Language INtegrated Query (LINQ) grammar. Many of these keywords are similar to the Structured Query Language (SQL) and some elements are different. In addition to keywords, there are capabilities that are represented both by keywords and extension methods and capabilities only available by blending extension methods with LINQ queries.

This chapter explores general query syntax, query operators, and extension methods, including filtering, quantifiers, partitioning, generation, conversions, concatenation, and sequence-equality testing. Refer to Chapter 2, “Using Compound Type Initialization,” for conversion operators, Chapter 7, “Sorting and Grouping Queries,” for information on sorting and grouping, Chapter 8, “Using Aggregate Operations,” for aggregate operators, Chapter 9, “Performing Set Operations,” for set operations, Chapter 10, “Mastering Select and SelectMany,” for projects with `select`, and Chapter 11, “Joining Query Results,” for detailed information on joins.

Understanding How LINQ Is Implemented

LINQ is an extension to .NET. The grammar is new and adds elements to languages like C# (and VB .NET). However, LINQ did not crop up like a weed in a suburban

IN THIS CHAPTER

- ▶ Understanding How LINQ Is Implemented
- ▶ Constructing a LINQ Query
- ▶ Filtering Information
- ▶ Using Quantifiers
- ▶ Partitioning with Skip and Take
- ▶ Using Generation Operations
- ▶ Equality Testing
- ▶ Obtaining Specific Elements from a Sequence
- ▶ Appending Sequences with Concat

garden. LINQ queries are emitted as calls to extension methods and generic delegates and these are integral parts of the .NET Framework.

Chapters 1 through 5 methodically demonstrate how anonymous types, compound initialization, extension methods, generics, projections, Lambda Expressions, and generic delegates all play a role in supporting LINQ.

Constructing a LINQ Query

LINQ queries start with the keyword `from`. Several people have asked why we don't begin queries with `select`—as it is, `select` comes at the end. The answer ostensibly is that because IntelliSense unfolds as you type the LINQ queries, the `from` clause has to be first.

Consider a projection. In the `select` clause, you can define new types and initialize named properties of those new types from elements of the source type. If the `select` were first, it would be difficult, if not impossible, to use features like named initialization to define the projections because the type you are projecting from would be as yet unknown. It's a reasonable argument.

After you get past `from-first` and `select-last`, basic LINQ queries are pretty straightforward. Between the `from` clause and the `select` clause, you can find joins, wheres, orderbys, and intos.

This chapter provides examples demonstrating query operators. Several of the larger, more complicated LINQ constructs, such as joins and groups, are covered in the chapters of Part II.

Filtering Information

There are two kinds of filters: the traditional filter with the `where` clause, which works similarly to SQL queries, and the new filter `OfType`. `OfType` filters on a value's ability to be cast to a specific type. Listing 6.1 contains a query that filters numbers that have a magnitude (absolute value) greater than 5, and Listing 6.2 contains a query that returns elements from an array that can be converted to an integer. (Figure 6.1 shows how the sequence is converted in Listing 6.1.)

```
-1, -32, 3, 5, -8, 13, 7, -41
```

Absolute value greater than 5

```
-32, -8, 13, 7, -41
```

FIGURE 6.1 The sequence of positive and negative integers is reduced to just those with an absolute value greater than five.

LISTING 6.1 Return Elements from an Array That Has an Absolute Value Greater Than 5

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace FilteringData
{
    class Program
    {
        static void Main(string[] args)
        {
            var numbers = new int[] { -1, -32, 3, 5, -8, 13, 7, -41 };
            var magnitude = from n in numbers where Math.Abs(n) > 5 select n;
            foreach (var m in magnitude)
                Console.WriteLine(m);
            Console.ReadLine();
        }
    }
}
```

LISTING 6.2 Using the Extension Method `OfType`; There Is No LINQ Keyword for `OfType`

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace OfTypeFilterDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            var randomData = new object[] { 1, "two", 3, "four", 5.5, "six", 7M };

            var canBeInt = from r in randomData.OfType<int>() select r;
            foreach (var i in canBeInt)
                Console.WriteLine(i);
            Console.ReadLine();
        }
    }
}
```

The code in Listing 6.2 returns the values 1 and 3 (see Figure 6.2). Although you can convert 7M (a decimal number) to an integer with the Convert class and ToInt32 method, OfType does not perform that conversion. You can learn more about conversion operators in Chapter 2.

```
1, "two", 3, "four", 5.5, "six", 7M
```

that can be converted to an integer by OfType<int>

```
1, 3
```

FIGURE 6.2 OfType can convert types in a sequence but not as thoroughly as the Convert class.

Using Quantifiers

The quantifiers All, Any, and Contains are extension methods that return a Boolean indicating whether some or all of the elements in a collection satisfy the argument condition of the quantifier. All determines if every member satisfies the condition. Any returns a Boolean indicating whether any one or more members satisfy a condition, and Contains looks for a single element matching the condition.

Listing 6.3 demonstrates the quantifiers All and Any. The first check uses Any and a Lambda Expression to determine whether the species of any “family member” is feline, and the second check tests to determine if every family member is a dog (Species == “Canine”).

LISTING 6.3 Demonstrating Quantifiers Any and All

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace QuantifierDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            var household = new List<FamilyMember>{
                new FamilyMember{Name="Paul Kimmel", Species="Lunkhead in Chief",
                    Gender="Male"},
```

LISTING 6.3 Continued

```

new FamilyMember{Name="Dena Swanson", Species="Boss", Gender="Female"},  

new FamilyMember{Name="Alex Kimmel", Species="Princess", Gender="Female"},  

new FamilyMember{Name="Noah Kimmel", Species="Annoying Brother",  

    Gender="Male"},  

new FamilyMember{Name="Joe Swanson", Species="Homosapien", Gender="Male"},  

new FamilyMember{Name="Ruby", Species="Canine", Gender="Female"},  

new FamilyMember{Name="Leda", Species="Canine", Gender="Female"},  

new FamilyMember{Name="Big Mama", Species="Feline", Gender="Female"}  

};  
  

bool anyCats = household.Any(m=>m.Species == "Feline");  

Console.WriteLine("Do you have cats? {0}", anyCats);  

bool allDogs = household.All(m=>m.Species == "Canine");  

Console.WriteLine("All gone to the dogs? {0}", allDogs);  

Console.ReadLine();  

}  

}  
  

public class FamilyMember  

{  

    public string Name{get; set; }  

    public string Species{ get; set; }  

    public string Gender{ get; set; }  
  

    public override string ToString()  

    {  

        return string.Format("Name={0}, Species={1}, Gender={2}",  

            Name, Species, Gender);  

    }
}

```

Notice that the class `FamilyMember` uses the automatic properties feature of .NET. Automatic properties are properties introduced with the normal property syntax—everything except the getter and setter methods. Use this technique for properties that do nothing more than return the underlying field or set the value of the underlying field. (Refer to Chapter 5, “Understanding Lambda Expressions and Closures,” for more on automatic properties.)

The `Contains` extension method accepts an instance of the type contained in the collection and tests to determine if the collection contains that particular member. You can pass a second argument that implements `IEqualityComparer`. The contract for `IEqualityComparer`s is that a class that implements `IEqualityComparer` must provide an implementation for `Equals` and `GetHashCode`.

Partitioning with Skip and Take

`Skip`, `SkipWhile`, `Take`, and `TakeWhile` are used to partition collections into two parts and then return one of the parts. These partition features are only implemented as extension methods. `Skip` jumps to the indicated argument position in the sequence and returns everything after that. `SkipWhile` accepts a predicate and instead of using a position, it skips until the predicate evaluates to `false`. `Take` grabs all of the elements up to the specified position in the index, and `TakeWhile` grabs all of the items in a sequence as long as the predicate evaluates to `true`.

Listing 6.4 demonstrates `Skip` and `Take`. `Take(5)` returns a sequence containing the first five elements, the numbers 1 through 5, and `Skip(3)` returns the sequence containing the numbers 4 through 10. `TakeWhile` and `SkipWhile` substitute a test condition instead of using a cardinal value. For example, `ints.TakeWhile(n=>n<=5)` yields the same result as `Take(5)`. (See Figure 6.3 for the output visually depicted.)

1, 2, 3, 4, 5, 6, 7, 8, 9, 10

Take 5 yields the sequence

1, 2, 3, 4, 5

Skip 3 yields the sequence

4, 5, 6, 7, 8, 9, 10

FIGURE 6.3 `Take(5)` and `Skip(3)` yield the sequences as depicted in this figure.

LISTING 6.4 Partitioning with Skip and Take

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
```

```
namespace SkipDemo
{
```

LISTING 6.4 Continued

```
class Program
{
    static void Main(string[] args)
    {
        var ints = new int[]{1,2,3,4,5,6,7,8,9,10};
        var result1 = ints.Take(5);
        var result2 = ints.Skip(3);
        Array.ForEach(result1.ToArray(), n=> Console.WriteLine(n));
        Console.ReadLine();
        Array.ForEach(result2.ToArray(), n=> Console.WriteLine(n));
        Console.ReadLine();
    }
}
```

Using Generation Operations

The generation operations are `DefaultIfEmpty`, `Empty`, `Range`, and `Repeat`. These are implemented as extension methods. Generation operations are useful for creating new sequences of values. For example, `Empty` creates an empty collection of a given type. The brief descriptions and examples (or references to examples already provided) follow.

DefaultIfEmpty

The following brief fragment purposely creates an empty array to demonstrate `DefaultIfEmpty`. In the fragment, `DefaultIfEmpty` creates a one-element collection with the default value of `\0` for the contained type—a char.

```
var empties = Enumerable.DefaultIfEmpty<char>(Enumerable.Empty<char>());
Console.WriteLine("Count: {0}", empties.Count());
Console.ReadLine();
```

For another `DefaultIfEmpty` example, see the section on left joins in Chapter 11.

Empty

As introduced in the previous subsection (and also demonstrated in Chapter 8), `Empty` creates an empty collection of the type expressed by the parameter. For example, `Enumerable.Empty<int>` creates an empty collection of integer types.

Range

The `Range` extension method fills a new sequence of numbers. The arguments are `start` and `count`. For example, `Enumerable.Range(0, 10)` creates a sequence starting with 0 and containing 10 integers—0, 1, 2, 3, 4, 5, 6, 7, 8, 9. Here is a fragment that creates a range of



numbers from 10 to 109 and uses the `Array.ForEach` method and an implicit generic `Action` delegate to display the sequence.

```
var sequence = Enumerable.Range(10/*start*/, 100/*count*/);
Array.ForEach(sequence.ToArray(), n=>Console.WriteLine(n));
Console.ReadLine();
```

Repeat

If you want to fill a sequence with a repeated value, use the `Repeat` extension method. (These generation methods are useful for producing simple test values.) The following fragment puts one hundred occurrences of the word “test” in a sequence. (The extra two lines let you explore the sequence; you can plug this into any sample console application to test the code.)

```
var someStrings = Enumerable.Repeat("test", 100);
Array.ForEach(someStrings.ToArray(), s=>Console.WriteLine(s));
Console.ReadLine();
```

Listing 6.5 uses compound named type initialization and `Repeat` to generate some test objects. (This beats what we usually have to do—especially authors—copy and paste data changing random parts of the data, and it might be useful for testers, too.)

LISTING 6.5 Using Repeat and Compound Named Type Initialization to Create Some Test Data

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace RangeDemo2
{
    class Program
    {
        public class Customer
        {
            public int ID{ get; set; }
            public string CompanyName{ get; set; }
            public string City{ get; set; }
        }

        static void Main(string[] args)
        {
            Action<Customer> print = c =>
```

LISTING 6.5 Continued

```
Console.WriteLine("ID: {0}\nCustomer: {1}\nCity: {2}\n",
    c.ID, c.CompanyName, c.City);

var customers =
    Enumerable.Repeat<Customer>(new Customer{
        ID=DateTime.Now.Millisecond,
        CompanyName="Test Company",
        City="Test City"}, 100);

Array.ForEach(customers.ToArray(), c=>print(c));
Console.ReadLine();
}
```

This example has only one hundred duplicate `Customer` objects, but for testing purposes this might be sufficient.

NOTE

If you want some objects with different key values, this approach won't work. `Repeat` uses shallow copies so the one hundred `Customer` objects in the example are really just one hundred references to one object.

Equality Testing

There was a time in the not-too-distant past that you had to know a lot about a computer's BIOS (basic input/output system) and interrupts and interrupt handlers to perform everyday tasks. (Peter Norton was one of the people who led that education charge.) For example, interrupt 0x21 and functions 0x4E and 0x4F performed the task of finding the first and subsequent files based on a file mask. Knowledge of interrupts wasn't just for assembly language programmers either. Writing interrupt handlers 15 years ago was as common as writing to the Windows application programming interface (API) was just 5 or 10 years ago.

These capabilities—these BIOS services—are still alive and well, but they are tidily bundled up under the .NET Framework and underneath that the Windows API. (This is progress.) Now, you can read all of the folders on a path and with LINQ compare and figure out what is different with four method calls: `Directory.GetDirectories` (twice), `Enumerable.SequenceEqual`, and `Enumerable.Except` (set-difference). (And, Peter Norton—a hero to some early PC geeks—is living comfortably in Santa Monica off the proceeds from the sale of Norton Utilities.)

Listing 6.6 demonstrates the `SequenceEqual` extension method. This method compares a source and target sequence and returns a Boolean indicating whether they contain the same elements.

LISTING 6.6 `SequenceEqual` Is Used Here to Determine if Two Paths Contain the Same Folders and the `Set-Difference` Method `Except` Indicates What's Missing

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.IO;

namespace SequenceEqualDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            var folder1 = Directory.GetDirectories("C:\\\\");
            var folder2 = Directory.GetDirectories("C:\\\\TEMP");

            // are they equal
            Console.WriteLine("Equal: " + folder1.SequenceEqual(folder2));
            Console.ReadLine();

            // What's the difference?
            var missingFolders = folder1.Except(folder2);

            Console.WriteLine("Missing Folders: {0}", missingFolders.Count());
            Array.ForEach(missingFolders.ToArray(),
                folder=>Console.WriteLine(folder));
            Console.ReadLine();
        }
    }
}
```

For more on set operations, refer to Chapter 9.

Obtaining Specific Elements from a Sequence

The *element operations* are all implemented as extension methods without LINQ keywords. The element methods are: `ElementAt`, `ElementAtOrDefault`, `First`, `FirstOrDefault`, `Last`, `LastOrDefault`, `Single`, and `SingleOrDefault`. Each in its way is designed to return a single element from a sequence.

`ElementAt` returns a single element from a sequence at the specified index. `ElementAtOrDefault` returns an element at the index or a default value if the element is out of range. `First` returns the first element of a sequence, and `FirstOrDefault` returns the first element of a sequence matching the criteria or a default value. `Last` and `LastOrDefault` methods act like `First` and `FirstOrDefault` but act on the last element. `Single` returns the only element if there is just one, and `SingleOrDefault` returns the only element that satisfies a condition or a default value. Listing 6.7 demonstrates each of these extension methods.

LISTING 6.7 Demonstrating Element Operators

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data;
using System.Data.SqlClient;

namespace ElementDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            var numbers = new int[]{1, 2, 3, 4, 5, 6, 7, 8, 9};

            // first
            var first = (from n in numbers select n).First();
            Console.WriteLine("First: {0}", first);
            var firstOrDefault = (from n in numbers select n).FirstOrDefault(
                n=>n>10);
            Console.WriteLine("First or default: {0}", firstOrDefault);

            // last
        }
    }
}
```

LISTING 6.7 Continued

```

var last = (from n in numbers select n).Last();
Console.WriteLine("Last: {0}", last);
var lastOrDefault = (from n in numbers select n).LastOrDefault(
    n=>n<5);
Console.WriteLine("Last or default: {0}", lastOrDefault);

// single
var single = (from n in numbers where n==5 select n).Single();
Console.WriteLine("Single: {0}", single);
var singleOrDefault = (from n in numbers select n).SingleOrDefault(
    n=>n==3);
Console.WriteLine("Single or default: {0}", singleOrDefault);

// element at
var element = (from n in numbers where n < 8 select n).ElementAt(6);
Console.WriteLine("Element at 6: {0}", element);

var elementOrDefault = (from n in numbers select n).ElementAtOrDefault(11);
Console.WriteLine("Element at 11 or default: {0}", elementOrDefault);
Console.ReadLine();
}

}
}

```

Appending Sequences with Concat

Concat is an extension method that concatenates one sequence onto a second. There are no LINQ keywords for concatenation. You can use `Concat` on an explicit sequence, providing the second sequence as the argument to `Concat`, or you can invoke `Concat` on an implicit sequence at the end of a query, as shown in Listing 6.7 with the element operators. Listing 6.8 demonstrates concatenating two sequences of numbers together. (See Figure 6.4 for the visualization.)

LISTING 6.8 Concatenating Sequences Together with the `Concat` Extension Method

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

```

LISTING 6.8 Continued

```

using System.Data;
using System.Data.SqlClient;

namespace ElementDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            var numbers = new int[]{1, 2, 3, 4, 5, 6, 7, 8, 9};
            var moreNumbers = new int[]{10, 11, 12, 13, 14, 15};
            foreach( var n in numbers.Concat(moreNumbers))
                Console.WriteLine(n);
            Console.ReadLine();
        }
    }
}

```

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

9

Concatenated to the sequence

10, 11, 12, 13, 14, 15

yields the new sequence

1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15

FIGURE 6.4 Concatenation appends two sequences to make a new sequence containing all elements.

Summary

This chapter introduced some of the standard query operators that make up part of the full capability of the .NET Framework. Many of these are implemented only as extension methods, but can be easily integrated alongside LINQ using simple method invocation syntax. Included in this group are filtering, quantifiers, partitioning, generation, equality testing, element operations, and concatenation.

There are additional and substantial LINQ keywords and query operations that are covered in detail in Part II (which makes up the next six chapters).

This page intentionally left blank

PART II

LINQ for Objects

IN THIS PART

CHAPTER 7	Sorting and Grouping Queries	137
CHAPTER 8	Using Aggregate Operations	151
CHAPTER 9	Performing Set Operations	167
CHAPTER 10	Mastering Select and SelectMany	185
CHAPTER 11	Joining Query Results	211
CHAPTER 12	Querying Outlook and Active Directory	239

This page intentionally left blank

CHAPTER 7

Sorting and Grouping Queries

IN THIS CHAPTER

- ▶ Sorting Information
- ▶ Grouping Information

"If they want peace, nations should avoid the pin-pricks that precede cannon shots."

—Napoleon Bonaparte

The most common thing after selecting data is ordering that data. This chapter explores sorting information as well as the reverse capability of Language Integrated Query (LINQ), and looks at the brother of the select, grouping. LINQ supports ascending and descending sorts as well as secondary sorts. If you are familiar with something like Structured Query Language (SQL) sorting, the concepts will be familiar and the implementation will be relatively easy to grasp.

Grouping concepts are similar to SQL grouping concepts, too. Although the implementation looks a little different, the general result—create a resultset out of multiple sources—is the same.

As mentioned in previous chapters, all of LINQ sits squarely on top of extension methods. Some capabilities in LINQ have LINQ keywords and some are only available through extension methods. Where only an extension method exists, it is pointed out in the material.

Sorting Information

LINQ supports sorting in ascending and descending order with the `orderby` keyword. LINQ also supports secondary sorts in both ascending and descending order by adding comma-delimited lists of items on which to sort.

A collection of items can also be reversed by invoking the `Reverse` extension method on a collection. `Reverse` is not a supported LINQ keyword in C#.

TIP

If you look at the Microsoft Developer Network (MSDN) help, it indicates which extension methods have LINQ equivalents, for example, sorting is handled by the `OrderBy` extension method and the `orderby` keyword, but `Reverse` is only supported in C# by calling the extension method.

Sorting in Ascending and Descending Order

.NET 3.5 introduces the extension methods `OrderBy`, `OrderByDescending`, `ThenBy`, `ThenByDescending`, and `Reverse`. These methods extend `Enumerable` and `Queryable`. Many LINQ keywords are nearly identical to their underlying extension methods and others are implicit. For example, sorts are ascending by default; if you add the `descending` keyword, the extension method `methodnameDescending` is invoked. Some LINQ capabilities use extension methods but don't have LINQ keywords. For example, `ThenBy` performs secondary sorts but it is the LINQ query and context that implicitly indicate that `ThenBy` needs to be emitted. (Refer to the subsection "Performing Secondary Sorts" to see a scenario that brings `ThenBy` into play.)

Listing 7.1 demonstrates how to split a string of words and sort those words in ascending order (see Figure 7.1 for a visualization), and Listing 7.2 shows how to introduce the `descending` keyword to convert the LINQ to a reverse-sort order sort. The introduction of the `descending` keyword implicitly instructs the compiler to emit the `OrderByDescending` extension method call. (For the remainder of this chapter, the underlying extension method isn't explicitly stated by name unless there is no LINQ keyword(s) that provides that behavior.)



FIGURE 7.1 The output from Listing 7.1 shows the words in the phrase sorted in ascending order.

LISTING 7.1 Splitting a String of Words and Sorting Them in Ascending Order

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace SortingDemo
{
```

LISTING 7.1 Continued

```

class Program
{
    static void Main(string[] args)
    {
        // inspired
        string quote =
            "Most people have the will to win, few have the will to prepare to win.";

        // make an array dropping empty items
        string[] words = quote.Split(new char[]{' ', ',', '.', ','},
            StringSplitOptions.RemoveEmptyEntries);

        // sort ascending
        var sorted = from word in words orderby word select word;

        foreach (var s in sorted)
            Console.WriteLine(s);

        Console.ReadLine();
    }
}

```

In Listing 7.1, the query is assigned to the anonymous type `sorted`, and the sort behavior is introduced by the `orderby word` clause. Can you guess the compiler-generated type of the variable `sorted`? If you answered `IEnumerable<string>`, you are on top of things. The actual type is `IOrderedEnumerable<string>`, but you wouldn't know that without detailed exploration of the .NET Framework and the emitted Microsoft Intermediate Language (MSIL). Listing 7.2 shows the introduction of the descending modifier on the `orderby` clause.

LISTING 7.2 Sorting the Same Array of Words in Descending Order

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace SortDescending
{

```

LISTING 7.2 Continued

```

class Program
{
    static void Main(string[] args)
    {
        // inspired
        string quote =
            "Most people have the will to win, few have the will to prepare to win.";

        // make an array dropping empty items
        string[] words = quote.Split(new char[] { ' ', ',', '.' },
            StringSplitOptions.RemoveEmptyEntries);

        // sort descending
        var descending = from word in words orderby word descending select word;

        foreach (var s in descending)
            Console.WriteLine(s);

        Console.ReadLine();
    }
}

```

The presence of the `descending` keyword in Listing 7.2 causes the compiler to emit a call to `Enumerable.OrderByDescending`. (Refer to Figure 7.2 to see the output from Listing 7.2.)



FIGURE 7.2 The same quote from Bobby Knight—basketball coach—sorted in descending-word order.

Sort in Descending Order Using the Extension Method Directly

To demonstrate the correlation between LINQ and extension methods, Listing 7.3 contains code that fundamentally is identical to Listing 7.2. The difference is that the code calls the `OrderByDescending` extension method directly with a Lambda Expression (`s => s`) indicating the `Key` that describes the `Element` to sort on.

LISTING 7.3 Sorting in Descending Order by Invoking the Extension Method Directly and Providing the Lambda Expression That Indicates the Sort Key

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace SortDescExtensionMethod
{
    class Program
    {
        static void Main(string[] args)
        {
            // inspired
            string quote =
                "Most people have the will to win, few have the will to prepare to win.";

            // make an array dropping empty items
            string[] words = quote.Split(new char[]{' ', ',', '.'},
                StringSplitOptions.RemoveEmptyEntries);

            // use extension method directly to sort descending
            foreach (var s in words.OrderByDescending(s => s))
                Console.WriteLine(s);

            Console.ReadLine();
        }
    }
}

```

Performing Secondary Sorts

A primary sort is the first field or predicate in the `orderby` clause. A secondary sort in extension-method speak is the `ThenBy` extension method. For LINQ, just use a comma-delimited list; items after the first item in the `orderby` clause become secondary sort criteria. Listing 7.4 sorts anonymous gamblers in ascending order by `LastName` and `Age` (see Figure 7.3). `Age` is the secondary sort criteria. Listing 7.5 demonstrates the secondary sort with the secondary sort item in descending order (see Figure 7.4). A secondary sort field with the `descending` modifier emits the `ThenByDescending` extension method.

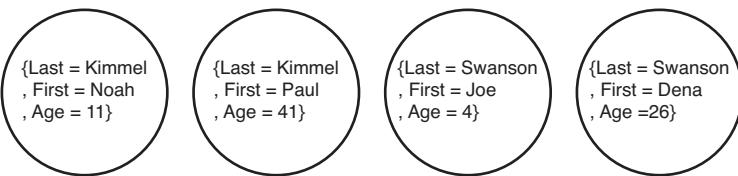


FIGURE 7.3 The gamblers with a secondary sort on age, both in ascending order. (The field names were shortened using compound type initialization to conserve page-space.)

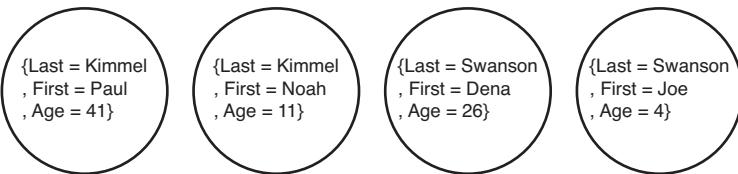


FIGURE 7.4 Now the younger people with the same last name follow the older people.

LISTING 7.4 Sorting on LastName and Age; Age Is the Secondary Sort Predicate

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace SecondarySort
{
    class Program
    {
        static void Main(string[] args)
        {
            var gamblers = new []
            {
                new {LastName="Kimmel", First="Paul", Age=41},
                new {LastName="Swanson", First="Dena", Age=26},
                new {LastName="Swanson", First="Joe", Age=4},
                new {LastName="Kimmel", First="Noah", Age=11};

                // thenby is implicit in second sort argument
                var sordid = from gambler in gamblers orderby gambler.LastName, gambler.Age
```

LISTING 7.4 Continued

```
    select gambler;

    foreach(var playa in sordid)
        Console.WriteLine(playa);

    Console.ReadLine();
}
}
}
```

LISTING 7.5 A Secondary Sort Where the Secondary Criteria Is in Descending Order

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace SecondarySort
{
    class Program
    {
        static void Main(string[] args)
        {
            var gamblers = new []
            {
                new {LastName="Kimmel", First="Paul", Age=41},
                new {LastName="Swanson", First="Dena", Age=26},
                new {LastName="Swanson", First="Joe", Age=4},
                new {LastName="Kimmel", First="Noah", Age=11};

            // thenby is implicit in second sort argument
            var sordid = from gambler in gamblers
                orderby gambler.LastName, gambler.Age descending
                select gambler;

            foreach(var playa in sordid)
                Console.WriteLine(playa);

            Console.ReadLine();
        }
    }
}
```

In Figure 7.3, the sort is name order followed by age order within the same last name. Younger gamblers are shown first. It should be no surprise—if you are familiar with sorts—that Figure 7.4 has simply reversed the order within the same name, as in Listing 7.5, with the younger gamblers last. (An anonymous type was used to generate the actual images, shortening the field names LastName and FirstName to First and Last, to create a better page fit.)

Reversing the Order of Items

The reverse behavior simply changes the order in a set; with n elements, the n^{th} becomes the first item, the $n^{\text{th}}-1$ item is second, and so on, and the first item is the last item after the reverse behavior is invoked. This happens regardless of the ordering of the items.

There is no reverse keyword in LINQ, so if you want to use it, you have to invoke the extension method. Listing 7.6 reorders the words of inspiration from the irascible Bobby Knight (winningest NCAA basketball coach in history). Refer to Figure 7.5 for the output.



FIGURE 7.5 The extension method Reverse in action.

LISTING 7.6 Invoking the Reverse Extension Method Reverses the Words of Inspiration from Texas Tech Coach Bobby Knight

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace SortingDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            // inspired
            string quote =
                "Most people have the will to win, few have the will to prepare to win./";

            // make an array dropping empty items
        }
    }
}
```

LISTING 7.6 Continued

```
string[] words = quote.Split(new char[] { ' ', ',', '.' },  
    StringSplitOptions.RemoveEmptyEntries);  
  
foreach (var s in words.Reverse())  
    Console.WriteLine(s);  
  
Console.ReadLine();  
}  
}  
}
```

In Listing 7.6, `String.Split` is used to split the quote by spaces, periods, and commas. The `StringSplitOptions.RemoveEmptyEntries` value removes whitespace or empty entries, such as would appear after the comma followed by a space (“...will to win, few have the will”) in the middle of the text.

Grouping Information

Many elements of LINQ are very similar to SQL programming concepts. It is important to remember, however, that LINQ works with custom objects, XML, and database data even though many of the Key constructs are like SQL constructs. One such construct is the ability to organize data by similarities into logical groupings.

Grouping behavior is supported by the `group by` clause in LINQ. In the code in Listing 7.7, the `group by` clause is used to organize an array of integers into logical groups by even and odd numbers (see Figure 7.6). Because the `group by` clause is pretty straightforward in the example, a nested `Array.ForEach` statement with nested Lambda Expressions were used to iterate over and display the two groups for added flavor. (You might encounter code like the nested `Array.ForEach` with a Lambda Expression with multiple statements, a ternary expression, and a second Lambda Expression, but this part of the code is not that straightforward.)



FIGURE 7.6 The group is represented as an instance of `GroupedEnumerable` containing multiple sequences referenced by a key.

LISTING 7.7 A group by Clause Organizing an Array of Integers into Odd and Even Sequences

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace GroupBy
{
    class Program
    {
        static void Main(string[] args)
        {
            var nums = new int[]{1,2,3,4,5,6,7,8,9,10};
            var result = from n in nums group n by n % 2;
            // really quite funky
            Array.ForEach(result.ToArray(), x=>{
                Console.WriteLine(x.Key==0? "evens:" : "odds:");
                Array.ForEach(x.ToArray(), y=>Console.WriteLine(y));});
            Console.ReadLine();
        }
    }
}
```

In the grouping, results are organized as two sequences with a **Key**. The **Key** is described by the group predicate, which is the result of $n \bmod 2$, or 0 and 1. In the result, if the **Key** is zero, you are iterating over the evens and if the **Key** is 1, you are iterating over the odds. If the group by predicate were $n \bmod 3$ ($n \% 3$), the result would contain three sequences with keys 0, 1, and 2.

The results of a group by are organized in **IGrouping** objects that contain a **Key** and **Element** pair. The **Key** represents the group that the datum belongs to and the **Element** represents the data. Listing 7.8 shows a common grouping of data. In the example, products and categories are read from the Northwind database into a list of custom objects and are grouped and ordered. (Refer to Figure 7.7 for an example of the output. The LINQ query is shown in bold.)

LISTING 7.8 Custom Objects Read from the Northwind Database and Grouped By a LINQ group...by...into Statement

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
```

LISTING 7.8 Continued

```
using System.Data;
using System.Data.SqlClient;

namespace GroupIntoDemo
{
    class Program
    {
        // set up a typical SQL join
        static string sql =
            "SELECT Categories.CategoryName, Products.ProductID, " +
            "Products.ProductName, Products.UnitsInStock, Products.UnitPrice " +
            "FROM Categories INNER JOIN " +
            "Products ON Categories.CategoryID = Products.CategoryID";

        // use the default northwind connection string
        static string connectionString =
            "Data Source=BUTLER;Initial Catalog=Northwind;Integrated Security=True";

        // define a simple class using automatic properties
        public class ProductItem
        {
            public string CategoryName{ get; set; }
            public int ProductID{ get; set; }
            public string ProductName{ get; set; }
            public int UnitsInStock{ get; set; }
            public decimal UnitPrice{ get; set; }

            public override string ToString()
            {
                const string mask =
                    "Category Name: {0}, " +
                    "Product ID: {1}, " +
                    "Product Name: {2}, " +
                    "Units In Stock: {3}, " +
                    "Unit Price: {4}";

                return string.Format(mask, CategoryName,
                    ProductID, ProductName, UnitsInStock,
                    UnitPrice);
            }
        }
    }
}
```

LISTING 7.8 Continued

```
        }

    }

    // a read helper
    static T SafeRead<T>(IDataReader reader, string name, T defaultValue)
    {
        object o = reader[name];
        if( o != System.DBNull.Value && o != null )
            return (T)Convert.ChangeType(o, defaultValue.GetType());

        return defaultValue;
    }

    static void Main(string[] args)
    {
        List<ProductItem> products = new List<ProductItem>();

        // read all of the data into custom objects
        using(SqlConnection connection = new SqlConnection(connectionString))
        {
            connection.Open();
            SqlCommand command = new SqlCommand(sql, connection);
            SqlDataReader reader = command.ExecuteReader();
            while(reader.Read())
            {
                products.Add(new ProductItem{
                    CategoryName=SafeRead(reader, "CategoryName", ""),
                    ProductID=SafeRead(reader, "ProductID", -1),
                    ProductName=SafeRead(reader, "ProductName", ""),
                    UnitsInStock=SafeRead(reader, "UnitsInStock", 0),
                    UnitPrice=SafeRead(reader, "UnitPrice", 0M)});
            }
        }

        // make sure I have some data
        Array.ForEach(products.ToArray(), y=>Console.WriteLine(y));
        Console.ReadLine();
    }
}
```

LISTING 7.8 Continued

```
Console.Clear();
string line = new string('-', 40);

// define the LINQ group
var grouped = from p in products
              orderby p.CategoryName, p.ProductName
              group p by p.CategoryName into grp
              select new {Category = grp.Key, Product = grp};

// dump each group
Array.ForEach(grouped.ToArray(), g=>
{
    Console.WriteLine(g.Category);
    Console.WriteLine(line);
    // dump each product in the group
    Array.ForEach(g.Product.ToArray(), p=>
        Console.WriteLine(p));
    Console.WriteLine(Environment.NewLine);
});

Console.ReadLine();

}

}
```

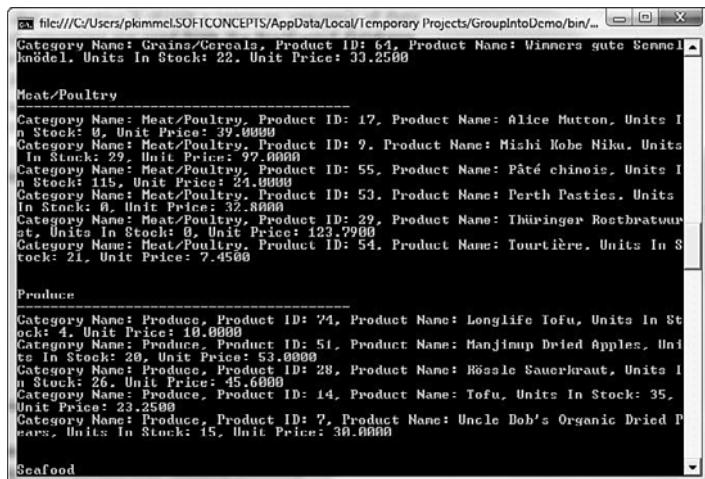


FIGURE 7.7 A partial view of the formatted result of the grouping query.

The projected result container grouped contains the product information sorted by CategoryName with a secondary sort on ProductName organized into groups by category (CategoryName). The CategoryName becomes the Key and each ProductItem object is stored in the Element named Product. The nested Array.ForEach dumped the grouped and ordered resultset. (You could try ObjectDumper for a simpler way to dump these projections. ObjectDumper was VB sample code. Look for the ObjectDumper sample code here: C:\Program Files\Microsoft Visual Studio 9.0\Samples\1033\CSharpSamples\LinqSamples\ObjectDumper.)

Finally, grouping is also supported by the extension method ToLookup. ToLookup creates a Dictionary of Key and value pairs. The difference is that with a Dictionary, you get a single Key and value pair, but ToLookup returns a Key and value pair with the value part being a collection of objects.

Summary

Sorting and grouping are an integral part of software development. LINQ has extensive support for both sorting and grouping. Primary and secondary sorts are supported directly and reversing the order of items is supported by the extension method Reverse. Grouping in LINQ creates Key and Element pairs. The Key defines the subgroup and the Element contains the sequence of items within that group.

CHAPTER 8

Using Aggregate Operations

“A great country worthy of the name does not have any friends.”

—Charles de Gaulle

The aggregate operations in this chapter are all implemented as extension methods only. You will use them in conjunction with Language Integrated Query (LINQ) on enumerable sequences before and after you run LINQ queries. Conceptually, the operations—`Aggregate`, `Count`, `LongCount`, `Min`, `Max`, `Average`, and `Sum`—are easy to grasp. For that reason, lots of code examples are included to give you some ideas how you might incorporate these operations in code, and as a means of exploring more of the C# language and the framework.

Aggregating

Aggregation operations compute a single value from a collection of values. The `Aggregate` extension method itself permits us to define a custom aggregate operation (as compared with, for example, `Sum`, which computes the sum of a collection). To demonstrate, the sample in this section begins with three distinct and separate sequences and aggregates them into a single sequence. This is accomplished with the `Enumerable.Empty` method and the `Union` set operation (see Listing 8.1).

IN THIS CHAPTER

- ▶ Aggregating
- ▶ Averaging Collection Values
- ▶ Counting Elements
- ▶ Finding Minimum and Maximum Elements
- ▶ Summing Query Results
- ▶ Median: Defining a Custom Aggregation Operation

LISTING 8.1 Custom Aggregation with the Aggregate Extension Method

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

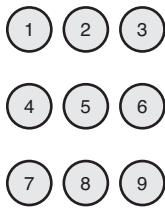
namespace EmptyCollectionDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            int[] numbers1 = new int[]{1, 2, 3};
            int[] numbers2 = new int[]{4, 5, 6};
            int[] numbers3 = new int[]{7, 8, 9};

            List<int[]> all = new List<int[]>{
                numbers1, numbers2, numbers3};

            var allNumbers = all.Aggregate(
                Enumerable.Empty<int>(), (current,next) =>
                current.Union(next));

            Array.ForEach(allNumbers.ToArray(), n=>Console.WriteLine(n));
            Console.ReadLine();
        }
    }
}
```

As shown in Figure 8.1, the code begins with three sequences of integers and collects them into a single sequence. The variable `all` represents the list of lists, and the code invokes the `Aggregate` extension method on that sequence. The first argument `Enumerable.Empty` indicates the type of the accumulator—that is, this method call is accumulating sequences of integers. The second argument is a Lambda Expression satisfying the generic delegate `Func` implicitly. The Lambda Expression has two inputs, `current` and `next`. The sequence `current` is Unioned to `next` successively until a single aggregate sequence is yielded. (For more on union and set operations, refer to Chapter 9, “Performing Set Operations.”)



Aggregate Union



FIGURE 8.1 The three integer arrays followed by the aggregate of all three Unioned together.

Visual Studio actually permits stepping through single methods and Lambda Expressions as if they were multiple statements (which, in actuality, they are, at least from an execution standpoint). The first time the `Union` statement is called, `current` is empty and `next` is the first sequence in the list of lists named "`all`". Then, `current` contains the elements from the first sequence, `next` is the next sublist, and so on until all of the values have been accumulated into the aggregate target. When finished, "`allNumbers`" contains all of the integers as a single list in this example. That is, all of the lists have been accumulated into a single list.

Because Listing 8.1 uses instance syntax—`all.Aggregate`—the real first argument is the object "`all`" itself. With instance syntax, as opposed to class syntax—`Enumerable.Aggregate`—you could also omit the accumulator argument and write the call to the `Aggregate` function, as follows:

```
var allNumbers = all.Aggregate((current,next) =>
    current.Union(next).ToArray<int>());
```

Or use class syntax, pass in the source, and provide the `Accumulator` and the `Func` generic delegate initialized by a Lambda Expression:

```
var allNumbers = Enumerable.Aggregate(all, Enumerable.Empty<int>(),
    (current, next)=>current.Union(next));
```

There are a couple more variations that work. For all of the overloaded versions of `Aggregate`, refer to the help documentation.

`Aggregate` is the general extension method for accumulating sequences into single values. For convenience, common aggregation operations for counting, summing, averaging, and computing minimum and maximum values have been included.

Averaging Collection Values

The Average extension method—as you probably guessed—accepts an input sequence and returns the single value representing the average of those values as a single result.

NOTE

In case a refresher is needed, an average is computed by summing all of the values in a group and dividing by the number of elements in that group.

This section shows three examples. Listing 8.2 computes a simple average. Listing 8.3 blends File IO and computes the average file size in a folder, and Listing 8.4 reads words in a text file and determines the average word length. Figure 8.2 shows the results for Listing 8.2, and Figure 8.3 shows the results for Listing 8.4. (No figure was provided for the file information as the `FileInfo` content and the number of files was too big to show in any meaningful way based on page-size constraints.)

LISTING 8.2 Computing the Simple Average

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace AverageDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            int[] numbers = new int[] { 13, 24, 5, 6, 78, 23 };
            Console.WriteLine(numbers.Average());
            Console.ReadLine();
        }
    }
}
```

LISTING 8.3 Read a List of Files and Determine the Average File Size (in Megabytes)

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.IO;
```

LISTING 8.3 Continued

```
namespace AverageFileSizeDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            string[] files = Directory.GetFiles("c:\\temp\\");
            double averageFileSize = (from file in files
                select (new FileInfo(file)).Length).Average();
            averageFileSize = Math.Round(averageFileSize/1000000, 2 );
            Console.WriteLine("Average file size (in c:\\temp) {0} Mbytes",
                averageFileSize );
            Console.ReadLine();
        }
    }
}
```



Average (Rounded to Two Places)



FIGURE 8.2 An array of integers and the average value (rounded to two decimal places).

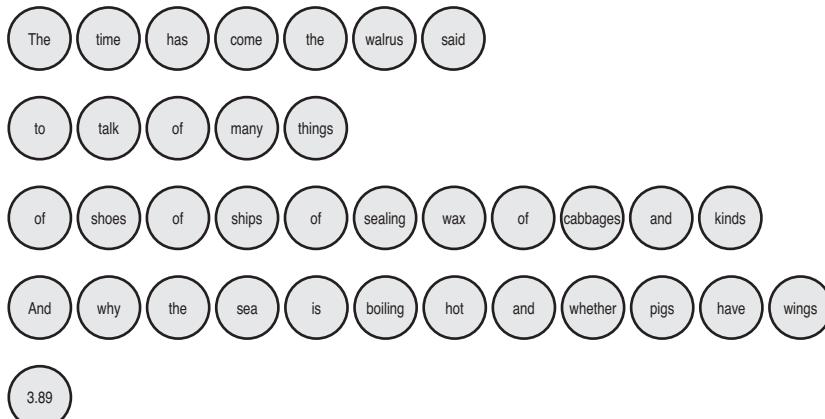


FIGURE 8.3 The four lines excerpted from Lewis Carroll's *Alice in Wonderland* and the average word length.

Listing 8.4 reads a text file, splits the lines into single words, and determines the average word length. The output is shown as a visualization in Figure 8.3.

LISTING 8.4 Averaging the Length of Words in a Text File

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Text.RegularExpressions;
using System.IO;

namespace AverageWordLengthInFile
{
    class Program
    {
        static void Main(string[] args)
        {
            string[] lines = File.ReadAllLines("../..\\WalrusAndCarpenter.txt");
            // sanity check
            Array.ForEach(lines, line => Console.WriteLine(line));

            Func<string, string[]> Split = str=>str.Split(new char[]{'-', ',', ' ', '.', ','}, 
                StringSplitOptions.RemoveEmptyEntries);

            List<string> all = new List<string>();
            //split out words
            Array.ForEach(lines, line => all.AddRange(Split(line)));

            // sanity check
            Array.ForEach(all.ToArray(), word => Console.WriteLine(word));

            //send result
            Console.WriteLine("Average word length {0}",
                Math.Round(all.Average(w => w.Length), 2));
            Console.ReadLine();
        }
    }
}
```

A variation on Listing 8.4 might be to split the lines of text into arrays of strings and then aggregate the strings into a single collection. It's worth noting that the relative complexity of the code remains about the same as that shown in Listing 8.4.

Counting Elements

Counting elements is supported by the extension methods `Count` and `LongCount`. Use `Count` to return an integer count for sequences less than the maximum value for an integer (about 2 billion) and the `LongCount` extension method for very large sequences. Both `Count` and `LongCount` are generic extension methods; generally, the parameterized type can be determined from the calling object. Refer to Listing 8.5 in the following section for an example that incorporates the `Count` extension method.

Finding Minimum and Maximum Elements

Sometimes, authors forget that even seemingly simple things are not so simple for everyone. I was reminded of this recently while studying a subject that I am by no means an expert in. For this reason, this section begins with a pedagogical example. The example, shown in Listing 8.5, declares an array of floating-point numbers and simply uses the `Min` and `Max` extension methods to determine the minimum and maximum values in the sequence. Figure 8.4 depicts the results visually.

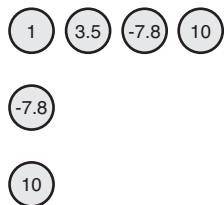


FIGURE 8.4 The minimum and maximum values of the sequence defined by numbers in Listing 8.5.

LISTING 8.5 Determining the Minimum and Maximum Values in a Simple Numeric Sequence

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
```

LISTING 8.5 Continued

```
namespace MinMaxNumber
{
    class Program
    {
        static void Main(string[] args)
        {
            var numbers = new float[] { 1.0f, 3.5f, -7.8f, 10f };
            Console.WriteLine("Min: {0}", numbers.Min());
            Console.WriteLine("Max: {0}", numbers.Max());
            Console.ReadLine();
        }
    }
}
```

From Listing 8.5, you can quickly determine that `Min` and `Max` behave as expected, at least when applied directly to a sequence.

Although `Min` and `Max` are not keywords in LINQ, you can incorporate these extension methods within a query or against the results of a sequence or query. In Listing 8.6, an array of video game titles is provided. `Min` is used in the first instance to return the length of the shortest title and in the second instance `games.Min` and the Lambda Expression `t => t.Length` is used as an operand in the `where` clause to return the title or titles whose title length is shortest. (In the example, “Halo 3” has the shortest title.)

LISTING 8.6 Using the `Min` Function in a `Where` Predicate

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace MinWordDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            var games = new string[] { "Halo 3", "Call of Duty 4", "Project Gotham 3" };
            Console.WriteLine("Shortest title: {0}", games.Min(t => t.Length));

            // shortest title name
            var title = from game in games
```

LISTING 8.6 Continued

```
        where game.Length ==
              games.Min(t => t.Length)
        select game;

        Array.ForEach(title.ToArray(), s => Console.WriteLine(s));
        Console.ReadLine();
    }
}
}
```

Listing 8.7 is the most complicated example in this section by far. Listing 8.7 demonstrates three ways in which you can ascertain which file in the C:\Windows folder is the largest. The first example applies the Take extension method on the sequence returned by the first query. The results are captured in the anonymous type named `max`. The second query uses the anonymous projection containing the filename and file size returning the maximum file size. We can only get the size value because the projection doesn't have a default implementation for object comparisons.

And, finally, in the third query captured in `max2` (in Listing 8.7), we can get the object containing the maximum size while returning the whole object because the results of the third query are placed in a sequence of `Temp` objects. A Lambda Expression for `Max` isn't needed in the last query because `Temp` implements `IComparable` and consequently `Max` returns a `while` object. Unfortunately, while `max2` contains the `Max Temp` object, the benefit of an anonymous project is lost.

Adding Support for Projecting Interfaces

A neat feature for LINQ would be to support implementing interfaces for anonymous projections. For example, if you were to provide a Lambda Expression for an element named `CompareTo`, LINQ could infer that the anonymous project—`FileName`, `Size`, and `CompareTo`—implements `IComparable`.

In fact, you can initialize a compound type with a Lambda Expression. If you drop the call to `Max(s=>s.Size)` from the query for `max1` in Listing 8.7 and add a helper method, you can initialize a named value `CompareTo` to a Lambda Expression that will invoke `IComparable`'s `CompareTo` method. Here is the revised query and the helper function:

```
var max1 = (from fileName in files
            let info = new FileInfo(fileName)
            select
                new { FileName = info.Name, Size = info.Length,
                      CompareTo=GetCompare(info.Length)});
```

```
private static Func<T, int> GetCompare<T>(T y) where T : IComparable
{
    return k => y.CompareTo(k);
}
```

LISTING 8.7 Finding the Largest File in a Folder With and Without an Anonymous Type and the IComparable Interface

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.IO;

namespace MinMaxDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            string[] files = Directory.GetFiles("C:\\WINDOWS",
                "*.*", SearchOption.AllDirectories);
            Console.WriteLine(files.Count());

            // gets maximum
            var max = (from fileName in files
                       let info = new FileInfo(fileName)
                       orderby info.Length descending
                       select
                           new { FileName = info.Name, Size = info.Length }).Take(1);

            Console.WriteLine("Using Take: {0}", max.ElementAt(0));

            // with anonymous type we have to indicate what to get the max of
            var max1 = (from fileName in files
                        let info = new FileInfo(fileName)
                        select
                            new { FileName = info.Name,
```

LISTING 8.7 Continued

```
        Size = info.Length
    }
).Max(s=>s.Size);
Console.WriteLine("Using Max with anonymous type: {0}", max1);

// with named type we lose convenience of anonymous type but get
// inheritance, realization, for example we can implement IComparable
// and get the max of the whole object
var max2 = (from fileName in files
            let info = new FileInfo(fileName)
            select
                new Temp{ FileName = info.Name, Size = info.Length}).Max();

Console.WriteLine("Using Max: {0}", max2);
Console.ReadLine();
}

public class Temp : IComparable<Temp>
{
    public string FileName { get; set; }
    public long Size { get; set; }
    public int CompareTo(Temp o)
    {
        return Size.CompareTo(o.Size);
    }
    public override string ToString()
    {
        return string.Format("FileName: {0}, Size: {1}", FileName, Size);
    }
}
```

Obtaining `FileInfo` details on a big folder like the C:\Windows folder might take a few extra seconds. Be patient for the code in Listing 8.7 to begin producing output.

Last, but not least, this section introduced the `let` clause. The LINQ keyword `let` permits storing the results of a subexpression to facilitate using it later in the query. Think of `let` as defining a temporary result within the body of a query, or as kind of a shortcut for a subexpression. In Listing 8.7, the `let` is used to create an instance of `FileInfo` that is used to initialize the projection's `FileName` and `Size` values in each of the queries.

Summing Query Results

`Sum` is a predefined aggregate operation that tallies a sequence. The example in Listing 8.8 reads a list of comma-separated values from a text file using a `let` clause to split each line of text and convert them into a collection of sequences. Each sequence contains the values converted to integers. The `for` loop employs the `Sum` extension method and a Lambda Expression that totals the elements by their columnar position using `ElementAt`. (`ElementAt` is roughly equivalent to indexing a sequence like an array.) The sample data is shown as a block comment at the end of Listing 8.8. (Refer to Figure 8.5 for a visualization of the result.)

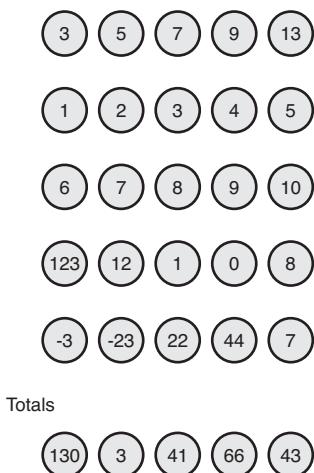


FIGURE 8.5 The sequences as read from the text file and the totals (after being added to a list for display purposes).

LISTING 8.8 Summing Column Values from a Comma-Delimited File

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.IO;

namespace SummingColumnValues
{
    class Program
    {
        static void Main(string[] args)
        {
```

LISTING 8.8 Continued

```
string[] lines = File.ReadAllLines("../..\\..\\csvData.txt");

// make sure we have some data
foreach (string s in lines)
    Console.WriteLine("{0}", s);

var results =
    from line in lines
    let values = line.Split(',')
    let y = values
    select (from str in y
            select Convert.ToInt32(str));

Console.WriteLine();
for (int i = 0; i < results.Count(); i++)
{
    Console.WriteLine("Total Column {0}: {1}", i+1,
        results.Sum(o => o.ElementAt(i)));
}
System.Threading.Thread.Sleep(5000);
}

/*
 * Data:
 * 3,5,7,9,13
 * 1,2,3,4,5
 * 6,7,8,9,10
 * 123,12,1,0,8
 * -3,-23,22,44, 7
 */
}
```

∞

Median: Defining a Custom Aggregation Operation

The extension method `Aggregate` is the general method for writing custom aggregation behavior. As demonstrated in this chapter, several convenience methods are provided for common kinds of aggregation behavior. You can also use the extension method capabilities of .NET and implement custom aggregation methods. For example, you could define a method `Median` that returns the middle value.

The basic idea is that the median value is the value in an ordered list that divides the lower half and upper half of a sequence. A rigorous example would split an odd-numbered list by the middle element and an even numbered list might return the average of the middle two elements. The example in Listing 8.9 depends on the list being ordered and simply returns the middle element.

LISTING 8.9 A Lightweight Implementation of the Aggregate Operation Median

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace MedianOperation
{
    class Program
    {
        static void Main(string[] args)
        {
            var numbers = new int[]{3,4,5,6,7,8,9,10};
            Console.WriteLine(numbers.Median());
            Console.ReadLine();
        }
    }

    public static class MyAggregate
    {
        public static T Median<T>(this IEnumerable<T> source)
        {
            return source.ToArray()[source.Count()/2];
        }
    }
}
```

To improve the example, you could add a LINQ query that orders the elements by modifying `Median` as follows:

```
public static T Median<T>(this IEnumerable<T> source)
{
    var ordered = from s in source orderby s select s;
    return source.ToArray()[ordered.Count()/2];
}
```

For the revision to work, the type of `T` must implement `IComparable` (which it does for integers). You would have to hand code the `IComparable` interface for custom objects, which means you would have to express the output type in the `select` clause—from a type that implements `IComparable`—rather than using an anonymous projection. (Remember, an anonymous projection happens when you use `select new {named type=value1, named type=value1}` without expressing a literal type after the `new` keyword. Listings 8.3 and 8.7 demonstrate how to use a known type in the `select` clause.)

Summary

Extension methods might be one of the singularly most powerful and essential capabilities of .NET. Extension methods represent a core feature that makes LINQ work. Some extension methods are implemented as LINQ keywords and some—like those in this chapter—do not have LINQ keywords. However, as demonstrated with the aggregate methods in this chapter, you can still combine extension methods to sequences regardless of whether they are derived from a query, and these methods might be embedded in a query.

This chapter began with the general `Aggregate` method; the specific, common aggregation operations like `Sum`, `Min`, and `Max`; and introduced the `let` clause for storing subexpressions.

This page intentionally left blank

CHAPTER 9

Performing Set Operations

IN THIS CHAPTER

- ▶ Finding Distinct Elements
- ▶ Defining Exclusive Sets with Intersect and Except
- ▶ Creating Composite Resultsets with Union

“Ah, well, then I suppose I shall have to die beyond my means.”

—Oscar Wilde

Set operations are like knights in the game chess. Set operations—like knights—help you move in imaginative ways. The set operations are implemented as extension methods. `Distinct` selects a distinct element in a sequence. `Union` performs set addition, `Intersect` performs set arithmetic, and `Except` helps exclude elements from a set.

Because this chapter is composed of a relatively few number of features, a lot of code samples are included. The code samples are intended to help you think of ways to incorporate set logic into your applications. As you will see, some of the samples manipulate data from a database. Obviously, that data could be manipulated in Structured Query Language (SQL) beforehand, so the samples present an alternative to writing the SQL to do the same thing.

Finding Distinct Elements

`Distinct` is the only set operation that has a Language INtegrated Query (LINQ) keyword, but that is only in Visual Basic (VB). `Distinct` is designed to examine two sets and eliminate duplicates. `Distinct` can be invoked with a sequence and by invoking an overloaded form using an object that implements `IEqualityComparer`.

Listing 9.1 shows a list of numbers representing grades from a group of students (see Figure 9.1). The example invokes the `Distinct` operation to remove duplicates and then determines the median or middle grade. (A favorite of

students, a college professor might use this to “grade on a curve.”) In this listing, the sequence (or set) has duplicates at 72, 75, 81, and 92; these are removed from the sequence. The median grade is then 85. (Chapter 8, “Using Aggregate Operations,” includes an example of implementing a custom aggregator, a `Median` extension method.)

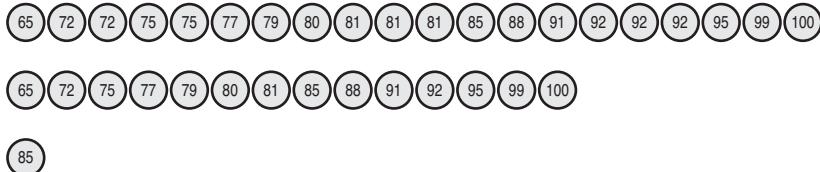


FIGURE 9.1 The original list of grades, after `Distinct` is applied, and the median value.

LISTING 9.1 Determining the Median Grade from a List of Numbers Representing Grades

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace DistinctDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            var grades = new int[]
            { 65, 72, 72, 75, 75, 77, 79, 80, 81, 81, 81, 85,
              88, 91, 92, 92, 92, 95, 99, 100 };

            // select distinct grades
            var distinct = grades.Distinct();
            Console.WriteLine("Median grade: {0}",
                distinct.ToArray<int>()[distinct.Count() / 2]);
            Console.ReadLine();
        }
    }
}
```

Finding Distinct Objects Using Object Fields

The example in this section combines SQL and custom objects. Assume you have a sales force and an objective to determine all of the unique cities in which you sell product. The example uses the `Distinct` method to determine the distinct list of cities (which might then be used to divvy up sales regions by sales representative).

Each part of the solution is provided in its own section with an elaboration on that part of the solution. The complete listing is provided at the end of this section.

Defining a Custom Order Object with Automatic Properties

You can define an entity class very quickly in C# using two methods. One is to use DevExpress's CodeRush product, which is a cool metaprogramming tool, and the next is to use automatic properties. In Listing 9.2, a custom `Order` class is defined using automatic properties. In conjunction with the automatic properties, nullable types are employed. Using nullable types permits assignment of `null` to fields that are actually null in the database rather than contriving arbitrary null representatives, such as the classic `-1` for integers.

LISTING 9.2 Creating a Custom Entity Class with Automatic Properties and Nullable Types

```
class Order
{
    public int? OrderID { get; set; }
    public string CustomerID { get; set; }
    public int? EmployeeID { get; set; }
    public DateTime? OrderDate { get; set; }
    public DateTime? RequiredDate { get; set; }
    public DateTime? ShippedDate { get; set; }
    public int? ShipVia { get; set; }
    public decimal? Freight { get; set; }
    public string ShipName { get; set; }
    public string ShipAddress { get; set; }
    public string ShipCity { get; set; }
    public string ShipRegion { get; set; }
    public string ShipPostalCode { get; set; }
    public string ShipCountry { get; set; }
}
```

Remember the basic rule is that automatic properties just mean that the basic setter and getter are defined for you and you can't refer to fields directly. (Refer to Chapter 7, "Sorting and Grouping Queries," for the introduction to automatic properties.) The absence of a constructor also means you can construct instances of `Order` with any combination of compound initialization your solution requires.

Instantiating Custom Objects from SQL

Entities are generally classes that represent tables in a database. These can be modeled either with an Entity Relational Diagram (a database modeling tool) or the Unified Modeling Language (and a UML tool). Generally, both kinds of models are not needed but one or the other is helpful. The biggest things to avoid are having entities in code that map to tables defined more than once in code and having entities in code for the sake of having entities. Some entities, such as linking tables, should be used to simply help construct composite objects.

Listing 9.3 uses the canonical Northwind database, a generic List (or Order objects), and a while loop to populate the list. The connection is created and the collection is accessible via an `IDataReader`. (In the example, the provider-agnostic types were used just to remind you of their availability.) It is also worth noting that the Order objects were constructed using compound type initialization introduced in Chapter 2, “Using Compound Type Initialization.”

LISTING 9.3 Constructing Entity Objects from a SQL Database Using a Generic List and Compound Type Initialization

```
public static List<Order> GetOrders()
{
    string connectionString =
        "Data Source=.\SQLExpress;AttachDbFilename=\"C:\\Books\\Sams\" +
        "\\\LINQ\\Northwind\\northwnd.mdf\";Integrated Security=True;Connect " +
        "Timeout=30;User Instance=True";

    List<Order> orders = new List<Order>();

    using ( IDbConnection connection = new SqlConnection(connectionString))
    {
        connection.Open();
        IDbCommand command = new SqlCommand("SELECT * FROM Orders");
        command.Connection = connection;
        command.CommandType = CommandType.Text;
        IDataReader reader = command.ExecuteReader();
        while(reader.Read())
        {
            orders.Add(new Order
            {
                OrderID = reader.IsDBNull(0) ? null : (int?)reader.GetInt32(0),
                CustomerID = reader.IsDBNull(1) ? null : reader.GetString(1),
                EmployeeID = reader.IsDBNull(2) ? null : (int?)reader.GetInt32(2),
                OrderDate = reader.IsDBNull(3) ? null :
                    (DateTime?)reader.GetDateTime(3),
                RequiredDate = reader.IsDBNull(4) ? null:

```

LISTING 9.3 Continued

```

        (DateTime?)reader.GetDateTime(4),
        ShippedDate = reader.IsDBNull(5) ? null :
            (DateTime?)reader.GetDateTime(5),
        ShipVia = reader.IsDBNull(6) ? null : (int?)reader.GetInt32(6),
        Freight = reader.IsDBNull(7) ? null : (decimal?)reader.GetDecimal(7),
        ShipName = reader.IsDBNull(8) ? null : reader.GetString(8),
        ShipAddress = reader.IsDBNull(9) ? null : reader.GetString(9),
        ShipCity = reader.IsDBNull(10) ? null : reader.GetString(10),
        ShipRegion = reader.IsDBNull(11) ? null : reader.GetString(11),
        ShipPostalCode = reader.IsDBNull(12) ? null : reader.GetString(12),
        ShipCountry = reader.IsDBNull(13) ? null : reader.GetString(13),
    }
}

return orders;
}
}

```

NORTHWIND DATABASE ISN'T INSTALLED WITH SQL SERVER 2005

The Northwind database is not installed with SQL Server 2005. You can download the Northwind database, if you don't already have a copy. As an aside, Northwind is generally being supplanted by the AdventureWorks database as the standard example database.

The only change you will need to make is to adjust the connection string to match the location of your copy of the Northwind Trading Company database. As a reminder, you can store the connection string in a .config file, and if you are interested, you can learn about writing a custom installer for dynamic database configuration in my article titled, "Implementing a Custom ConnectionString Installer for Setup" on developer.com at www.developer.com/security/article.php/11580_3704391_4.

Implementing an IEqualityComparer

To augment this example, you can perform custom comparisons on Order objects. Previously, it was mentioned that distinct cities are desired. By default, objects perform default referential comparisons. In this case, you want comparison by city. To accomplish this, you can implement an **IEqualityComparer** for Order objects.

Listing 9.4 demonstrates an example of **IEqualityComparer**. The basic rule is that two objects are considered equal if the **Equals** method returns true and the hash code of each object is the same. For that, you need to implement an **Equals** function that compares the

`ShipCity` of `Order` objects and returns the hash code from the `ShipCity` too. The result is shown in Listing 9.4.

LISTING 9.4 Implementing an `IEqualityComparer` By Implementing `Equals` and `GetHashCode`

```
class CityComparer : IEqualityComparer<Order>
{
    #region IEqualityComparer<Order> Members

    public bool Equals(Order x, Order y)
    {
        return x.ShipCity == null || y.ShipCity == null ?
            false : x.ShipCity.Equals(y.ShipCity);
    }

    public int GetHashCode(Order obj)
    {
        return obj.ShipCity == null ? -1 : obj.ShipCity.GetHashCode();
    }

    #endregion
}
```

Implementing a Simple Object Dumper

It is always helpful to be able to dump the state of an object. It is always too time consuming to implement such behavior manually for every kind of object. To make the state of an object available for debugging and testing, you can implement an object Dumper using reflection.

In Listing 9.5, the dumper uses a `StringBuilder` and `Reflection` to get the public properties—representing the state of an object—and iterate through the properties to display the name and value of each property. This is easier in .NET 3.5 because you can use the `Array.ForEach` method and the generic delegate `Action`. The `Action` is implicit as the second argument of the `ForEach` method.

LISTING 9.5 Creating a General Object Dumper Utility Based on Reflection and Properties

```
public static string Dump<T>(T obj)
{
    Type t = typeof(T);
    StringBuilder builder = new StringBuilder();
```

LISTING 9.5 Continued

```

 PropertyInfo[] infos = t.GetProperties();

 // feel the enmity of your peers if you write code like this
 Array.ForEach( infos.ToArray(), p => builder.AppendFormat(
     "{0}={1} ", p.Name, p.GetValue(obj, null) == null ? "" :
     p.GetValue(obj, null)));
 builder.AppendLine();

 return builder.ToString();
}

```

The comment in Listing 9.5, “feel the enmity of your peers if you write code like this,” is a bit lighthearted. A general rule of thumb is that it is okay to write esoteric code, such as the use of the ternary operator in the second argument of `ForEach`, as long as the following is true:

- ▶ All the code isn’t esoteric.
- ▶ Esoteric code is only used occasionally.
- ▶ It comprises the bulk of a singular—as opposed to monolithic—function.
- ▶ The code is in a disposable versus domain-significant method.
- ▶ If the code is buggy, it can be easily replaced with a more verbose form.

As a general rule, code like that in the `Dumper` is a little showy. However, this is a book, and a book is like a commercial for the new 620 Corvette. It’s okay to talk about going 220 miles per hour for marketing purposes because it’s fun, but on real city streets and highways (and in production code), it might be a little dangerous.

Listing 9.6 provides the complete listing as well as the code that uses the `Distinct` method. The `Main` function requests the list of `Order` objects with `GetOrders`. A quick dump of these objects reveals that we have everything. In Version 1—as noted by the comment—we can request the distinct list of cities by directly calling the extension method and passing in an instance of the `CityComparer`. Version 1 also uses `OrderBy` as an extension method to sort the distinct list of cities. Version 2 combines the sort and the call to `Distinct` in a single LINQ query. Notice that `Distinct` is used as an extension method in the second example tool; the only real difference between version 1 and 2 is how the sorting behavior is invoked.

LISTING 9.6 Sorting and Returning a Distinct List of Cities from the Northwind Order Table

```

using System;
using System.Collections.Generic;

```

LISTING 9.6 Continued

```
using System.Linq;
using System.Text;
using System.Data;
using System.Data.SqlClient;
using System.Reflection;
using System.Collections;

namespace DistinctDemo2
{
    class Program
    {
        /// <summary>
        /// Demonstrates distinct ship city
        /// </summary>
        /// <param name="args"></param>
        static void Main(string[] args)
        {
            List<Order> orders = GetOrders();
            Array.ForEach(orders.ToArray(), o => Console.WriteLine(Dump(o)));
            Console.WriteLine(orders.Count);

            // version 1 with extension method OrderBy
            var cities = orders.Distinct(new CityComparer());
            Array.ForEach(cities.OrderBy(o => o.ShipCity).ToArray(),
                orderedByCity => Console.WriteLine(orderedByCity.ShipCity));

            // version 2 with query
            var cities2 = from order in orders.Distinct(new CityComparer())
                         orderby order.ShipCity
                         select order;
            Array.ForEach(cities2.ToArray(),
                orderedByCity => Console.WriteLine(orderedByCity.ShipCity));
            Console.WriteLine(cities2.Count());
            Console.ReadLine();
        }

        public static string Dump<T>(T obj)
        {
            Type t = typeof(T);
            StringBuilder builder = new StringBuilder();

            PropertyInfo[] infos = t.GetProperties();
```

LISTING 9.6 Continued

```
// feel the enmity of your peers if you write code like this
Array.ForEach( infos.ToArray(), p => builder.AppendFormat(
    "{0}={1} ", p.Name, p.GetValue(obj, null) == null ? "" :
    p.GetValue(obj, null)));
builder.AppendLine();
return builder.ToString();
}

public static List<Order> GetOrders()
{
    string connectionString =
        "Data Source=.\SQLExpress;AttachDbFilename=\\\"C:\\Books\\Sams\\
        \"\\LINQ\\Northwind\\northwnd.mdf\\\";Integrated Security=True;Connect " +
        "Timeout=30;User Instance=True";

    List<Order> orders = new List<Order>();

    using(IDbConnection connection = new SqlConnection(connectionString))
    {
        connection.Open();
        IDbCommand command = new SqlCommand("SELECT * FROM Orders");
        command.Connection = connection;
        command.CommandType = CommandType.Text;
        IDataReader reader = command.ExecuteReader();
        while(reader.Read())
        {
            orders.Add(new Order
            {
                OrderID = reader.IsDBNull(0) ? null : (int?)reader.GetInt32(0),
                CustomerID = reader.IsDBNull(1) ? null : reader.GetString(1),
                EmployeeID = reader.IsDBNull(2) ? null : (int?)reader.GetInt32(2),
                OrderDate = reader.IsDBNull(3) ? null :
                    (DateTime?)reader.GetDateTime(3),
                RequiredDate = reader.IsDBNull(4) ? null :
                    (DateTime?)reader.GetDateTime(4),
                ShippedDate = reader.IsDBNull(5) ? null :
                    (DateTime?)reader.GetDateTime(5),
                ShipVia = reader.IsDBNull(6) ? null : (int?)reader.GetInt32(6),
                Freight = reader.IsDBNull(7) ? null : (decimal?)reader.GetDecimal(7),
                ShipName = reader.IsDBNull(8) ? null : reader.GetString(8),
                ShipAddress = reader.IsDBNull(9) ? null : reader.GetString(9),
                ShipCity = reader.IsDBNull(10) ? null : reader.GetString(10),
```

LISTING 9.6 Continued

```
        ShipRegion = reader.IsDBNull(11) ? null : reader.GetString(11),
        ShipPostalCode = reader.IsDBNull(12) ? null : reader.GetString(12),
        ShipCountry = reader.IsDBNull(13) ? null : reader.GetString(13),
    }
}
return orders;
}
}

class CityComparer : IEqualityComparer<Order>
{
#region IEqualityComparer<Order> Members
public bool Equals(Order x, Order y)
{
    return x.ShipCity == null || y.ShipCity == null ?
        false : x.ShipCity.Equals(y.ShipCity);
}

public int GetHashCode(Order obj)
{
    return obj.ShipCity == null ? -1 : obj.ShipCity.GetHashCode();
}
#endregion
}

class Order
{
    public int? OrderID { get; set; }
    public string CustomerID { get; set; }
    public int? EmployeeID { get; set; }
    public DateTime? OrderDate { get; set; }
    public DateTime? RequiredDate { get; set; }
    public DateTime? ShippedDate { get; set; }
    public int? ShipVia { get; set; }
    public decimal? Freight { get; set; }
    public string ShipName { get; set; }
    public string ShipAddress { get; set; }
}
```

LISTING 9.6 Continued

```
public string ShipCity { get; set; }
public string ShipRegion { get; set; }
public string ShipPostalCode { get; set; }
public string ShipCountry { get; set; }
}
}
```

Defining Exclusive Sets with Intersect and Except

The `Intersect` extension method compares an argument sequence with the source sequence returning only those elements in the source sequence that are also in the target sequence. The `Except` extension method returns those elements in the source object—the one invoking the `Except` method—that are not in the argument object.

Listing 9.7 contains an array of event numbers and a short sequence of Fibonacci numbers. The evens excluding the Fibonacci numbers are all of the evens excluding 2, 8, and 34, which are in the Fibonacci sequence. (Each of the sequences is shown in Figure 9.2.)

Fibonacci Numbers

My very good friend of long ago, Dan McCarthy, introduced me to Fibonacci numbers. Dan worked for the State Department, spoke seven languages, including Mandarin Chinese, and played classical piano. Dan was also very blind. When expressing incredulity at Dan's many talents, he told me he could do so many other things because he didn't (and couldn't) watch TV. Taken to heart, it is repeated by me to this day to others when asked how I have time to write books.

Leonardo of Pisa, or Fibonacci, a nickname, was a twelfth-century mathematician who helped spread the Hindu-Arabic numeral system—a positional decimal system—from which our modern decimal is derived. Fibonacci numbers were named after Leonardo, although he did not invent them. The credit for Fibonacci numbers goes to Virahanka, a sixth-century Indian mathematician.

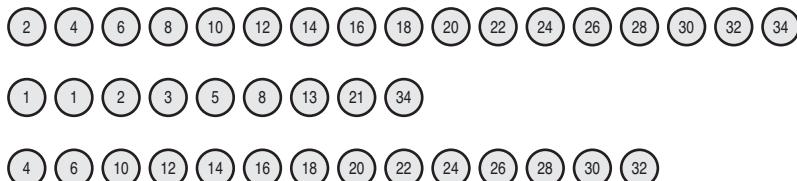


FIGURE 9.2 An array of even integers, a short Fibonacci sequence and the evens excluding those that are also in the Fibonacci sequence.

LISTING 9.7 Calculating a Set Difference and Displaying the Results

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ExceptDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            var evens = new int[] { 2, 4, 6, 8, 10, 12, 14, 16, 18,
                20, 22, 24, 26, 28, 30, 32, 34 };
            var fibos = new int[] { 1, 1, 2, 3, 5, 8, 13, 21, 34 };
            var setDifference = evens.Except(fibos);
            Array.ForEach<int>(setDifference.ToArray(), e => Console.WriteLine(e));
            Console.ReadLine();
        }
    }
}
```

Listing 9.8 is a throwback to Listing 9.6. In this example, the `Main` function is replaced by an exclusion list that returns all of the cities (for the top salesman) except for Münster and Albuquerque. In Listing 9.8, the `Distinct` method and `OrderBy` capability are used in their extension method form.

LISTING 9.8 Creating an Ordered and Distinct List of Customer Cities, Except for Those Specified in an Exclusion List

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data;
using System.Data.SqlClient;
using System.Reflection;
using System.Collections;

namespace ExceptDemo2
{
    class Program
```

LISTING 9.8 Continued

```
{  
    /// <summary>  
    /// Demonstrates except ship city  
    /// </summary>  
    /// <param name="args"></param>  
    static void Main(string[] args)  
    {  
        List<Order> orders = GetOrders();  
        Console.WriteLine(orders.Count);  
  
        List<Order> exclusions = new List<Order>();  
        exclusions.Add(new Order { ShipCity = "Münster" });  
        exclusions.Add(new Order { ShipCity = "Albuquerque" });  
        var cities = orders.Distinct(new CityComparer()).Except(  
            exclusions, new CityComparer());  
  
        Array.ForEach(cities.OrderBy(o => o.ShipCity).ToArray(),  
            orderedByCity => Console.WriteLine(orderedByCity.ShipCity));  
  
        Console.WriteLine(cities.Count());  
        Console.ReadLine();  
    }  
}
```

What Do Münster and Dr. Feynman Have in Common?

Münster has one of the first centers for nanotechnology. Introduced by Dr. Richard Feynman in 1959, a physicist of Manhattan Project fame, nanotechnology is the applied science of the control of matter on the atomic or molecular scale. Along with robotics (check out iRobot.com) and biotechnology with support from computer science, nanotechnology might be the industry of the twenty-first century—as soon as information technology has enjoyed a nice long run.

6

For a real fun read on nanotechnology from the author of *Jurassic Park* and *ER* fame, read Michael Crichton's *Prey*.

Intersect is an extension method that determines the elements found in the source that also exist in the argument sequence returning only elements found in both. Listing 9.9

returns the elements in both evens and the short Fibonacci sequence. (The visualized results are shown in Figure 9.3.)

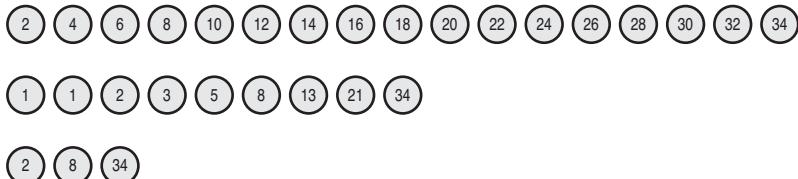


FIGURE 9.3 Even numbers, a short Fibonacci sequence, and the intersection of evens that are in the Fibonacci sequence.

LISTING 9.9 Set Intersection Returns a New Collection Containing the Elements Found in Both Collections

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace IntersectDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            var evens = new int[] { 2, 4, 6, 8, 10, 12, 14, 16, 18,
                20, 22, 24, 26, 28, 30, 32, 34 };
            var fibos = new int[] { 1, 1, 2, 3, 5, 8, 13, 21, 34 };
            var intersection = evens.Intersect(fibos);
            Array.ForEach<int>(intersection.ToArray(), e => Console.WriteLine(e));
            Console.ReadLine();
        }
    }
}
```

This chapter suggests that the extension methods described apply to collections. In actuality, these extension methods and LINQ apply to things that implement `IEnumerable`, which can be referred to in a general sense as collections; however, the extension methods and LINQ apply to things that implement `IQueryable`, too. `IQueryable` inherits from `IEnumerable`. (Remember that interfaces can also use inheritance.)

Listing 9.10 uses `Intersect` to find the files that exist in both .NET Framework 2.0 and 3.5. The code uses `Directory.GetFiles` from `System.IO` to get files in both framework directories and returns those in common in both directories. (There are surprisingly few

similarities given the way each framework folder is configured.) Such a technique could be used to find and remove duplicate files, for example.

LISTING 9.10 Comparing Files Found in Two Separate Directories

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.IO;
using System.Security.Permissions;

namespace IntersectDemo2
{
    class Program
    {
        static void Main(string[] args)
        {
            var v2 = Directory.GetFiles(
                @"C:\Windows\Microsoft.NET\Framework\v2.0.50727", "*.*",
                SearchOption.AllDirectories);
            var v35 = Directory.GetFiles(
                @"C:\Windows\Microsoft.NET\Framework\v3.5", "*.*",
                SearchOption.AllDirectories);

            var frameworkMatches = v35.Intersect(v2);
            Console.WriteLine("Framework matches between version 2 and 3.5");
            Array.ForEach(frameworkMatches.ToArray(), file => Console.WriteLine(file));
            var frameworkDifferences = v35.Except(v2);
            Console.WriteLine("Framework differences between version 2 and 3.5");
            Array.ForEach(frameworkDifferences.ToArray(), file => Console.WriteLine(new
FileInfo(file).Name));
            Console.ReadLine();
        }
    }
}
```

The second part of the example uses `Except` to find differences. There are many differences between the two folders. Both `Intersect` and `Except` in the example use the `Array.ForEach` method, a generic `Action` delegate, and a Lambda Expression to send the results to the console.

Creating Composite Resultsets with Union

In college, computer science and math majors study *discrete mathematics*. It is the study of the propositional and predicate calculus. Union is basically arithmetic in the predicate calculus.

The Union extension method adds two sequences together and returns the unique members found in both sequences. If you wrote code to add all of the elements of both sequences and then performed the Distinct operation on the result, you would get the same sequence as produced by Union. For example, given a sequence of 1, 2, 3, 4 and 2, 4, 6, the Union of these sequences will result in a new sequence with the elements 1, 2, 3, 4, 6. Union is set addition. The Union extension method returns all of the unique members found in both sets, excluding duplicates. Listing 9.11 contains a short Union example using the evens and fibos that we used earlier in the chapter.

LISTING 9.11 An Example Demonstrating Set Addition—Union Extension Method

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace UnionDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            var evens = new int[] { 2, 4, 6, 8, 10, 12, 14, 16, 18,
                20, 22, 24, 26, 28, 30, 32, 34 };
            var fibos = new int[] { 1, 1, 2, 3, 5, 8, 13, 21, 34 };
            var union = evens.Union(fibos);
            Array.ForEach<int>(union.ToArray(), e => Console.WriteLine(e));
            Console.ReadLine();
        }
    }
}
```

Listing 9.12 uses the code from Listing 9.6 to get a list of Order objects from the Northwind database. The first LINQ query returns a subsequence that contains orders shipped to Mexico, and the second query returns orders shipped to New Mexico and Texas—ShipRegion equals “NM” or “TX”. The results are added together to produce sales orders by territory—perhaps the southwestern United States and Mexico.

LISTING 9.12 Orders Where ShipCountry Equals Mexico and ShipRegion Equals Texas and New Mexico

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data;
using System.Data.SqlClient;
using System.Reflection;
using System.Collections;

namespace UnionDemo2
{
    class Program
    {
        /// <summary>
        /// Demonstrates except ship city
        /// </summary>
        /// <param name="args"></param>
        static void Main(string[] args)
        {
            List<Order> orders = GetOrders();
            Console.WriteLine(orders.Count);

            var citiesInMexico = from order in orders
                                 where order.ShipCountry == "Mexico"
                                 select order;
            var citiesinNewMexico = from order in orders
                                   where order.ShipRegion == "NM"
                                   || order.ShipRegion == "TX"
                                   select order;

            var territory = citiesInMexico.Union(citiesinNewMexico);

            Array.ForEach(territory.ToArray(),
                input => Console.WriteLine(Dump(input)));
            Console.ReadLine();
        }
    }
}
```

Summary

Set operations are ubiquitous. Because so many programmers perform these tasks manually, it is easy to overlook how often we want some subset of what we already have. This chapter looked at `Union`, `Intersect`, `Except`, and `Distinct`. With these extension methods and LINQ, you might never again write nested loops scanning lists for elements in common or distinct. Sure, it's possible to perform some of these operations with SQL, but not all objects originate in a database, and it is here that set operations will really shine.

CHAPTER 10

Mastering Select and SelectMany

"Be kind, O Bacchus, take this empty pot offered to thee by Xenophon, the sot, Who, giving this, gives all that he has got."

—Eratosthenes

We have used `select` in many of the preceding chapters. This chapter focuses attention on `select` through the conduit of many examples. This chapter provides several samples that demonstrate `select` using things such as external application programming interfaces (APIs), `InteropServices`, and a discussion of `SelectMany`. `SelectMany` demonstrates how to get data from multiple sequences.

Of critical importance in this chapter is a demonstration of a constructive way to use `select` in the ubiquitous *business layer* that many n-tier applications contain.

Exploring Select

`Select` is both an extension method and a LINQ keyword. Programmers often use `select` in LINQ queries, but sometimes they want to know the index of items in a sequence. This section has several code examples that demonstrate both variations of `select`, including an example in Listing 10.4 that shows how to use the extension method form to add the indexing capability of the `select` function.

IN THIS CHAPTER

- ▶ Exploring `Select`
- ▶ Projecting New Types from Multiple Sources
- ▶ Creating a New Sequence from Multiple Sequences with `SelectMany`
- ▶ Using `SelectMany` with Indexes

Selecting with Function Call Effects

Keeping Secrets

On a plane trip to Seattle a couple of years ago, I met a math professor from McGill University in Quebec. He was on his way to a math symposium in New Orleans. Being an occasional chatty passenger on a long flight, I inquired about what was hot in mathematics these days. His answer: cryptography and prime numbers, basically keeping secrets. In this day and age, secrets seem to be important on every side of the political fence.

Having been an aspiring mathematician—but a poor scholar—I am familiar with a couple of interesting algorithms that can be used to calculate prime numbers and other related math algorithms. Two of them are the *sieve of Eratosthenes* for calculating prime numbers and the Euclidean algorithm for determining greatest common divisors (GCD). Listing 10.1 demonstrates both algorithms. The code demonstrates a brute force Prime calculator that indicates a number is prime if it has no prime factors—the GCD algorithm—and a slightly faster one that uses the “Sieve” algorithm. (See Figure 10.1 for the output from code.)

```
4801
4813
4817
4831
4861
4871
4877
4889
4903
4909
4919
4931
4933
4937
4943
4951
4957
4967
4969
4973
4987
4993
4999
Elapsed: Days: 0, Hours: 0, Minutes: 0, Seconds: 1, Mils: 416
```

FIGURE 10.1 Partial output of the primes from 2000 to 4999.

TIP

A number is prime if its divisors are itself and 1.

LISTING 10.1 Select with Function Call Effects

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using System.Text;
```

LISTING 10.1 Continued

```

namespace SelectDemo1
{
    class Program
    {
        static void Main(string[] args)
        {
            DateTime start = DateTime.Now;

            const long upper = 1000000;
            var numbers = new long[upper];
            for (long i = 2000; i < 5000; i++)
                numbers[i] = i+1;

            /* prohibitively slow */
            //var primes = from n in numbers
            //  where IsPrime(n)
            //  select n;

            // use Sieve of Eratosthenes; of course now we have primes
            BuildPrimes(upper);
            var primes = from n in numbers
                         where IsPrime2(n)
                         select n;
            DateTime stop = DateTime.Now;
            StringBuilder builder = new StringBuilder();
            Array.ForEach(primes.ToArray(), n=>builder.AppendFormat("{0}\n", n));
            Console.WriteLine(builder.ToString());
            Console.WriteLine("Elapsed: {0}", Elapsed(start, stop));
            Console.ReadLine();
        }

        /// <summary>
        /// Brute force prime tester, very slow.
        /// </summary>
        /// <param name="v"></param>
        /// <returns></returns>
        private static bool IsPrime(long v)
        {
            if (v <= 1) return false;
            for (long i = 1; i < v; i++)
                if (Gcd(i, v) > 1)
                    return false;
        }
    }
}

```

LISTING 10.1 Continued

```
        return true;
    }

/// <summary>
/// Use the Sieve of Eratosthenes: no number is divisible
/// by a number greater than its square root
/// </summary>
/// <param name="v"></param>
/// <returns></returns>
private static bool IsPrime2(long v)
{
    for(int i=0; i<Primes.Count; i++)
    {
        if(v % Primes[i] == 0) return false;
        if(Primes[i] >= Math.Sqrt(v)) return true;
    }

    return true;
}

private static List<long> Primes = new List<long>();
private static void BuildPrimes(long max)
{
    Primes.Add(2);
    if (max < 3) return;

    for (long i = 2; i <= max; i++)
    {
        if (IsPrime2(i))
            Primes.Add(i);
    }
}

/// <summary>
/// Recursive Euclidean algorithm
/// </summary>
/// <param name="num"></param>
/// <param name="den"></param>
/// <returns></returns>
private static long Gcd(long num, long den)
{
    return den % num == 1 ? 1 :
           den % num == 0 ? num : Gcd(den % num, num);
```

LISTING 10.1 Continued

```
}

private static string Elapsed(DateTime start, DateTime stop)
{
    TimeSpan span = stop - start;
    return string.Format("Days: {0}, Hours: {1}",
        "Minutes: {2}, Seconds: {3}, Mils: {4}",
        span.Days, span.Hours, span.Minutes, span.Seconds, span.Milliseconds);
}
}
```

In Listing 10.1, `IsPrime` or `IsPrime2` is called in the `where` clause returning only prime numbers in the sequence. `IsPrime` uses Euclid's algorithm, recursively testing all of the numbers from 1 to n as possible divisors. This version is obviously quite slow as n gets to be very large. (You could speed up `IsPrime` by only testing 1 to the square root of n , which is from Eratosthenes. This revision to `IsPrime` is left as an example.) The second version `IsPrime2` requires that you build all of the primes by testing a stored list of primes, storing the Primes in a `List<T>` as you check candidates, and you only test the primes up to those that are less than or equal to the square root of the candidate number. The `List<int>` of primes is seeded with the first Prime number 2.

As a useful general utility, the function `Elapsed` can be used to see how long it takes for a function to complete. If you run `IsPrime` up to a relatively small number like 1,000,000, the GCD approach is very slow. Using Eratosthenes' algorithm to calculate primes, the code runs much faster but still grinds to a halt after about 10,000,000 or so. In practice, cryptographers are looking for huge prime numbers that even supercomputers cannot crack using any brute force algorithm. That's why very large primes are useful in public key encryption; not because they can't be determined but because they can't be determined in any reasonable time frame.

The Sieve of Atkin

There is a more efficient algorithm, the sieve of Atkin, you can check on Wikipedia (or Google). The largest known prime is $2^{32,582,657}-1$. An interesting problem for PCs is how would such a number be expressed because the largest number that can be stored is a decimal number (at least in .NET) and the maximum string length is only 2 billion characters, which is considerably smaller than the number of digits needed to express the largest prime.

Manipulating Select Predicates

The `select` clause can be determined by calling functions. These predicates can also be manipulated directly. For example, if the `from` clause uses `num` in numbers, then `select` is not limited to using `num as its operand`. You can manipulate `num` in the `select` clause. Listing 10.2 uses a sequence of digits and multiplies each by 2 to yield a sequence of even numbers.

LISTING 10.2 Multiply Select Predicate

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace SelectDemo2
{
    class Program
    {
        static void Main(string[] args)
        {
            var numbers = new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
            var toEvens = from num in numbers
                         select num * 2;
            Array.ForEach(toEvens.ToArray(), n => Console.WriteLine(n));
            Console.ReadLine();
        }
    }
}
```

Returning Custom Business Objects from a Data Access Layer

Although manipulating simple numbers is useful, a more common problem might be how to use LINQ in the style of programming employed for business applications today. That is, how do you use LINQ in what is commonly referred to as a business layer? The answer is that you define the query to construct a known type instead of an anonymous type, and you define the functions in the business tier to return an `IEnumerable<T>` object where `T` is the business entity.

Listing 10.3 is a bit longer than the first two listings in this chapter, but this example need something that is practical as a business entity. In Listing 10.3, a `Supplier` class is defined using automatic properties and an overloaded `ToString` method for dumping entity state. An extension method is defined in `ExtendsReader` to extend `IDataReader`. This approach permits treating `Supplier` as self-initializing but you wouldn't necessarily have to convolute the entity class with ActiveX Data Objects (ADO) knowledge. `ReadSupplier` acts

like a member of `IDataReader` and a generic delegate `Func` is used to supply all of that tedious null checking.

Finally, a `DataAccess` class provides a general `ReadSuppliers` method that reads all suppliers (from Northwind) and a second `ReadSuppliers` that accepts a generic `Func` delegate to filter suppliers. The parameterless `ReadSuppliers` returns an `IEnumerable<Supplier>` and is pretty plain vanilla ADO.NET code. `ReadSuppliers` with the predicate uses the first parameterless `ReadSuppliers` and then uses a LINQ query with the `Where` extension method and the predicate `Func<Supplier, bool>` to filter the resultset. (Recall that Chapter 5, “Understanding Lambda Expressions and Closures,” introduced generic delegates.) In the example, the consumer provides the predicate. The predicate to `ReadSuppliers—s => s.Country == "USA"`—is provided in the main function of Listing 10.3.

LISTING 10.3 Using LINQ to Return Custom Business Objects

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data;
using System.Data.SqlClient;
using System.Reflection;

namespace SelectDemo3
{
    class Program
    {
        static void Main(string[] args)
        {
            IEnumerable<Supplier> USSuppliers =
                DataAccess.ReadSuppliers(s => s.Country == "USA");

            Array.ForEach(USSuppliers.ToArray(), s => Console.WriteLine(s));
            Console.ReadLine();
        }
    }

    public class Supplier
    {
        public int SupplierID { get; set; }
        public string CompanyName { get; set; }
        public string ContactName { get; set; }
        public string ContactTitle { get; set; }
        public string Address { get; set; }
        public string City { get; set; }
        public string Region { get; set; }
    }
}
```

LISTING 10.3 Continued

```
public string PostalCode{ get; set; }
public string Country{ get; set; }
public string Phone{ get; set; }
public string Fax{ get; set; }
public string HomePage { get; set; }

public override string ToString()
{
    StringBuilder builder = new StringBuilder();
    PropertyInfo[] props = this.GetType().GetProperties();

    // using array for each
    Array.ForEach(props.ToArray(), prop =>
        builder.AppendFormat("{0} : {1}", prop.Name,
            prop.GetValue(this, null) == null ? "<empty>\n" :
            prop.GetValue(this, null).ToString() + "\n"));

    return builder.ToString();
}

}

public static class DataAccess
{
    private static readonly string connectionString =
        "Data Source=.\SQLExpress;" +
        "AttachDbFilename=\"C:\\Books\\Sams\\\" +
        "LINQ\\Northwind\\northwnd.mdf\";Integrated Security=True;" +
        "Connect Timeout=30;User Instance=True";

    public static IEnumerable<Supplier>
        ReadSuppliers(Func<Supplier, bool> predicate)
    {
        IEnumerable<Supplier> suppliers = ReadSuppliers();
        return (from s in suppliers select s).Where(predicate);
    }

    //public static List<Supplier> ReadSupplier(Func<
    public static IEnumerable<Supplier> ReadSuppliers()
    {
        using(SqlConnection connection = new SqlConnection(connectionString))
        {
            connection.Open();
            SqlCommand command = new SqlCommand("SELECT * FROM SUPPLIERS", connection);

```

LISTING 10.3 Continued

```

IDataReader reader = command.ExecuteReader();

List<Supplier> list = new List<Supplier>();
while(reader.Read())
{
    list.Add(reader.ReadSupplier());
}

return list;
}
}

public static class ExtendsReader
{
    static Func<string, IDataReader, string, string> SafeRead =
        (fieldName, reader, defaultValue) =>
    reader[fieldName] != null ?
    (string)Convert.ChangeType(reader[fieldName], typeof(string)) :
    defaultValue;

    public static Supplier ReadSupplier(this IDataReader reader)
    {
        Supplier supplier = new Supplier();
        supplier.SupplierID = reader.GetInt32(0);
        supplier.CompanyName = SafeRead("CompanyName", reader, "");
        supplier.ContactName = SafeRead("ContactName", reader, "");
        supplier.ContactTitle = SafeRead("ContactTitle", reader, "");
        supplier.Address = SafeRead("Address", reader, "");
        supplier.City = SafeRead("City", reader, "");
        supplier.Region = SafeRead("Region", reader, "");
        supplier.PostalCode = SafeRead("PostalCode", reader, "");
        supplier.Country = SafeRead("Country", reader, "");
        supplier.Phone = SafeRead("Phone", reader, "");
        supplier.Fax = SafeRead("Fax", reader, "");
        supplier.HomePage = SafeRead("HomePage", reader, "");
        return supplier;
    }
}

```

Remember to adjust the connection string for your copy of the Northwind sample database. The results of the query are shown in Figure 10.2.

```

file:///C:/Books/Addison Wesley/LINQ/SOURCE/Chapter 10>SelectDemo3>SelectDemo3/bin/Debug/...
ContactName : Cheryl Sailor
ContactTitle : Regional Account Rep.
Address : 3400 - 8th Avenue Suite 210
City : Seattle
Region : WA
PostalCode : 97101
Country : USA
Phone : <503> 555-9931
Fax :
HomePage :

SupplierID : 19
CompanyName : New England Seafood Cannery
ContactName : Robb Merchant
ContactTitle : Wholesale Account Agent
Address : Order Processing Dept. 2100 Paul Revere Blvd.
City : Boston
Region : MA
PostalCode : 02134
Country : USA
Phone : <617> 555-3267
Fax : <617> 555-3389
HomePage :

```

FIGURE 10.2 The formatted U.S. suppliers sent to the console.

Using Select Indexes to Shuffle (or Unsort) an Array

The `Select` extension method has an implicit index variable. By using the `Select` extension method and expressing an input argument for the Lambda Expression, you can define a projection that includes the ordinal position of each element in a sequence. Listing 10.4 demonstrates the technique.

LISTING 10.4 Using the `Select` Method's Index to Create an Anonymous Type Containing an Index and a Random Number; Sort the Random Numbers and the Indexes Are Shuffled

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace SortingWithSelectIndex
{
    class Program
    {
        static void Main(string[] args)
        {
            int[] cards = new int[52];

            Random rand = new Random();

            var shuffler = cards.Select((num, index) =>
                new { Key = index, Random = rand.Next() });

            Array.ForEach(shuffler.OrderBy(s=>s.Random).ToArray(),
                s => Console.WriteLine(s));
        }
    }
}

```

LISTING 10.4 Continued

```
        Console.ReadLine();  
    }  
}  
}
```

The only drawback here is that you can't use the index in the LINQ version of the code. For example

```
var shuffler = from card in cards select new {Key = index, Random = rand.Next()};
```

is not a valid LINQ query. However, you can simulate the index using the LINQ form of the call by defining an integer before the query and incrementing the integer each time the `select` clause executes. This code has the same effect as the code in Listing 10.4:

```
int index = 0;
var result = from n in nums
    select new
    {
        value = n,
        key = index++
    };
}
```

You can expand on this idea of randomizing, unsorting, or shuffling an array to form the basis of a card game, an element of which is the chance of the draw.

Forming the Basis of a Card Game Like Blackjack

A couple of years ago, during a Blackjack craze, I wrote a Blackjack for Windows game. The game was designed to hint at statistically correct plays for all combinations, for example, what to do with a pair of 3s against a dealer 6. (It has helped my Blackjack game.) A short while after that, a woman from Harrah's casino in Biloxi asked permission to use it as a pillow favor—sort of a chocolate on the pillow—at the casino. The game was fun to write, and I was happy to share.

The game (refer to Figure 10.3) uses the `cards.dll` that is installed with Windows Solitaire and supports standard Blackjack play, including splitting, doubling-down, surrendering, holding, and hitting with hints that coach statistically perfect play.

Internally, the game combines Windows API calls to the `cards.dll`, combining `InteropServices` with `GDI+`. One small challenge was how to shuffle the cards. (There is no `randomize` array method in `.NET`, although for symmetry there should be an `unsort` behavior.) The solution: Define a structure with a random number and key. Increment the keys from 0 to 51 to represent the 52 cards, assign a random number to the `random` field, and by sorting the `random` field, scramble the cards. This actually takes a dozen lines of code or so.

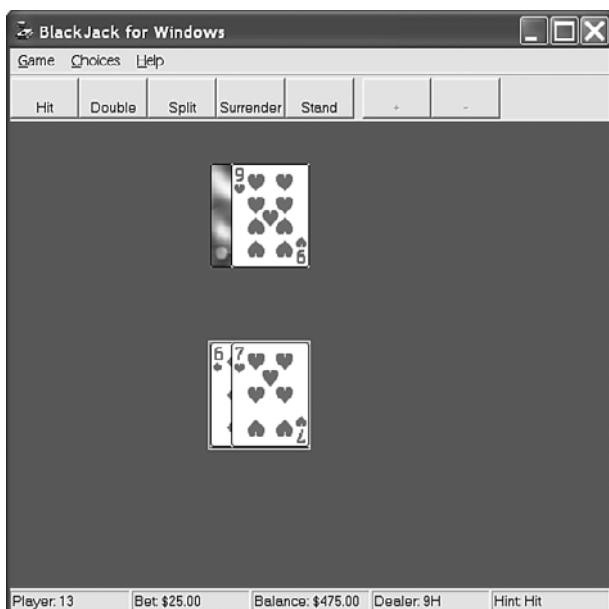


FIGURE 10.3 A Blackjack game for Windows written by the author in .NET.

The code in the rest of this section and its subsections demonstrates how to use LINQ to “shuffle” the cards, return a shuffled deck, and form the basis of a basic card game. The sample code in Listings 10.5, 10.6, and 10.7 contains code to import APIs from the `cards.dll`, use GDI+ with unmanaged API methods, draw cards, and shuffle a digital representation of a deck of cards. Let’s start with the `Main` function shown in Listing 10.5, which also contains all of the code together. (The subsections “Projecting New Types from Calculated Values,” “Importing DLLs,” and “Using GDI+ with the Windows API (or External DLL) Methods” elaborate on those subjects.)

A card is represented by a class of the same name. The class contains one each of the enumeration `Face` and `Suit`. There is also a `Shuffler` struct that is used to represent the ordered cards; sorting the random field unorders, or shuffles, the cards. For example, `shuffler.key` equal to 0 represents a card with the face of `One`, an ace, and the `Club` suit, 1 is the two of clubs, and so on.

In the `Main` function, an array of the `Shuffler` struct is created. The array contains 52 elements representing 52 cards. A simple loop is used to initialize the `Shuffler.key` to the values 0 through 51 and `Shuffler.random` to a random number. (It doesn’t matter if some of the random numbers are duplicated.) Next is a LINQ query ordering on `Shuffler.random` and projecting a new instance of `Card` for each element in the sequence. The actual `Card` objects `Suit` and `Face` are derived with arithmetic. `Suit` equals `key` divided by 13, the number of faces, and the `Face` is derived by `key` modulo 13. Every fourteenth card is in the next `Suit`.

LISTING 10.5 Shuffling (and Displaying) a Deck of Cards

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using System.Drawing;
using System.Runtime.InteropServices;
using System.Diagnostics;

namespace SortingCards
{
    class Program
    {
        public enum Face { One, Two, Three, Four, Five, Six, Seven,
            Eight, Nine, Ten, Jack, Queen, King };
        public enum Suit { Club, Diamond, Heart, Spade };

        private static int width;
        private static int height;

        public struct Shuffler
        {
            public int key;
            public int random;
        }

        public class Card
        {
            public Face Face { get; set; }
            public Suit Suit { get; set; }

            public override string ToString()
            {
                return string.Format("{0} of {1}s", Face, Suit);
            }
        }

        static void Main(string[] args)
        {
            const int MAX = 52;

            Random rand = new Random(DateTime.Now.Millisecond);
```

LISTING 10.5 Continued

```
Shuffler[] shuffler = new Shuffler[MAX];

for (int i = 0; i < MAX; i++)
{
    shuffler[i].key = i;
    shuffler[i].random = rand.Next();
}

// elem.key contains card number randomized by
// sorting on random numbers
var shuffledCards = from s in shuffler
    orderby s.random
    select new Card { Suit = (Suit)(s.key / 13), Face = (Face)(s.key % 13) };

cdtInit(ref width, ref height);
try
{

    using (Form form = new Form())
    {
        form.Show();

        int i = 0;
        foreach (var card in shuffledCards)
        {
            Graphics graphics = form.CreateGraphics();
            graphics.Clear(form.BackColor);
            string text = string.Format("Index: {0}, Card: {1}",
                i++, card);
            graphics.DrawString(text, form.Font, Brushes.Black, 10F,
                (float)(height + 20));
            PaintFace(graphics, card, 10, 10, width, height);
            form.Update();
            System.Threading.Thread.Sleep(500);
        }
    }
    finally
    {
        cdtTerm();
    }
}
```

LISTING 10.5 Continued

```

        Console.ReadLine();
    }

private static void PaintFace(Graphics g, Card card,
    int x, int y, int dx, int dy)
{
    IntPtr hdc = g.GetHdc();
    try
    {
        int intCard = (int)card.Face * 4 + (int)card.Suit;
        cdtDrawExt(hdc, x, y, dx, dy, intCard, 0, 0);
    }
    catch (Exception ex)
    {
        Debug.WriteLine(ex);
    }
    finally
    {
        g.ReleaseHdc(hdc);
    }
}

[DllImport("cards.dll")]
static extern bool cdtInit(ref int width, ref int height);

[DllImport("cards.dll")]
public static extern bool cdtDraw (IntPtr hdc, int x,
    int y, int card, int type, long color);

[DllImport("cards.dll")]
public static extern bool cdtDrawExt(IntPtr hdc, int x, int y, int dx,
    int dy, int card, int suit, long color);

[DllImport("cards.dll")]
    public static extern void cdtTerm();
}
}

```

At the end of Main, the cards library is initialized, a form is created, and each card is displayed for a half a second—`Thread.Sleep(500)`—in its new position. When all of the cards have been displayed, the cards library cleanup happens—`cdtTerm` is called. (Refer to Figure 10.4 for an example of the output to expect.)

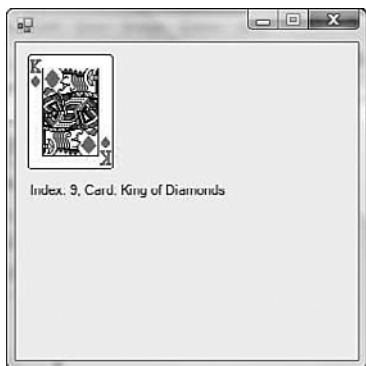


FIGURE 10.4 Displaying shuffled cards from cards.dll. (Vista users might need to grab a copy of this DLL from XP or the web.)

NOTE

The shuffle behavior could have been implemented with an anonymous type using the `Select` extension method with `index`. By assigning the index to a key field, the random number to a random field, and sorting on the random field, you get the same unshuffled representation of a deck too.

```
var shuffler = (new int[MAX]).Select((num, index) =>
    new { key = index, random = rand.Next() });
```

By attaching a call to the extension method `OrderBy` and a Lambda Expression on the `random` field, we could include the shuffled result in single statement.

Projecting New Types from Calculated Values

The LINQ query that starts with `var shuffledCards = from s in shuffler` demonstrates a projection to a specific, as opposed to anonymous, type. By using compound initialization with named parameters, you can construct `Card` objects specifying the card Suit and Face.

In practice, modify the code to permit initialization with the value available, `key`. In `key`'s setter in the `Card` class, add the logic to convert the `key` to the `Suit` and `Face`. This is just good form. Using a factory to create cards is an appropriate alternative, too.

Importing DLLs

Library methods are imported with the `DllImportAttribute` and the name of the external DLL. They are also prefixed with `static` and `extern` keywords. For example, Listing 10.5 imports `cdtInit`, `cdtDraw`, `cdtDrawExt`, and `cdtTerm` from `cards.dll`.

There are a couple of challenges when using external, nonmanaged API calls. One challenge is discerning correct usage. In this example, the `cards.dll` was designed to expect a cleanup call to `cdtTerm`. A second challenge is to get the declaration statement right.

When the types are straightforward as is the case in this example, the declarations are also straightforward. Things get muddied up when the API takes function pointers. Then, you need to use the `MarshalAsAttribute` defined in `System.Runtime.InteropServices` on the parameter.

Calls to external DLLs like `cards.dll` can use raw Device Context (DC) handles in unmanaged ways. GDI+ uses a managed handle. The next section looks at how to convert a managed DC (from GDI+) to an unmanaged DC for GDI-based or older API calls.

Using GDI+ with Windows API (or External DLL) Methods

GDI+ uses an uncached managed version of a device context, or DC. Device contexts are essentially graphics objects in GDI+. Older Windows API methods were designed to use a raw, unmanaged DC. Therefore, if you need to pass a device context to unmanaged library code, you need to do so safely. The method `PaintFace`, excerpted from Listing 10.5, shows how you can request the DC from a graphics object, pass the DC to an external, old-style API method, and clean up the code in a `finally` block with `Graphics.ReleaseHdc` (shown separately in Listing 10.6).

LISTING 10.6 Older-Style GDI API Methods Need a Raw Device Context

```
private static void PaintFace(Graphics g, Card card, int x,
    int y, int dx, int dy)
{
    IntPtr hdc = g.GetHdc();
    try
    {
        int intCard = (int)card.Face * 4 + (int)card.Suit;
        cdtDrawExt(hdc, x, y, dx, dy, intCard, 0, 0);
    }
    catch (Exception ex)
    {
        Debug.WriteLine(ex);
    }
    finally
    {
        g.ReleaseHdc(hdc);
    }
}
```

Using Select to I-Cap Words

Writing books is a little like mind reading or forecasting. Part of the job is to figure out things the reader might want to do that they might not know how to do already. With many different levels of readers, different levels of code samples are devised. Some are easy

and others are more challenging. The example demonstrates how you might convert the first letter of an array of words to an uppercase letter.

The literal string and subsequent splitting of that string is used here for convenience. The query starting with var icapped demonstrates how to capitalize the first character of a string.

LISTING 10.6 I-Capping Words in an Array

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ICappingSelectDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            // array of strings for convenience
            string wayfarism = "I'm still vertical";

            // create to array of strings
            var values = wayfarism.Split(new char[] { ' ' },
                StringSplitOptions.RemoveEmptyEntries);

            // icap
            var icapped = from s in values
                select s.Replace(s[0], Convert.ToChar(s[0].ToString().ToUpper()));

            icapped.WriteLineAll();
            Console.ReadLine();
        }
    }

    public static class Extender
    {
        public static void WriteLineAll<T>(this IEnumerable<T> ary)
        {
            Array.ForEach(ary.ToArray(), item=>Console.WriteLine(item));
        }
    }
}
```

Many other excellent resources are available on the web, including blogs, such as Bill Blogs in C# at <http://srtsolutions.com/blogs/billwagner/> and ScottGu's (Scott Guthrie) Blog at <http://weblogs.asp.net/scottgu/>, as well as excellent resources with general articles like my column *VB Today* at <http://www.codeguru.com>, InformIT.com, and 101 LINQ Samples by Microsoft at <http://msdn2.microsoft.com/en-us/vcsharp/aa336746.aspx>. You are encouraged to explore these resources and share their (free) availability with peers.

If you devise some clever queries, consider blogging about your own solutions or even submitting them to great sites like MSDN, InformIT, devsource.com, developer.com, or codeguru.com. Many of these sites compensate the author of original material.

Projecting New Types from Multiple Sources

When a LINQ query uses elements from one or more source types to define a new type, this is referred to as a projection. The compiler emits a class, part of whose name is `AnonymousType`. (Although you can use reflection to determine a projected type's name and use that name in code, it isn't a recommended practice.)

There are two key concepts for projecting new types: the `Select` feature and the `SelectMany`. Both `Select` and `SelectMany` are extension methods in the .NET Framework, but only `Select` is a LINQ keyword. The `SelectMany` method comes into play when you write queries that have multiple `from` clauses. Thus far, there have been many examples of `select`; Listing 10.7 demonstrates an implicit `SelectMany`. Listing 10.6 uses two classes: `Customer` and `Order`, and two collections containing instances of each. In the query that begins with `var orderInfo`, the query is selecting from customers and orders and correlating the objects on `customer.ID` and `order.CustomerID`. The presence of the clause `select new {Name = customer.CompanyName, Item=order.ItemDescription}` causes the compiler to emit a new class, an anonymous type referred to as a project.

LISTING 10.7 The Presence of Two `from` clauses in the LINQ Query Causes the Compiler to Emit a Call to `Enumerable.SelectMany` and the `select new` Clause Emits a New Anonymous Type Called a Projection

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data;
using System.Data.SqlClient;

namespace SelectManyDemo
{
    class Program
    {
```

LISTING 10.7 Continued

```

public class Customer
{
    public int ID{get; set;}
    public string CompanyName{get; set; }
}

public class Order
{
    public int ID{get; set; }
    public int CustomerID{get; set; }
    public string ItemDescription{get; set; }
}

static void Main(string[] args)
{
    List<Customer> customers = new List<Customer>{
        new Customer{ID=1, CompanyName="Tom's Toffees"},
        new Customer{ID=2, CompanyName="Karl's Coffees"}};

    List<Order> orders = new List<Order>{
        new Order{ID=1, CustomerID=1, ItemDescription="Granulated Sugar"},
        new Order{ID=2, CustomerID=1, ItemDescription="Molasses"},
        new Order{ID=3, CustomerID=2, ItemDescription="French Roast Beans"},
        new Order{ID=4, CustomerID=2, ItemDescription="Ceramic Cups"}};

    var orderInfo = from customer in customers
                    from order in orders
                    where customer.ID == order.CustomerID
                    select new {Name=customer.CompanyName,
                               Item=order.ItemDescription};

    Array.ForEach(orderInfo.ToArray(), o=>Console.WriteLine(
        "Company: {0}, Item: {1}", o.Name, o.Item));
    Console.ReadLine();
}
}
}

```

Listing 10.7 has `Customer` and `Order` classes. A generic `List<T>` is instantiated for each. The LINQ query selects from each of the lists of customers and orders and joins on the customer ID with a `where` clause. The `select` statement projects a new type containing a customer name assigned to a `Name` property and the item description assigned to an `Item` property to create a new anonymous type.

The presence of more than one `from` clause is technically taking two sequences and converting them into a single sequence. This is the equivalent of a SQL join. (Recall that in SQL, you can join with the literal join or by using `where` predicates; the same is true for LINQ.)

The capability of LINQ to project new types—also called data shaping—is powerful. LINQ queries enable you to quickly and easily come up with brand-new types that previously hadn't been defined in code, and mitigate the need to hand code classes for every possible combination of data you might need.

Creating a New Sequence from Multiple Sequences with `SelectMany`

Phishing is what people are trying to do when they send out an email or phony link to try to get usernames and passwords from unsuspecting people. Excessive pop-ups, Trojans, worms, and viruses are all other kinds of despicable things that people do with computers. In some very small way, these programs challenge application and operating system writers to do a better job, so in some way people who write these obnoxious programs are useful (as much as a flesh-eating bacteria is useful).

One thing many of these harmful bits of code have in common is that they attack the Windows Registry. So, as a computer programmer and a computer user, it is helpful to understand how the Registry works. In addition, it is helpful to understand how the Registry works as a general programming aid—for when you need to add useful information to the Registry.

TIP

.NET introduced code access security (CAS). One of the things code access security does is permit assigning permissions to code. That is, CAS permissions can prohibit code from specific sources from doing things. For example, code access security can prohibit downloaded code from accessing the Registry.

The Windows Registry is a big dictionary (or database) of sorts containing hierarchical name and value pairs. .NET makes accessing the Registry for reads and writes pretty straightforward. With LINQ, you can actually write queries to scan and explore the Registry quite easily.

Consider the following scenario. Your program uses Windows event logging to write application exceptions to the event log. To work when installed, it needs to create an event source in the Registry at `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\Eventlog\`.

.NET provides the `System.Diagnostics.EventLog` class and `CreateEventSource` for this purpose. But, what if you want to validate that the key was created? You'd need to access the Registry and look for the key. A failure to create the key might be due to a lack of code access permissions, but you wouldn't know unless a check was performed. To solve the problem, you could write a query that searched for the required key after the installer was to have created it. If the key is not found, then the installed application won't work.

Another scenario might be to scan the Registry looking for differences. For example, you might want to look for keys on one part of the Registry that should exist in another part, or vice versa. The following code (see Listing 10.8) compares two sections of the Registry—`LocalMachine` and `CurrentUser`—and uses an implied `SelectMany` with two `from` and two `where` clauses to find keys in common.

LISTING 10.8 Comparing Two Sections of the Registry—`LocalMachine` and `CurrentUser`—Looking for Keys in Common Using an Implicit `SelectMany` with Two `from` and Two `where` Clauses

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Microsoft.Win32;

namespace SelectingRegistryKeys
{
    class Program
    {
        static void Main(string[] args)
        {
            var localMachineKeys = Registry.LocalMachine.OpenSubKey("Software").GetSub-
KeyNames();
            var userKeys = Registry.CurrentUser.OpenSubKey("Software").GetSubKeyNames();

            // keys in common
            var commonKeys = from machineKey in localMachineKeys
                            where machineKey.StartsWith("A")
                            from userKey in userKeys
                            where userKey.StartsWith("A") &&
                            /*where*/ machineKey == userKey
                            select machineKey;

            Array.ForEach(commonKeys.ToArray(), key => Console.WriteLine(key));
            Console.ReadLine();
        }
    }
}
```

Using set operations as described in Chapter 9, “Performing Set Operations,” you could use extension methods like `Distinct`, `Except`, or `Intersect` to find similarities or differences in the keys in the various Registry sections. (This is left as an exercise for the reader.)

Using `SelectMany` with Indexes

`SelectMany` is an extension method and is supported in LINQ with the keyword `select` and multiple source sequences. Like the `Select` extension method, the `SelectMany` extension method supports the generic delegate `Func` and `indexes`.

This chapter’s final example reuses some of the code from Listing 10.7 and an explicit `SelectMany` to assign a random customer number to the customers. You might want to do this if you don’t want to reveal information useful to hackers, such as the actual primary key IDs in your database’s schema.

LISTING 10.9 Using the Extension Method `SelectMany` and Its Ability to Use an Implicit Index to Assign a Sequential Number to an Object in a Sequence (Rather Than Using the Primary Key Field)

```
using System;
using System.Linq;
using System.Collections.Generic;

namespace SelectManyWithIndex
{
    class Program
    {

        public class Customer
        {
            public int ID { get; set; }
            public string CompanyName { get; set; }
        }

        public class Order
        {
            public int ID { get; set; }
            public int CustomerID { get; set; }
            public string ItemDescription { get; set; }
        }

        static void Main(string[] args)
        {
            List<Customer> customers = new List<Customer>{
                new Customer{ID=1, CompanyName="Tom's Toffees"},
                new Customer{ID=2, CompanyName="Karl's Coffees"};
```

LISTING 10.9 Continued

```

List<Order> orders = new List<Order>{
    new Order{ID=1, CustomerID=1, ItemDescription="Granulated Sugar"},
    new Order{ID=2, CustomerID=1, ItemDescription="Molasses"},
    new Order{ID=3, CustomerID=2, ItemDescription="French Roast Beans"},
    new Order{ID=4, CustomerID=2, ItemDescription="Ceramic Cups"}};

var orderInfo =
    customers.SelectMany(
        (customer, index) =>
            from order in orders
            where order.CustomerID == customer.ID
            select new { Key = index + 1, Customer = customer.CompanyName,
Item = order.ItemDescription });

Array.ForEach(orderInfo.ToArray(), o=>Console.WriteLine(
    "Key: {0}, Name: {1}, Item: {2}", o.Key, o.Customer, o.Item));
Console.ReadLine();
}
}
}

```

The code in Listing 10.9 uses this overloaded version of `SelectMany`:

```

public static IEnumerable<TResult> SelectMany<TSource, TResult>(
    this IEnumerable<TSource> source,
    Func<TSource, int, IEnumerable<TResult>> selector

```

The `static` qualifier and `this` for the first parameter `IEnumerable<TSource>` indicates that the method is an extension method for `IEnumerable<T>`. The argument represented by `this` is actually the instance argument `customers` in the listing. The second parameter beginning with the generic delegate `Func` indicates that the first generic parameter is the input customer and the second is an index. The Lambda Expression

```
(customer, index) => from order in orders where order.CustomerID
    == customer.ID select new { Key = index + 1, Customer = customer.CompanyName,
Item = order.ItemDescription }
```

satisfies the generic delegate `Func` and the expression body—the LINQ query—returns the result type `IEnumerable<TResult>` of the generic delegate.

It is important not to get behind on advanced subjects like generics because languages build complexity in layers. The very cryptic looking `SelectMany` overloaded method is a perfect (and somewhat intimidating) example of that layering of capabilities.

Summary

You can't write a LINQ query without using `select`. So, you have seen many `select` statements in previous chapters, and you will see many more in the remaining chapters. But, the purpose of this book is to zoom our microscope in on a specific branch of the .NET Framework.

This chapter focused on exploring several examples using `select` and `select many`. You now have most of the basis and underpinnings mastered, so this part wraps up with an exploration of joins and exploring external objects models. Parts III and IV look at how LINQ works with ADO.NET and Extensible Markup Language (XML).

This page intentionally left blank

CHAPTER 11

Joining Query Results

"I'm still vertical."

—Jackson Wayfare

If you have experience with SQL joins, the basics of LINQ joining are pretty straightforward. If you have no SQL experience, this chapter covers the basics and more advanced join queries. This chapter describes when the literal keyword `join` can be used to define a join and when where predicates are used instead. Inner joins, left joins, cross joins, group joins, custom joins, and joins based on composite keys are all covered.

Using Multiple From Clauses

The presence of multiple `from` clauses in a LINQ query does not necessarily represent a join. A correlation is required. A correlation is where the query indicates how one sequence is related to another. This can be done in the `join` clause with the `equals` keyword, an inequality expression, or multiple expressions. Listing 11.1 contains an example that has multiple `from` clauses but no correlation between the sequence in the first `from` and second `from` clause.

LISTING 11.1 Multiple from Clauses But No Correlated Join

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
```

IN THIS CHAPTER

- ▶ Using Multiple From Clauses
- ▶ Defining Inner Joins
- ▶ Using Custom, or Nonequijoins
- ▶ Implementing Group Join and Left Outer Join
- ▶ Implementing a Cross Join
- ▶ Defining Joins Based on Composite Keys

LISTING 11.1 Continued

```

namespace MultipleFroms
{
    class Program
    {
        static void Main(string[] args)
        {

            string[] willRogers =
            {
                "Don't gamble; take all your savings ",
                "and buy some good stock and hold it ",
                "till it goes up, then sell it. If it ",
                "don't go up, don't buy it. Will Rogers"
            };

            // part, word, words, and one are all called range variables
            var horseSense = from part in willRogers
                            let words = part.Split(';', '.', ' ', ',')
                            from word in words
                            let one = word.ToUpper()
                            where one.Contains('I')
                            select word;

            Array.ForEach(horseSense.ToArray(), w => Console.WriteLine(w));
            Console.ReadLine();
        }
    }
}

```

The quotation attributed to Will Rogers (`willRogers`) is an array of strings. The query contains two `from` clauses, one from the array of strings and the second from the `let` variable `words`. The query selects each of the strings from the original quote. Each string is split and assigned to `words`. The second `from` clause selects each word in the `words`, converts them to uppercase, and selects those that contain `I`.

In the clause “`from part in willRogers`”, `part` is called the *range*. The range variable is like the iteration variable in a `for` loop. The keyword `let` is also used to define a range. Thus, in the query in Listing 11.1, there are four range variables: `part`, `words`, `word`, and `one`. LINQ infers the type of range variables from the context.

Range variables are a limiting factor that defines when you can use `join` and when you can’t. This concept is covered in more detail in the section titled “Using Custom, or `Nonequijoins`.”

Defining Inner Joins

Given two sources of data (this chapter refers to one as the left source and the other as the right source), an inner join is a join such that only objects that have matching criteria in both the left and right side are matched. In LINQ, this literally translates to all the items in the first `from` clause are returned only if there are items in the second `from` clause. The number of items in the first and second sequences does not have to be equal, but all of the items that have matching criteria in both sequences are returned to form a new sequence.

Inner joins are generally based on what is commonly referred to as master-detail or parent-child relationships. Generally, there is one item in the master source for every one or more items in the detail source. Certain things can happen to the data, like children without parents—called orphans—but this doesn’t change the way you request an inner join sequence. Listing 11.2 demonstrates an inner join based on a contrived data set.

LISTING 11.2 A Typical Inner Join Based on a Single “Key,” the Customer Object’s ID.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace JoinDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            List<Customer> customers = new List<Customer>{
                new Customer{ID=1, CompanyName="Tom's Toffees"},
                new Customer{ID=2, CompanyName="Karl's Coffees"},
                new Customer{ID=3, CompanyName="Mary's Marshmallow Cremes"}};

            List<Order> orders = new List<Order>{
                new Order{ID=1, CustomerID=1, ItemDescription="Granulated Sugar"},
                new Order{ID=2, CustomerID=1, ItemDescription="Molasses"},
                new Order{ID=3, CustomerID=2, ItemDescription="French Roast Beans"},
                new Order{ID=4, CustomerID=2, ItemDescription="Ceramic Cups"}};

            // join
            var join1 = from customer in customers
                       join order in orders on customer.ID equals
                           order.CustomerID
                       select new {Name=customer.CompanyName,
                           Item=order.ItemDescription};
        }
    }
}
```

LISTING 11.2 Continued

```

        Array.ForEach(join1.ToArray(), o=>Console.WriteLine
            ("Name: {0}, Item: {1}", o.Name, o.Item));
        Console.ReadLine();
    }
}

public class Customer
{
    public int ID{get; set;}
    public string CompanyName{get; set; }
}

public class Order
{
    public int ID{get; set; }
    public int CustomerID{get; set; }
    public string ItemDescription{get; set; }
}
}

```

In Listing 11.2, the customers and orders are related based on the customer ID in both the Customer and Order classes. Tom's Toffees is used twice because it appears once in the customers sequence and is associated with two orders. Mary's Marshmallow Cremes does not show up in the resultset because Mary's has no orders in the data.

NOTE

One style you can use is `ID` for the primary key when it makes sense and `tablenameID` for the matching foreign key in related tables. This causes pairings like `customer.ID` and `order.CustomerID`. This style works for some, but might not be the way you design logical schemas. One of the most important things is consistency, so if your way works for you, use it.

The query in Listing 11.2 is called an equijoin. This query is testing for equality, so it uses the `join` keyword. Notice that the LINQ query's `on` predicate uses the keyword `equals` instead of the operator `==`. You can write custom joins based on inequality, called nonequijoins, but you can't use the keyword `join` for inequality tests.

Using Custom, or Nonequijoins

A literal `join` clause performs an equijoin using the `join`, `on`, and `equals` keywords. You can perform nonequijoins for cross joins and customs joins without the `join` keyword when the `join` is predicated on inequality, multiple expressions of equality or inequality,

or when a temporary range variable using `let` is introduced for the right-side sequence. To define a custom join, define the correlation between sequences in a `where` clause.

Defining a Nonequal Custom Join

If there are two sequences containing keys, equijoins based on those keys are straightforward. However, if the second sequence is part of search criteria or input from a user, the second sequence would not be composed of objects and, hence, there would be no keys to join on. In this scenario, you could use a custom join and a `where` predicate.

Listing 11.3 defines two extension methods for `IDataReader`. The extension methods read and initialize a collection of products and suppliers. The `Product` and `Supplier` code is boilerplate code representing the Northwind Products and Suppliers tables. The classes `Product` and `Supplier` are entity classes.

In Listing 11.3, `productIDsWanted` represents the selection criteria and the LINQ query uses a range variable `ID` to determine which `Product` objects to return. The `SupplierID` and the `ProductName` are displayed to the console using the `Array.ForEach` method and the `Action<T>` delegate.

LISTING 11.3 Custom Join Based on Selection Criteria Sequence

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.IO;
using System.Data;
using System.Data.SqlClient;

namespace CustomJoin
{

    class Program
    {

        static void Main()
        {

            var productIDsWanted = new int[] { 1, 2, 3, 4, 5, 6 };

            List<Product> products = GetProducts();

            var results =
                from p in products
                from id in productIDsWanted
                where p.SupplierID == id
```

LISTING 11.3 Continued

```
orderby p.SupplierID
select p;

Array.ForEach(results.ToArray(), target => Console.WriteLine(
    "Supplier ID={0}, Product={1}",
    target.SupplierID, target.ProductName
));
Console.ReadLine();

}

private static readonly string connectionString =
    "Data Source=.\SQLExpress;AttachDbFilename="" +
    "C:\Books\Sams\LINQ\Northwind\northwnd.mdf"" +
    "Integrated Security=True;Connect Timeout=30;User Instance=True";

private static List<Product> GetProducts()
{
    const string SQL = "SELECT * FROM PRODUCTS";
    using(SqlConnection connection = new SqlConnection(connectionString))
    {
        connection.Open();
        SqlCommand command = new SqlCommand(SQL, connection);
        IDataReader reader = command.ExecuteReader();
        return reader.ReadProducts();
    }
}

private static List<Supplier> GetSuppliers()
{
    const string SQL = "SELECT * FROM Suppliers";
    List<Supplier> suppliers = new List<Supplier>();

    using (SqlConnection connection = new SqlConnection(connectionString))
    {
        connection.Open();
        SqlCommand command = new SqlCommand(SQL, connection);
        IDataReader reader = command.ExecuteReader();
        return reader.ReadSuppliers();
    }
}
```

LISTING 11.3 Continued

```
public class Product
{
    public int? ProductID{ get; set; }
    public string ProductName{ get; set; }
    public int? SupplierID{ get; set; }
    public int? CategoryID{ get; set; }
    public string QuantityPerUnit{ get; set; }
    public decimal? UnitPrice{ get; set; }
    public int? UnitsInStock{ get; set; }
    public int? UnitsOnOrder{ get; set; }
    public int? ReorderLevel{ get; set; }
    public bool? Discontinued{ get; set; }
}

public class Supplier
{
    public int? SupplierID{ get; set; }
    public string CompanyName{ get; set; }
    public string ContactName{ get; set; }
    public string ContactTitle{ get; set; }
    public string Address{ get; set; }
    public string City{ get; set; }
    public string Region{ get; set; }
    public string PostalCode{ get; set; }
    public string Country{ get; set; }
    public string Phone{ get; set; }
    public string Fax{ get; set; }
    public string HomePage{ get; set; }
}

public static class ExtendsReader
{
    public static List<Product> ReadProducts(this IDataReader reader)
    {
        List<Product> list = new List<Product>();
        while (reader.Read())
        {
            Product o = new Product();
            o.ProductID = reader.GetInt32(0);
            o.ProductName = reader.GetString(1);
            o.SupplierID = reader.GetInt32(2);
            o.CategoryID = reader.GetInt32(3);
            o.QuantityPerUnit = reader.GetString(4);
            o.UnitPrice = reader.GetDecimal(5);
            o.UnitsInStock = reader.GetInt16(6);
        }
    }
}
```

LISTING 11.3 Continued

```
    o.UnitsOnOrder = reader.GetInt16(7);
    o.ReorderLevel = reader.GetInt16(8);
    o.Discontinued = reader.GetBoolean(9);
    list.Add(o);
}

return list;
}

public static List<Supplier> ReadSuppliers(this IDataReader reader)
{
    List<Supplier> list = new List<Supplier>();
    while(reader.Read())
    {
        Supplier o = new Supplier();
        o.SupplierID = reader.GetInt32(0);
        o.CompanyName = reader.GetString(1);
        o.ContactName = reader.GetString(2);
        o.ContactTitle = reader.GetString(3);
        o.Address = reader.GetString(4);
        o.City = reader.GetString(5);
        o.Region = reader["Region"] ==
            System.DBNull.Value ? "" : reader.GetString(6);
        o.PostalCode = reader.GetString(7);
        o.Country = reader.GetString(8);
        o.Phone = reader.GetString(9);
        o.Fax = reader["Fax"] ==
            System.DBNull.Value ? "" : reader.GetString(10);
        o.HomePage = reader["HomePage"] == System.DBNull.Value ?
            "" : reader.GetString(11);

        list.Add(o);
    }

    return list;
}
}
```

Notice the use of the formatting sequence on the `Console.WriteLine` statement `{0,4}`. You can use additional formatting input to manage the layout or formatting. `Console.WriteLine` has an overloaded version that invokes `string.Format` and `{0,4}` pads the output for format argument 0 to a width of 4, left-justifying the output.

Defining a Custom Join with Multiple Predicates

If there are multiple selection criteria, you can add additional predicates to the `where` clause of a custom join. Listing 11.4 shows a revision to the LINQ query from Listing 11.3. The new query further refines the selection criteria to those elements between 2 and 5.

LISTING 11.4 Refined Custom Join Based on the Selection Criteria Sequence and Additional Predicates in the `where` Clause

```
var results =
    from p in products
    from id in productIDsWanted
    where p.SupplierID == id
    && id > 2 && id < 5
    orderby p.SupplierID
    select p;
```

Suppose you want the actual supplier objects and the product objects to get the name of the supplier as well as the name of the product. You could add an equijoin on supplier and product (you would need to obtain a list of suppliers with the `GetSuppliers` method) and a custom join on the sequence of integers. The `Main` function in Listing 11.5 can be used to replace `Main` in Listing 11.3. The revised `Main` function in Listing 11.5 displays the company name of the supplier instead of the ID and the product name.

LISTING 11.5 A Revision to `Main` in Listing 11.3 That Displays the Supplier's Name Instead of the ID and the Product Name

```
static void Main()
{
    var productIDsWanted = new int[] { 1, 2, 3, 4, 5, 6 };

    List<Product> products = GetProducts();
    List<Supplier> suppliers = GetSuppliers();

    var results =
        from p in products
        join s in suppliers on p.SupplierID equals s.SupplierID
        from id in productIDsWanted
        where p.SupplierID == id
        && id > 2 && id < 5
        orderby p.SupplierID
        select new { SupplierName = s.CompanyName, ProductName = p.ProductName };

    Array.ForEach(results.ToArray(), target => Console.WriteLine(
        "Supplier ID={0}, Product={1}",
```

LISTING 11.5 Continued

```

        target.SupplierName, target.ProductName
    ));
Console.ReadLine();
}

```

Defining Custom Joins with a Temporary Range Variable

As mentioned earlier in the chapter, the variable introduced in the `from` clause before the `in` keyword is referred to as a range variable. It plays the same role as the iterator variable in a `for` loop. You can introduce additional range variables using the `let` keyword. If you introduce a range variable for the right side of an inner join, you can't use the `join on equals` clause; instead, you have to use a custom join with a `where` clause and a predicate.

Listing 11.6 is a variation of Listing 11.3. (That's why the entire listing is here.) Listing 11.6 shows how you can combine multiple queries in a single SQL statement used to initialize the `SqlCommand` object. `IDataReader.Read` iterates through all of the elements in a single resultset and `IDataReader.NextResult` iterates over the resultsets. Because the `IDataReader` is forward only, you want to get all of the elements from each resultset before moving to the next resultset, as demonstrated in Listing 11.6.

LISTING 11.6 Custom Join with Temporary Range Variable Using `let`

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.IO;
using System.Data;
using System.Data.SqlClient;

namespace NonEquijoin
{
    class Program
    {
        static void Main()
        {
            const string connectionString =
                // "Data Source=BUTLER;Initial Catalog=Northwind;Integrated Security=True";

                "Data Source=.\SQLEXPRESS;AttachDbFilename=\" \" +
                "C:\Books\Sams\LINQ\Northwind\northwnd.mdf\";" +
                "Integrated Security=True;Connect Timeout=30;User Instance=True";
        }
    }
}

```

LISTING 11.6 Continued

```
const string SQL = "SELECT * FROM PRODUCTS; SELECT * FROM SUPPLIERS";
List<Product> products = new List<Product>();
List<Supplier> suppliers = new List<Supplier>();

using(SqlConnection connection = new SqlConnection(connectionString))
{
    connection.Open();
    SqlCommand command = new SqlCommand(SQL, connection);
    IDataReader reader = command.ExecuteReader();
    products = reader.ReadProducts();

    if(reader.NextResult())
        suppliers = reader.ReadSuppliers();

}

var nonEquiJoin =
    from p in products
    let master = from s in suppliers select s.SupplierID
    where master.Contains(p.SupplierID)
    orderby p.SupplierID
    select new {SupplierID=p.SupplierID, ProductName = p.ProductName};

Array.ForEach(nonEquiJoin.ToArray(), target=>Console.WriteLine(
    "Supplier ID={0}, Product={1}",
    target.SupplierID, target.ProductName
));
Console.ReadLine();
}

public class Product
{
    public int? ProductID{ get; set; }
    public string ProductName{ get; set; }
    public int? SupplierID{ get; set; }
    public int? CategoryID{ get; set; }
    public string QuantityPerUnit{ get; set; }
    public decimal? UnitPrice{ get; set; }
    public int? UnitsInStock{ get; set; }
    public int? UnitsOnOrder{ get; set; }
    public int? ReorderLevel{ get; set; }
    public bool? Discontinued{ get; set; }
}
```

LISTING 11.6 Continued

```
public class Supplier
{
    public int? SupplierID{ get; set; }
    public string CompanyName{ get; set; }
    public string ContactName{ get; set; }
    public string ContactTitle{ get; set; }
    public string Address{ get; set; }
    public string City{ get; set; }
    public string Region{ get; set; }
    public string PostalCode{ get; set; }
    public string Country{ get; set; }
    public string Phone{ get; set; }
    public string Fax{ get; set; }
    public string HomePage{ get; set; }
}

public static class ExtendsReader
{
    public static List<Product> ReadProducts(this IDataReader reader)
    {

        List<Product> list = new List<Product>();
        while (reader.Read())
        {
            Product o = new Product();
            o.ProductID = reader.GetInt32(0);
            o.ProductName = reader.GetString(1);
            o.SupplierID = reader.GetInt32(2);
            o.CategoryID = reader.GetInt32(3);
            o.QuantityPerUnit = reader.GetString(4);
            o.UnitPrice = reader.GetDecimal(5);
            o.UnitsInStock = reader.GetInt16(6);
            o.UnitsOnOrder = reader.GetInt16(7);
            o.ReorderLevel = reader.GetInt16(8);
            o.Discontinued = reader.GetBoolean(9);
            list.Add(o);
        }

        return list;
    }

    public static List<Supplier> ReadSuppliers(this IDataReader reader)
    {
        List<Supplier> list = new List<Supplier>();
        while(reader.Read())
        {

```

LISTING 11.6 Continued

```
Supplier o = new Supplier();
o.SupplierID = reader.GetInt32(0);
o.CompanyName = reader.GetString(1);
o.ContactName = reader.GetString(2);
o.ContactTitle = reader.GetString(3);
o.Address = reader.GetString(4);
o.City = reader.GetString(5);
o.Region = reader["Region"] ==
    System.DBNull.Value ? "" : reader.GetString(6);
o.PostalCode = reader.GetString(7);
o.Country = reader.GetString(8);
o.Phone = reader.GetString(9);
o.Fax = reader["Fax"] ==
    System.DBNull.Value ? "" : reader.GetString(10);
o.HomePage = reader["HomePage"] == System.DBNull.Value ?
    "" : reader.GetString(11);

list.Add(o);
}

return list;
}
}
```

NOTE

Unfortunately, the ability to introduce additional SQL statements as part of the command string for a `SQLCommand` is the same capability that allows SQL injection to work against an application's integrity. For this reason, rather than passing user data right to SQL commands and using SQL code in application code, you might prefer to use stored procedures.

You might also use stored procedures—sometimes called *sprocs*—for other reasons. For example, a stored procedure is an excellent place to partition work among multiple developers. Highly experienced SQL developers can write the SQL code in parallel with highly skilled C# (or VB .NET) programmers writing .NET code. If both of those people are the same person, then, aside from being talented and blessed, using stored procedures is a great way to facilitate intensity of task focus.

The LINQ query in Listing 11.6 uses the range variable `master` defined from an additional LINQ query that returns a sequence of just `SupplierIDs`. Because the listing uses `let`, a `where` clause implements the implied custom join, returning only those products that contain the predicated `SupplierID`. The results are ordered by the `SupplierID`.

Implementing Group Join and Left Outer Join

The group join and the left outer join use a group join but produce different resultsets. A group join is emitted to Microsoft Intermediate Language (MSIL, managed code's assembly language output) as an extension method `GroupJoin` when the `into` clause is used in a LINQ query. The group is represented as a master-detail relationship with a single instance of elements from the master sequence and subordinate, related sequences representing the details (or children). A left outer join uses the `GroupJoin` method (and `into` clause) but the data is flattened back out, denormalizing elements from both sequences. Unlike the inner join, however, you still get master sequence elements that have no child, detail sequences.

For example, if you perform group join on customers and orders from Listing 11.2, you get one result for Tom's, one for Karl's, and one for Mary's. Tom's contains a child sequence containing Granulated Sugar and Molasses, Karl's contains a sequence containing French Roast Beans and Ceramic Cups, and Mary's has no child sequence. If you convert the group to a left join, the customer data reappears each time for every child element and Mary's appears once with an empty child element.

The next subsection demonstrates the group join and the following subsection demonstrates the revision that produces the flattened left join.

Defining a Group Join

SQL experts know what groups do. As a reminder for the rest of us, a group is a way of returning items in a first sequence and correlating each of the items with the related elements from the second sequence. For example, in Listing 11.2, Tom's Toffees sells Granulated Sugar and Molasses. With an inner join, Tom's Toffees is returned and repeated for each of the items Tom sells. If you use a group join, Tom's Toffees is returned one time and is correlated with the items he sells. A group join, in effect, is a way to represent the normalized relationship existing in relational databases, and an inner join denormalizes (or repeats) data.

With the group join, the data is handled differently. An outer sequence represents the groups and an inner sequence in each group element represents the elements of the group. The result is a logical, master-detail relationship represented by the return type. In technical reality, you still have an `IEnumerable<T>` return result from group join queries, but each group is associated with a subordinate sequence related by a key value.

Listing 11.7 shows a straightforward use of the group join. It is the presence of the `into` keyword and predicate that instructs the compiler to emit the call to the `GroupJoin` extension method (see Figure 11.1).

LISTING 11.7 Defining a Query That Uses the GroupJoin Behavior, Capturing the Master-Detail Relationship Between Customers and Orders

```
using System;
using System.Collections.Generic;
using System.Linq;
```

LISTING 11.7 Continued

11

```
using System.Text;

namespace GroupJoin
{
    class Program
    {
        static void Main(string[] args)
        {
            List<Customer> customers = new List<Customer>{
                new Customer{ID=1, CompanyName="Tom's Toffees"},
                new Customer{ID=2, CompanyName="Karl's Coffees"},
                new Customer{ID=3, CompanyName="Mary's Marshmallow Cremes"}};

            List<Order> orders = new List<Order>{
                new Order{ID=1, CustomerID=1, ItemDescription="Granulated Sugar"},
                new Order{ID=2, CustomerID=1, ItemDescription="Molasses"},
                new Order{ID=3, CustomerID=2, ItemDescription="French Roast Beans"},
                new Order{ID=4, CustomerID=2, ItemDescription="Ceramic Cups"}};

            // group join
            var group = from customer in customers
                        join order in orders on customer.ID equals
                        order.CustomerID into children
                        select new { Customer = customer.CompanyName,
                                    Orders = children };

            string separator = new string('-', 40);
            foreach (var customer in group)
            {
                Console.WriteLine("Customer: {0}", customer.Customer);
                Console.WriteLine(separator);
                foreach (var order in customer.Orders)
                {
                    Console.WriteLine("\tID={0}, Item={1}", order.ID, order.ItemDescription);
                }
                Console.WriteLine(Environment.NewLine);
            }

            Console.ReadLine();
        }
    }

    public class Customer
    {
        public int ID{get; set;}
        public string CompanyName{get; set; }
```

LISTING 11.7 Continued

```

}
public class Order
{
    public int ID{get; set; }
    public int CustomerID{get; set; }
    public string ItemDescription{get; set; }
}
}

```

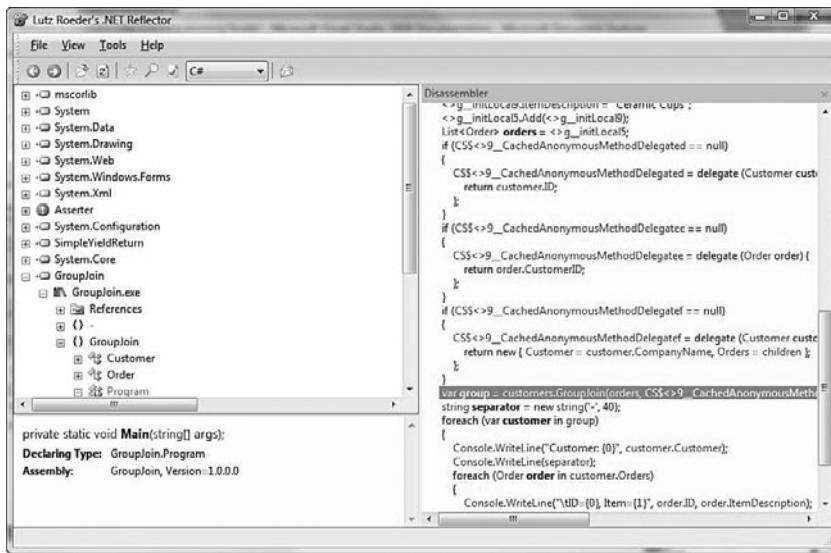


FIGURE 11.1 Lutz Roeder's tool Reflector showing the disassembled GroupJoin sample with the emitted GroupJoin extension method emitted for the LINQ query.

The way the query is designed in this example provides a master-detail relationship, which includes Mary's Marshmallow Cremes even though there are no orders for Mary's. Note the use of the nested `for` loop to get at the child sequence data. The output from Listing 11.7 is shown in Figure 11.2.

Implementing a Left Outer Join

A left outer join is an inner join plus all of the elements in the first range that don't have related elements in the second range, so a left join is always greater than or equal to the size of an inner join.

A left joins starts out as a group join. You can use the `join...on...equals...into` phrase to start the left outer join, but you'll want to flatten the resultset. To flatten the resultset, add an additional `from` clause based on the group. If you stop here, though, you are

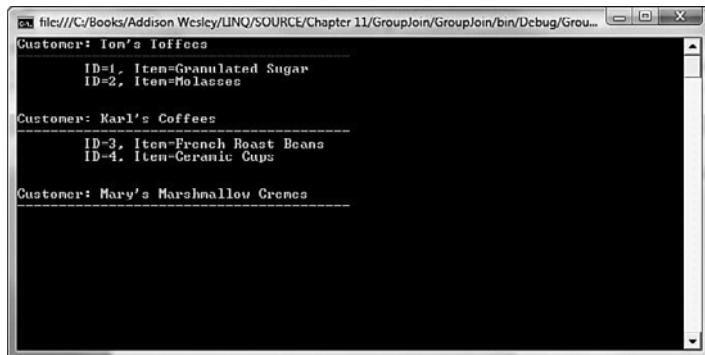


FIGURE 11.2 The output from the group join demonstrates child sequences for every parent element even if the sequence is empty.

simply taking the long route to an inner join. To get master elements that don't have detail elements, you need to incorporate the `DefaultIfEmpty` extension method and express how to handle absent detail elements.

Listing 11.8 shows a revision to the `Main` function for Listing 11.7. This version adds an additional `from` clause and range child. The range child represents all of the elements in the subordinate sequences represented by children. The predicate

```
child in children.DefaultIfEmpty(new Order())
```

means that if there is no `Order` for that child, you need to create an empty instance of the `Order` class. (This is a subtle implementation of the Null Object pattern; the basic concept is also described in Martin Fowler's book, *Refactoring*, as the "Introduce Null Object" refactoring.) The result is that we get Tom's, Karl's, and Mary's in the resultset.

LISTING 11.8 A Revision to Listing 11.7's Main Function That Converts the Basic Group Join into a Left Outer Join, Flattening the Master-Detail Relationship, Which Is the Nature of Group Relationships, into a Standard Join

```

static void Main(string[] args)
{
    List<Customer> customers = new List<Customer>{
        new Customer{ID=1, CompanyName="Tom's Toffees"},
        new Customer{ID=2, CompanyName="Karl's Coffees"},
        new Customer{ID=3, CompanyName="Mary's Marshmallow Cremes"}};

    List<Order> orders = new List<Order>{
        new Order{ID=1, CustomerID=1, ItemDescription="Granulated Sugar"},
        new Order{ID=2, CustomerID=1, ItemDescription="Molasses"},
        new Order{ID=3, CustomerID=2, ItemDescription="French Roast Beans"},
        new Order{ID=4, CustomerID=2, ItemDescription="Ceramic Cups"}};

```

LISTING 11.8 Continued

```
// group join
var group = from customer in customers
            join order in orders on customer.ID equals
            order.CustomerID into children
            from child in children.DefaultIfEmpty(new
            Order())
select new { Customer = customer.CompanyName,
            Item=child.ItemDescription };

foreach (var customer in group)
{
    Console.WriteLine("Customer={0}, Item={1}", customer.Customer,
                      customer.Item);
}
Console.ReadLine();
}
```

Notice that in the `foreach` loop, the internal loop is no longer needed. That's because the query in Listing 11.8 is really no longer represented as a master-detail hierarchical result-set; it is simply a left outer join. As you can see from the output in Figure 11.3, the master elements—the `customers`—have been denormalized and are now repeated for each child element, and the `Customer` objects are present even if no child exists, as in Mary's case.

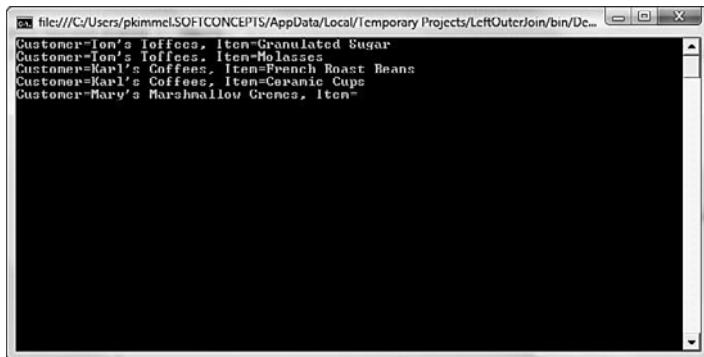


FIGURE 11.3 The group join is converted to a left outer join by including an additional range variable on the group and using `DefaultIfEmpty` to handle null child sequences.

Implementing a Cross Join

A cross join or Cartesian join is the basis for every other join. A cross join is a join between two sources absent any correlative predicate. The word *Cartesian* (after French mathematician René Descartes) refers to the product of the join. For example, if table 1

has 10 rows and table 2 has 25 rows, a cross (or Cartesian) join will produce a result with 250, or every combination of data from table 1 and table 2. Assuming you have a sequence of products and a sequence of customers, the following query (a cross join) would produce all of the combinations of customers and products (the Cartesian product):

```
var test = from customer in customers
           from product in products
           select new
           {
               customer.CompanyName,
               product.ProductName
           };
```

As you can imagine, given a left set with m elements and a right set with n elements, the cross join results in a sequence with $m * n$ elements. For example, a customer table with 1,000 customers and a products table with a 1,000 products cross joined would return 1,000,000 rows of data in the resultset. You can imagine that cross join results can quickly become unmanageable.

In the routine case, a cross join is either intentionally being used to produce test data or is the result of a poorly formed correlation between the data sets. Some clever people have devised a useful purpose for cross joins. Suppose, for example, you have to produce a report for all customers and all products, even showing quantities for products a customer hasn't purchased (where quantity equals 0). (The same approach would work for showing only products a customer isn't buying.) A cross join of customers and products would include every combination of customer and product. Then, you could perform a left join to return every customer and product combination with actual sales figures, filling in quantities and sales values and zeroing out customer-product combinations that have no sales. Listing 11.9 contains such a SQL Server query for the Northwind database that produces a cross join of customers and products and then a left join to fill in the sales figures for customer-product combinations where orders exist. (Table 11.1 contains the first 10 rows of output from the query, and Figure 11.4 shows part of the execution plan, visually describing how SQL Server will execute the query.)

TABLE 11.1 The First 10 Rows Returned from the Cross Join Query in Listing 11.9

Customer ID	Company Name	Product Name	Quantity	Unit Price	Discount	Total Sales
ALFKI	Alfreds Futterkiste	Alice Mutton	0	0.0000	0	0
ALFKI	Alfreds Futterkiste	Aniseed Syrup	6	10.0000	0	60
ALFKI	Alfreds Futterkiste	Boston Crab Meat	0	0.0000	0	0

TABLE 11.1 Continued

Customer ID	Company Name	Product Name	Quantity	Unit Price	Discount	Total Sales
ALFKI	Alfreds Futterkiste	Camembert Pierrot	0	0.0000	0	0
ALFKI	Alfreds Futterkiste	Carnarvon Tigers	0	0.0000	0	0
ALFKI	Alfreds Futterkiste	Chai	0	0.0000	0	0
ALFKI	Alfreds Futterkiste	Chang	0	0.0000	0	0
ALFKI	Alfreds Futterkiste	Chartreuse verte	21	18.0000	0.25	283.5
ALFKI	Alfreds Futterkiste	Chef Anton's Cajun Seasoning	0	0.0000	0	0
ALFKI	Alfreds Futterkiste	Chef Anton's Gumbo Mix	0	0.0000	0	0

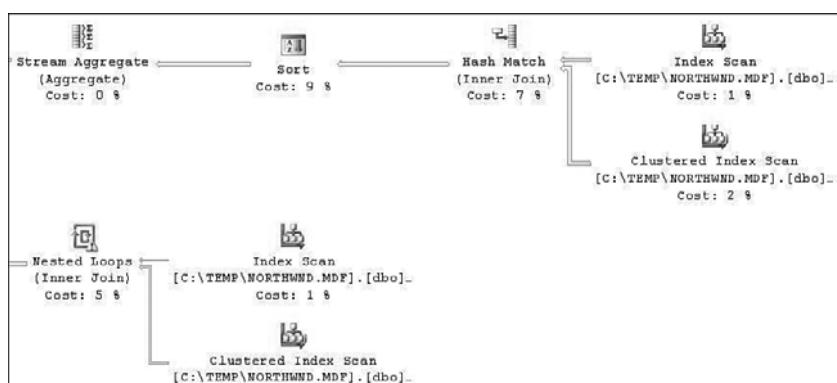


FIGURE 11.4 Part of the execution plan from Microsoft SQL Server Management Studio.

LISTING 11.9 A SQL Query That Uses the Cross Join to Fill in Sales Figures for Customer and Products, Including Zeros for Customer-Product Combinations with No Sales Orders

```

SELECT c.CustomerID, c.CompanyName, p.ProductName, ISNULL(S.Quantity, 0) AS Quantity,
       ISNULL(S.UnitPrice, 0) AS UnitPrice, ISNULL(S.Discount, 0) AS Discount,
       ISNULL(ROUND((S.Quantity * S.UnitPrice) * (1 - S.Discount), 2), 0) AS TotalSales
  FROM Customers AS c CROSS JOIN
       Products AS p LEFT OUTER JOIN
       (SELECT o.CustomerID, od.ProductID, SUM(od.UnitPrice) AS UnitPrice,
              SUM(od.Quantity) AS Quantity, od.Discount
         FROM Orders AS o INNER JOIN
              [Order Details] AS od ON o.OrderID = od.OrderID
        GROUP BY o.CustomerID, od.ProductID, od.Discount) AS S
    ON S.CustomerID = c.CustomerID AND p.ProductID = S.ProductID
 ORDER BY c.CustomerID, p.ProductName, UnitPrice DESC

```

The SQL version (and the LINQ version) require some effort to get right, but in many instances you will be trading lines of code for time because these kinds of monolithic queries are difficult to write, difficult to debug, and difficult to test. Listing 11.10 is an approximation of the query in Listing 11.9. (The plumbing and some sample code was added for testing purposes, but in practice this code would be necessary to write with or without LINQ.)

LISTING 11.10 An Approximation of the SQL in Listing 11.9 as a LINQ Query

```

using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Reflection;

namespace CrossJoinReportMagic
{
    class Program
    {
        static void Main(string[] args)
        {

            var orderInfo = /* left outer join left hand side */
                           from cross in
                           (
                               /* cross join customer and products */
                               from customer in customers
                               from product in products

```

LISTING 11.10 Continued

```
        select new
        {
            customer.CustomerID,
            customer.CompanyName,
            product.ProductID,
            product.ProductName
        })
/* left outer join right hand side */
join groupJoin in
(
    /* group join on orders and details */
    (from order in orders
     join detail in orderDetails on
     order.OrderID equals detail.OrderID
     group detail by new
    {
        order.CustomerID,
        detail.ProductID,
        detail.UnitPrice,
        detail.Discount
    } into groups
    from item in groups
    let Quantity = groups.Sum(d => d.Quantity)
    select new
    {
        groups.Key.CustomerID,
        item.ProductID,
        item.UnitPrice,
        Quantity,
        item.Discount
    }).Distinct()).DefaultIfEmpty()
on new { cross.CustomerID, cross.ProductID } equals
    new { groupJoin.CustomerID, groupJoin.ProductID }
into grouped
from result in grouped.DefaultIfEmpty()
orderby cross.CompanyName, cross.ProductID,
    (result != null ? result.Quantity : 0) descending
select new
{
    CustomerID = cross.CustomerID,
    CompanyName = cross.CompanyName,
    ProductID = cross.ProductID,
    ProductName = cross.ProductName,
    UnitPrice = result== null ? 0 : result.UnitPrice,
```

LISTING 11.10 Continued

```
        Quantity = result== null ? 0 : result.Quantity,
        Discount = result == null ? 0 : result.Discount,
        Total = result == null ? 0 :
            result.Quantity * result.UnitPrice *
            (1 - result.Discount)
    };

Dump(orderInfo);
Console.ReadLine();
}

public static void Dump(object data)
{
    if (data is IEnumerable)
    {
        IEnumerable list = data as IEnumerable;
        IEnumerator enumerator = list.GetEnumerator();
        int i = 0;
        bool once = false;
        while(enumerator.MoveNext())
        {
            i++;
            object current = enumerator.Current;

            PropertyInfo[] props = current.GetType().GetProperties();
            if(!once /* write headers */)
            {
                foreach(PropertyInfo info in props)
                {
                    Console.Write("{0,-10}", info.Name.Length > 9 ? info.Name.Remove(9):
                        info.Name.PadRight(10));
                    once = true;
                }
            }

            foreach (PropertyInfo info in props)
            {
                object value = null;
                try{ value = info.GetValue(current, null); }
                catch { value = "<empty>"; }

                string val = (value.ToString().Length > 9) ?
                    value.ToString().Remove(9) : value.ToString().PadRight(10);
            }
        }
    }
}
```

LISTING 11.10 Continued

```
        if (info.PropertyType == typeof(decimal))
            Console.WriteLine("{0,-10:C02}", Convert.ToDecimal(val));
        else
            if (info.PropertyType == typeof(int))
                Console.WriteLine("{0,-10}", Convert.ToInt32(val));
            else
                if (info.PropertyType == typeof(float))
                    Console.WriteLine("{0,-10:F02}", Convert.ToSingle(val));
                else
                    Console.WriteLine("{0,-10}", val);
    }

    Console.WriteLine(Environment.NewLine);
}

Console.WriteLine("Item count: {0}", i);
}
}

#region collection initializations
private static Customer[] customers = new Customer[]{
    new Customer{CustomerID="ALFKI", CompanyName="Alfreds Futterkiste"},
    new Customer{CustomerID="ANSTR",
        CompanyName="Ana Trujillo Emparedados y helados"},
    new Customer{CustomerID="ANTON", CompanyName="Antonio Moreno Taquería"},
    new Customer{CustomerID="AROUT", CompanyName="Around the Horn"},
    new Customer{CustomerID="BERGS", CompanyName="Berglunds snabbköp"},
    new Customer{CustomerID="SPECD", CompanyName="Spécialités du monde"}
};

private static Product[] products = new Product[]{
    new Product{ProductID=1, ProductName="Chai"},
    new Product{ProductID=2, ProductName="Chang"},
    new Product{ProductID=3, ProductName="Aniseed Syrup"},
    new Product{ProductID=4, ProductName="Chef Anton's Cajun Seasoning"},
    new Product{ProductID=5, ProductName="Chef Anton's Gumbo Mix"},
    new Product{ProductID=6, ProductName="Grandma's Boysenberry Spread"},
    new Product{ProductID=7, ProductName="Uncle Bob's Organic Dried Pears"},
    new Product{ProductID=8, ProductName="Northwoods Cranberry Sauce"},
    new Product{ProductID=9, ProductName="Mishi Kobe Niku"},
    new Product{ProductID=10, ProductName="Ikura"}
};

private static Order[] orders = new Order[]{
    new Order{ OrderID= 10278, CustomerID="BERGS"},
```

LISTING 11.10 Continued

```
new Order{ OrderID= 10308, CustomerID="BERGS"},  
new Order{ OrderID= 10355, CustomerID="AROUT"},  
new Order{ OrderID= 10365, CustomerID="ANTON"},  
new Order{ OrderID= 10383, CustomerID="AROUT"},  
new Order{ OrderID= 10384, CustomerID="BERGS"},  
new Order{ OrderID= 10444, CustomerID="BERGS"},  
new Order{ OrderID= 10445, CustomerID="BERGS"},  
new Order{ OrderID= 10453, CustomerID="AROUT"},  
new Order{ OrderID= 10507, CustomerID="ANTON"},  
new Order{ OrderID= 10524, CustomerID="BERGS"},  
new Order{ OrderID= 10535, CustomerID="ANTON"},  
new Order{ OrderID= 10558, CustomerID="AROUT"},  
new Order{ OrderID= 10572, CustomerID="BERGS"},  
new Order{ OrderID= 10573, CustomerID="ANTON"},  
new Order{ OrderID= 10625, CustomerID="ANATR"},  
new Order{ OrderID= 10626, CustomerID="BERGS"},  
new Order{ OrderID= 10643, CustomerID="ALFKI"}  
};  
  
private static OrderDetail[] orderDetails = new OrderDetail[]{  
    new OrderDetail{OrderID=10278, ProductID=1,  
        UnitPrice=15.5000M, Quantity=16, Discount=0},  
    new OrderDetail{OrderID=10278, ProductID=2,  
        UnitPrice=44.0000M, Quantity=15, Discount=0},  
    new OrderDetail{OrderID=10278, ProductID=3,  
        UnitPrice=35.1000M, Quantity=8, Discount=0},  
    new OrderDetail{OrderID=10278, ProductID=4,  
        UnitPrice=12.0000M, Quantity=25, Discount=0},  
    new OrderDetail{OrderID=10280, ProductID=5,  
        UnitPrice=3.6000M, Quantity=12, Discount=0},  
    new OrderDetail{OrderID=10280, ProductID=6,  
        UnitPrice=19.2000M, Quantity=20, Discount=0},  
    new OrderDetail{OrderID=10280, ProductID=7,  
        UnitPrice=6.2000M, Quantity=30, Discount=0},  
    new OrderDetail{OrderID=10308, ProductID=1,  
        UnitPrice=15.5000M, Quantity=5, Discount=0},  
    new OrderDetail{OrderID=10308, ProductID=9,  
        UnitPrice=12.0000M, Quantity=5, Discount=0},  
    new OrderDetail{OrderID=10355, ProductID=3,  
        UnitPrice=3.6000M, Quantity=25, Discount=0},  
    new OrderDetail{OrderID=10355, ProductID=5,  
        UnitPrice=15.6000M, Quantity=25, Discount=0},  
    new OrderDetail{OrderID=10365, ProductID=7,  
        UnitPrice=16.8000M, Quantity=24, Discount=0},
```

LISTING 11.10 Continued

```
new OrderDetail{OrderID=10383, ProductID=9,
    UnitPrice=4.8000M, Quantity=20, Discount=0},
new OrderDetail{OrderID=10383, ProductID=2,
    UnitPrice=13.0000M, Quantity=15, Discount=0},
new OrderDetail{OrderID=10383, ProductID=4,
    UnitPrice=30.4000M, Quantity=20, Discount=0}
};

#endregion
}

public class Customer
{
    public string CustomerID { get; set; }
    public string CompanyName { get; set; }
}

public class Product
{
    public int ProductID { get; set; }
    public string ProductName { get; set; }
}

public class Order
{
    public int OrderID { get; set; }
    public string CustomerID { get; set; }
}

public class OrderDetail
{
    public int OrderID { get; set; }
    public int ProductID { get; set; }
    public decimal UnitPrice { get; set; }
    public int Quantity { get; set; }
    public decimal Discount { get; set; }
}
```

The LINQ query starts on the line that begins with `var orderInfo` and extends for 56 lines. This probably can't be stressed enough: Although a language, parser, and compiler (and the "Rainman") might be able to comprehend code like the query in Listing 11.10,

normal humans generally can't with the amount of effort that makes it unproductive to produce. As a general practice, it is better to write lucid code and let compilers optimize the code; this is true 99.9% of the time.

Defining Joins Based on Composite Keys

A key is a generally a field that represents some value that is correlated to a like value somewhere else. For example, the preceding section used `CustomerID` from `Customer` and `Order` objects; `CustomerID` is an example of a key. A composite key is a key that is composed of multiple fields. In the parlance of SQL, composite keys are thought of as an index based on multiple columns (or fields). The same basic premise holds true for LINQ.

In LINQ where you would generally expect a single value to express a correlated relationship, composite keys use the new keyword and multiple fields. Listing 11.11 shows a query that defines two composite keys: The first is based on the cross join range variable `cross.CustomerID` and `cross.ProductID` and the second is based on the range variable `groupJoin.CustomerID` and `groupJoin.ProductID`. (Listing 11.10 also shows a composite key for grouping in the phrase that begins with "group detail by new".)

LISTING 11.11 Defining a Composite Key to Correlate Objects That Have Relationships Expressed By More Than a Single Field

```
on new { cross.CustomerID, cross.ProductID } equals
      new { groupJoin.CustomerID, groupJoin.ProductID }
```

Summary

There is a lot to LINQ joins just as there is a lot to SQL joins. One of the biggest challenges is: When does good taste suggest you write a compound series of statements versus a monolithic statement? There were some studies done that suggest the human, short-term mind can juggle 7 to 10 items at a time. So, a good rule of thumb is if you feel yourself dropping some juggled items, your successors will find the queries difficult to juggle too.

Master-detail relationships composed of two sequences to create a new, third sequence are pretty straightforward and, luckily, they are the most common type. To address everyday kinds of joins, this chapter showed how LINQ can support inner joins, group joins, left joins, cross joins, custom joins, and joins based on composite keys. The inner join is probably one of the most routine, followed by the left join. If you find yourself needing right joins (which this chapter didn't cover), cross joins, or custom joins, like everything else, it will take a lot of practice to become proficient. Hopefully, this chapter helps get you started.

This page intentionally left blank

CHAPTER 12

Querying Outlook and Active Directory

“Find a job you like and you add five days to every week.”

—H. Jackson Brown, Jr.

A lot of data comes from traditional relational databases. From these sources, it is plausible to choose SQL or LINQ for similar kinds of tasks. A lot of other data comes from places other than relational databases. Here, LINQ shines as the best alternative.

Anything that serves data as an enumerable collection can be queried with LINQ. For example, Microsoft Outlook mailboxes, contacts, and calendar items can be queried because the Outlook object model exposes these elements as collections. This chapter provides an Outlook example. Active Directory, unfortunately, has its own query language—probably an oddity that shouldn’t be. However, the .NET Framework supports implementing a custom `IQueryable` provider. Queryable providers let us conceptually treat Active Directory like a LINQ-queryable repository. This chapter also demonstrates how to create a provider for LINQ. You can use the provider example to create providers for other applications such as SharePoint.

LINQ to Outlook

OLE (Object Linking and Embedding) automation is a pretty old technology. It is part of what is collectively called COM (Component Object Model). Many Windows tools have their own exposed object model. For example, Microsoft PowerPoint, Microsoft Excel, Microsoft Access, and Outlook all have an object model that you can code

IN THIS CHAPTER

- ▶ LINQ to Outlook
- ▶ Querying Active Directory with Straight C# Code
- ▶ LINQ to Active Directory
- ▶ Querying Active Directory with LINQ

against. Further, because these object models contain data as collections, you can use LINQ to query these resources. (For more on Outlook programming with Visual Basic for Applications [VBA], check out Patricia DiGiacomo's book *Special Edition Using Microsoft Office Outlook 2007* from Que.)

Listing 12.1 uses LINQ to read Outlook's Inbox and contacts. Any email address that is in the Inbox and not in the contactList is added as a contact. (As long as you clean junk mail out of your Inbox before you run this code, it's a great way to update your contact list.)

LISTING 12.1 Scanning the Inbox and Contacts with LINQ and Adding New Email Addresses to the Contacts Folder

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Microsoft.Office.Interop.Outlook;
using System.Runtime.InteropServices;

namespace QueryOutlook
{
    class Program
    {
        static void Main(string[] args)
        {
            _Application outlook = new Application();
            if (outlook.ActiveExplorer() == null)
            {
                Console.WriteLine("open outlook and try again");
                Console.ReadLine();
                return;
            }

            MAPIFolder inbox =
                outlook.ActiveExplorer().Session
                .GetDefaultFolder(OlDefaultFolders.olFolderInbox);
            MAPIFolder contacts =
                outlook.ActiveExplorer().Session
                .GetDefaultFolder(OlDefaultFolders.olFolderContacts);

            var emails = from email in inbox.Items.OfType<MailItem>()
                         select email;

            var contactList = from contact in contacts.Items.OfType<ContactItem>()
                             select contact;

            var emailsToAdd = (from email in emails
```

LISTING 12.1 Continued

```
let existingContacts = from contact in contactList
select contact.Email1Address
where !existingContacts.Contains(email.SenderEmailAddress)
orderby email.SenderName
select new { email.SenderName, email.SenderEmailAddress }).Distinct();

Array.ForEach(emailsToAdd.ToArray(), e =>
{
    ContactItem contact = ContactItem(outlook.CreateItem(
        OlItemType.olContactItem));
    contact.Email1Address = e.SenderEmailAddress;
    contact.Email1DisplayName = e.SenderName;
    contact.FileAs = e.SenderName;
    try
    {
        contact.Save();
    }
    catch (COMException ex)
    {
        Console.WriteLine(ex.Message);
        Console.ReadLine();
    }
    Console.WriteLine("Adding: {0}", e.SenderName);
}
);
Console.ReadLine();
}
}
```

To reproduce the example, add a reference to the Microsoft.Office.Interop.Outlook assembly (see Figure 12.1). (The example uses Outlook 2007 but Outlook 2003 should work too.) Then, you need to create an instance of Outlook. Listing 12.1 is written in such a way that it anticipates that Outlook is running at present.

TIP

All the Office object models use Application as the object model's class name. If you are programming against multiple Office object models at the same time, you need to qualify the Application class with the namespace.

The statement

```
Application outlook = new Application();
```

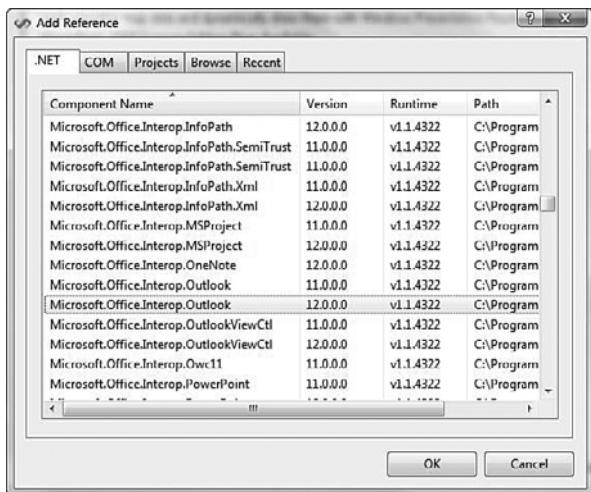


FIGURE 12.1 Add a reference to the `Microsoft.Office.Interop.Outlook` assembly to program against Outlook's object model.

creates an instance of the Outlook application. The two statements that begin with `MAPIFolder` get the Inbox and Contacts folders, respectively. The `MAPIFolder` class's `Items` property realizes the `IEnumerable` interface and that's all you need to be able to use LINQ. The first two LINQ queries use the `OfType` generic conversion operator to convert the `Items` collections into queryable sequences of `MailItem` and `ContactItem` objects, respectively.

After the code obtains the email items from the Inbox `MAPIFolder` and the contacts from the contacts `MAPIFolder`, you can use both sequences to find only email addresses in the Inbox that aren't in the Contacts folder. The query beginning with the anonymous type `emailsToAdd` finds the email addresses.

The anonymous type `emailsToAdd` is a collection of projections containing the sender's name and the sender's email address using the `Distinct` set extension method to exclude duplicates. The range value `email` is the iterator for each email address. The range value `existingContacts` is defined with a `let` clause that selects existing contact email addresses. The `where` clause predicate excludes email addresses that are already in the Contacts folder and the results are ordered.

Finally, the `Array.ForEach` method and compound Lambda Expression representing the `Action` creates new contact items for new email addresses filling in the sender's name, email address, and the `FileAs` field and saves the result.

Unfortunately, because of unscrupulous twits and miscreants, Outlook will display the Allow access dialog box (see Figure 12.2). Check the Allow Access For check box (1 minute should be plenty of time) and click Yes to permit the code to update your Contacts folder.



FIGURE 12.2 Sadly, there are smart people in the world who don't always act so smart, creating problems for the rest of us, making speed bumps like the Allow access dialog box a necessary evil.

Querying Active Directory with Straight C# Code

Active Directory is pretty much a de facto standard for authentication and authorization. Some applications might not use it. For instance, some public Internet sites don't manage subscribers with Active Directory, but for applications and internal web applications, it's a great tool for storing user information and defining roles.

A couple of minor drawbacks to using Active Directory are that it uses its own cryptic query language and the schema names are a little goofy—*dn* for distinguished name, *ou* for organizational unit, and similar abbreviations don't make querying Active Directory intuitive. You can remedy that, however, which you learn to do in the next section, "LINQ to Active Directory." First, let's look at some straight C# code that queries Active Directory (see Listing 12.2).

LISTING 12.2 Straight C# Code That Uses System.DirectoryServices and Straight C# Code to Query Active Directory

```
private static readonly string LDAP =
    "LDAP://sci.softconcepts.com:3268/DC=softconcepts,DC=COM";
private static DirectoryEntry activeDirectory = new DirectoryEntry(LDAP);
private static readonly string user = "xxxxxxxx";
private static readonly string password = "xxxxxxxx";

using(DirectoryEntry entry = new
    DirectoryEntry(LDAP, user, password, AuthenticationTypes.None))
{
    string filter = "(&(objectClass=person))";
    DirectorySearcher searcher = new DirectorySearcher(entry, filter);
    var results = searcher.FindAll();
```

LISTING 12.2 Continued

```
foreach (SearchResult result in results)
{
    Console.WriteLine(result.Properties["Name"][0]);
    Console.WriteLine(Environment.NewLine);
}
Console.ReadLine();
```

Active Directory is based on the Lightweight Directory Access Protocol called *LDAP*. LDAP, like many protocols, is based on whitepapers (or RFCs, *Request for Comments*). There are several RFCs for Active Directory. For example, RFC 3377 is the LDAP version 3 technical specification and RFC 2254 is the specification for LDAP search filters. You can find these specifications by Googling “LDAP RFC” if you are interested.

RFCs for standardized technologies are written by interested parties, generally software and hardware vendors for computer technologies. The idea is that stakeholders who will be using these technologies want a say in the details of the specification. (Unfortunately, sometimes committee designs produce oddities like *DN*, *OU*, and *DC* and foist these decisions on the rest of us who are just trying to remain sane.)

Active Directory is an implementation of the LDAP specification from Microsoft. The basic implementation in conjunction with .NET Framework’s `System.DirectoryServices` requires that you create an instance of a `DirectoryEntry` using a uniform resource identifier (URI) with the `LDAP://` moniker, the path, the port, and some parameters to the Active Directory instance. In the LDAP constant string, the path to an Active Directory server is included along with the arguments for the DC (domain component). (There are other options, but whole books have been written on Active Directory. For more information, see Gil Kirkpatrick’s book *Active Directory Programming* from Que.)

TIP

If you have access to the server containing an Active Directory instance, you can use the `dsquery` utility to experiment with Active Directory queries. The following `dsquery` command is roughly equivalent to the query in Listing 12.2:

`Dsquery * domainroot –filter “(&(objectClass=Person))”`

Having defined the connection URI and created an instance of a `DirectoryEntry`, you can now define a `DirectorySearcher` with a search filter. In the example, the filter is grabbing everything defined as a person. The results will be people stored in Active Directory.

By passing in a test name and some minor modifications, you can use the code in Listing 12.2 to authenticate application users. The catch is that Active Directory is one of those

technologies that require specialized knowledge. If you implement an Active Directory provider for .NET, you can make Active Directory easier to use without requiring that every developer learn how to string together Active Directory filters.

LINQ to Active Directory

12

.NET supports implementing an `IQueryable` provider. `IQueryable` is an interface that you can inject between some repository or data source that you want to query using LINQ. Basically, it works like this: You write a LINQ query, LINQ calls your custom `IQueryable` object handing the LINQ query off, and you write code (one time) that converts the LINQ query to something the repository can understand. For instance, you want to write LINQ queries but Active Directory still wants filters in the predetermined format. The custom provider performs the conversion.

This takes some doing because LINQ and Active Directory are substantially different, but the important thing is you only need to do this once and then everyone benefits. Before we begin, go get a cup of coffee. (I will wait.)

NOTE

Bart De Smet of Microsoft has an excellent implementation of an `IQueryable` provider for Active Directory at <http://www.codeplex.com/LinqToAD>. Bart permitted me to borrow heavily from his implementation and provided a lot of insights into how his code worked. Thanks, Bart.

Back? Good.

When you use `DirectoryServices`, `DirectoryEntry` objects, and `DirectorySearcher` objects, the data you get back is `SearchResult` objects. The challenge is that the `SearchResult` objects are key and value pairs, necessitating writing code like `result.Properties["Name"][0]` to get the Name attribute. In addition to being unsightly, this code is very natural to read or write. By creating the provider, you can query Active Directory with clearly defined entities and LINQ, resulting in a much more natural looking query.

Creating an `IQueryable` LINQ Provider

The amount of code to create an `IQueryable` provider is much longer than is generally convenient to include in a book. The total listing is a couple of thousand lines. However, this topic is really the kind of juicy material we want in an Unleashed book. Also, because

the provider is intertwined with the framework—calls are made from the framework to the provider—it is a little hard to figure out what is going on just by looking at the code.

To solve the problem of space, enough of the code is listed to make it clear how each piece works, but some of the code is elided. All of the code is available for download from Sams on the book's product page (www.informit.com/title/9780672329838). In addition, to help provide an overview of the elements, a couple of class diagrams and a sequence diagram with a brief explanation of how to read the diagrams are also provided later in this chapter. Finally, all of the pieces are described in their own section, including the relevant code, models, and other visualizations.

NOTE

Microsoft has an `IQueryable` provider for their Terra Server and other samples like the `ObjectDumper` available for download at <http://msdn2.microsoft.com/en-us/bb330936.aspx>.

The following is an overview of the steps you'll need to follow and the code to create to implement the Active Directory provider for LINQ:

1. Create a solution with class library and console projects. The class library will contain the provider, and the console application is for testing the provider.
2. Code a class that implements `IQueryable<T>` or `IOrderedQueryable<T>`. You'll implement `IOrderedQueryable<T>` so that your provider supports `orderby` clauses.
3. Define entities that map to Active Directory schema elements, such as `User`.
4. Define the context class, the `IQueryProvider` that is used as the entry throughway from LINQ queries to Active Directory.
5. Define the code that translates LINQ expression trees into Active Directory filters and executes those filters against the Active Directory server.
6. Finally, define Active Directory attributes that facilitate mapping programmer-friendly names to elements in the Active Directory schema.

The last step is to write some LINQ queries and test the provider. Let's begin with the `IQueryProvider` interface.

Implementing the `IQueryProvider`

The `IQueryProvider` interface defines four methods: two versions of `CreateQuery` and two versions of `Execute`. In this sample, direct execution isn't supported and entity types are defined, so it just implements the generic version of `CreateQuery`. Listing 12.3 shows the complete listing of the `DirectoryQueryProvider`.

LISTING 12.3 The `IQueryProvider` Class Is the Starting Point for LINQ Query Execution

```
using System;
using System.Linq;
using System.Linq.Expressions;

namespace LinqToActiveDirectory
{
    public class DirectoryQueryProvider : IQueryProvider
    {

        public IQueryable<TElement> CreateQuery<TElement>(Expression expression)
        {
            return new DirectoryQuery<TElement>(expression);
        }

        public IQueryable CreateQuery(Expression expression)
        {
            throw new NotImplementedException();
        }

        public TResult Execute<TResult>(Expression expression)
        {
            throw new NotImplementedException();
        }

        public object Execute(Expression expression)
        {
            throw new NotImplementedException();
        }
    }
}
```

When the framework is ready to process the query, it asks the `IOrderedQueryable` provider for the `IQueryProvider` (which is the class in Listing 12.3) and `CreateQuery` is called to begin the transmogrification of the LINQ query to an Active Directory filter and search operation.

The role played by the `IQueryProvider` is shown in the redacted part of the sequence diagram shown in Figure 12.3. The sequence diagram is read from top left to bottom right. The classes are the boxes across the top and the lines are method calls. The start of the line is the class containing the call and the arrow points to the class implementing the method. The convention of `get_name` is used to indicate a property call, as in `get_Provider`. (Property calls are really method calls under the hood.)

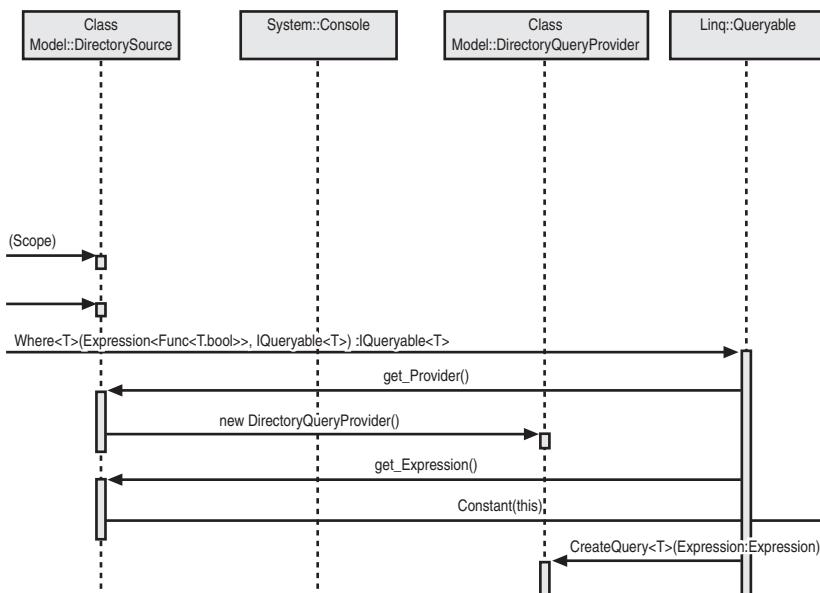


FIGURE 12.3 The `System.Linq.Queryable` framework class calls `DirectorySource`, which implements `IOrderedQueryable` to request the provider—the `IQueryProvider`—and, in turn, creates the `DirectoryQuery` class, which implements the `IQueryable` interface.

Without getting too far ahead of ourselves here, the goal is for the framework to get an instance of the translator (the provider) that converts LINQ query elements into analogous calls to the source of the data; in this case, Active Directory. Because Microsoft has no way of knowing all of the future possible providers people will want, they have to use interfaces to construct a common pattern for implementing providers in general.

Defining Active Directory as the Data Source

The next piece of code defines is the `IQueryable` class. Well, we will actually define a class that implements `IOrderedQueryable`, which inherits from `IQueryable`. `IQueryable` is the interface that has all of those neat extension methods like `Where<T>` that were introduced as the underpinnings for LINQ. We want `IOrderedQueryable` because it inherits from `IQueryable` so we can use `Select<T>` and `Where<T>`, but `IOrderedQueryable` is the interface that is the return type for the extension method `OrderBy`, which permits us to use `orderby` clauses with ascending and descending modifiers. Here is the definition of the `OrderBy` method in `System.Linq`:

```

public static IOrderedQueryable<TSource> OrderBy<TSource, TKey>(
    this IQueryable<TSource> source,
    Expression<Func<TSource, TKey>> keySelector,
    IComparer<TKey> comparer
)

```

A Masterpiece of Software Engineering

I hope you begin to get a mental picture of all of the complex relationships, language features, and design that went into making the LINQ plumbing work. LINQ encompasses extension methods, generics, polymorphism—of both the class and interface variety—and much more. The .NET Framework really is a masterpiece of software engineering.

`IOrderedQueryable` requires that you implement a `GetEnumerator` method and the properties `ElementType`, `Expression`, and `Provider`. Listing 12.4 contains the complete listing for the `IOrderedQueryable` class, including a `TextWriter` property that lets you bind the `Console.Out` stream (the console window) to the `DirectorySource` class. The `TextWriter` property makes it easy to send information from the provider to the console (which was a clever trick that Bart De Smet employed in his example).

LISTING 12.4 The `IOrderedQueryable` Provider Lets Us Use LINQ to Active Directory.

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.ComponentModel;
using System.DirectoryServices;
using System.IO;
using System.Linq;
using System.Linq.Expressions;

namespace LinqToActiveDirectory
{
    public class DirectorySource<T> : IOrderedQueryable<T>, IDirectorySource
    {
        private DirectoryEntry searchRoot;
        private SearchScope searchScope;
        private TextWriter log;

        public DirectorySource(DirectoryEntry searchRoot, SearchScope searchScope)
        {
            this.searchRoot = searchRoot;
            this.searchScope = searchScope;
        }

        public TextWriter Log
        {
            get { return log; }
            set { log = value; }
        }
    }
}
```

LISTING 12.4 Continued

```
public DirectoryEntry Root
{
    get { return searchRoot; }
}

public SearchScope Scope
{
    get { return searchScope; }
}

public Type ElementType
{
    get { return typeof(T); }
}

public Type OriginalType
{
    get { return typeof(T); }
}

public Expression Expression
{
    get { return Expression.Constant(this); }
}

public IQueryProvider Provider
{
    get { return new DirectoryQueryProvider(); }
}

IEnumerator IEnumerable.GetEnumerator()
{
    return GetEnumerator();
}

public IEnumerator<T> GetEnumerator()
{
    return new DirectoryQuery<T>(this.Expression).GetEnumerator();
}
```

The `DirectorySource` in Listing 12.4 keeps track of the `DirectoryEntry`, which provides the underlying access to Active Directory through `DirectoryServices`. `DirectorySource` also implements the `IOrderedQueryable` interface and is asked by the .NET Framework to return the `Expression`, `Provider`, element type, and the enumerator. `Provider` is the class that we define that implements `IQueryProvider`, `GetEnumerator` returns access to the underlying data from Active Directory, and `Expression` is the abstract expression class that represents expression tree nodes.

The partial sequence diagram in Figure 12.3 shows the point in time at which the expressions are requested. Later, the expressions are parsed in a loop into its constituent elements by the `DirectoryQuery` class (see Figure 12.4).

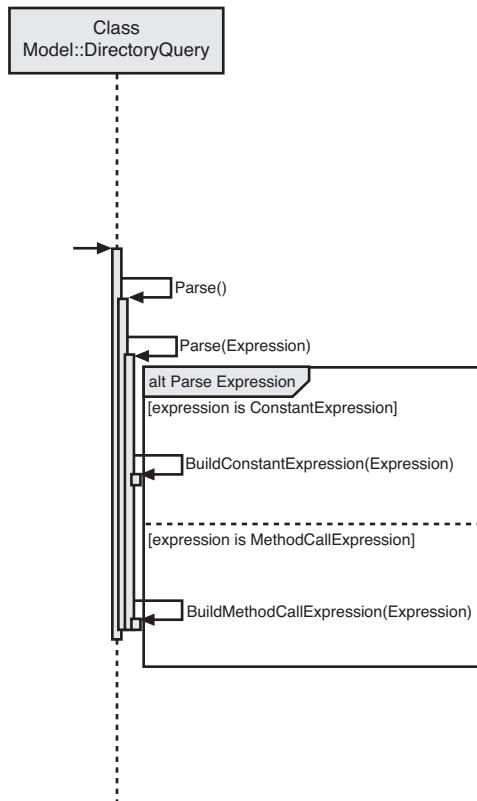


FIGURE 12.4 The abstract expression class is the `DirectorySource<T>`—for instance `DirectorySource<User>`—representing extension methods and Lambda Expressions.

The abstract `Expression` is an instance of `DirectorySource<T>`, for example, `DirectorySource<User>`. It represents parts of the expression that are composed of extension methods such as `Where<T>` and Lambda Expressions such as `user.Name == "Paul"`

Kimmel". The `DirectoryQuery` converts these Lambda Expressions into Active Directory queries.

TIP

Add a `TextWriter` property to your classes and assign the `textWriter` to the `Console.Out` property to support writing state information to the console for debugging and testing.

Remember that LINQ queries are converted to extension method calls containing Lambda Expressions and that Lambda Expressions can be compiled as code or expression tree data (refer to Chapter 5, "Understanding Lambda Expressions and Closures"). Chapter 5 stated that expression trees are used to convert LINQ to other forms like SQL. The convertible expression tree mechanism is what is used for `IQueryable` providers.

Converting a LINQ Query to an Active Directory Query

The `DirectoryQuery` is the real heart of this example. Although it would be nice to list it all here, its 593 lines and really long code listings don't make for very good reading. Salient parts are listed, but what the class does at its heart is this: `DirectoryQuery` has to convert a LINQ query to an Active Directory query.

NOTE

All of the code for this book is available at the book's product page (www.informit.com/title/9780672329838), but if you have any difficulties getting the code or understanding it, feel free to write me at pkimmel@softconcepts.com.

`DirectoryQuery` is fed all of the expression when `DirectoryQuery` is constructed. When `DirectoryQuery.GetEnumerator` is called, it recursively parses the expression tree. The parse behavior looks at the expression and asks, "*Do I have an extension method call or do I have the Lambda Expression argument?*" (For example, is it the `Where` or the Lambda Expression predicate that needs parsing right now?) If it is at the point of parsing the `Where<T>` extension method, it recoursees and parses the argument to `Where`. When `Parse` returns from the recursive call, it figures out what method call it has—in this example, `Where`—and turns the `Where` into an Active Directory filter. Thus,

```
where user.Name == "Paul Kimmel" || user.Name == "Guest"
```

in the LINQ query becomes

```
(| (Name=Paul Kimmel) (Name=Guest)
```

its Active Directory equivalent.

The `DirectoryQuery` class is where the code was borrowed heavily from Bart De Smet. However, the code was refactored to fit my writing style and new elements were added and some removed. Bart's code at <http://www.codeplex.com/LinqToAD> contains an Active Directory Update capability. That's removed from the download sample from this book. I added the ability to sort in ascending and descending order. Sorts are not provided directly by Active Directory filters but are provided by `DirectoryServices`.

Listing 12.5 shows a `BuildMethodCallExpression` that handles `OrderBy` and `OrderByDescending` (`orderby` ascending or `orderby` descending). Rather than build a filter (which doesn't exist for sorting), a `SortOption`'s properties are set and the `SortOption` object is assigned to the `DirectorySearcher.Sort` property in the `GetResults` method (see Listing 12.6).

LISTING 12.5 BuildMethodCallExpression Assigns Fields from Active Directory Values to Project Fields for Select, Builds a Filter for Where, and Creates a SortOption Object for the DirectorySearcher for OrderBy and OrderByDescending

```
private void BuildMethodCallExpression(MethodCallExpression methodCallExpression)
{
    CheckDeclaringType(methodCallExpression);
    Parse(methodCallExpression.Arguments[0]);
    // always a unary expression
    LambdaExpression unary = ((UnaryExpression)
        methodCallExpression.Arguments[1]).Operand as LambdaExpression;

    switch (methodCallExpression.Method.Name)
    {
        case "Where":
            BuildPredicate(unary);
            break;

        case "Select":
            BuildProjection(unary);
            break;
        case "OrderBy":
        case "OrderByDescending":
            option = new SortOption();
            option.PropertyName = GetPropertyNameFromUnary(unary);
            option.Direction = GetSortDirection(methodCallExpression.Method.Name);
            break;
        default:
            ThrowUnsupported(methodCallExpression.Method.Name);
            break;
    }
}
```

LISTING 12.6 Ties All of the Elements of Select, Where, and OrderBy Together to Construct a DirectorySearcher from DirectoryServices and Obtain the Data from Active Directory

```
private IEnumarator<T> GetResults()
{
    DirectorySchemaAttribute[] attribute = GetAttribute();
    DirectoryEntry root = directorySource.Root;
    string formattedQuery = GetFormattedQuery(attribute);

    DirectorySearcher searcher = new DirectorySearcher(root,
        formattedQuery, properties.ToArray(), directorySource.Scope);

    if(option != null)
        searcher.Sort = option;

    WriteLog(formattedQuery);
    Type helper = attribute[0].HelperType;

    foreach (SearchResult searchResult in searcher.FindAll())
    {
        DirectoryEntry entry = searchResult.GetDirectoryEntry();
        object result = Activator.CreateInstance(project == null ?
            typeof(T) : originalType);
        if (project == null)
        {
            foreach ( PropertyInfo info in typeof(T).GetProperties())
                AssignResultProperty(helper, entry, result, info.Name);
            yield return (T)result;
        }
        else
        {
            foreach (string prop in properties)
                AssignResultProperty(helper, entry, result, prop);
            yield return (T)project.DynamicInvoke(result);
        }
    }
}
```

Listing 12.7 shows how the `Where` predicate is converted to an Active Directory search filter. Listing 12.8 shows how reflection is used in conjunction with `GetResults` to assign Active Directory properties to select projection properties, and Figure 12.5 shows a UML class diagram to illustrate how the primary elements are related to each other.

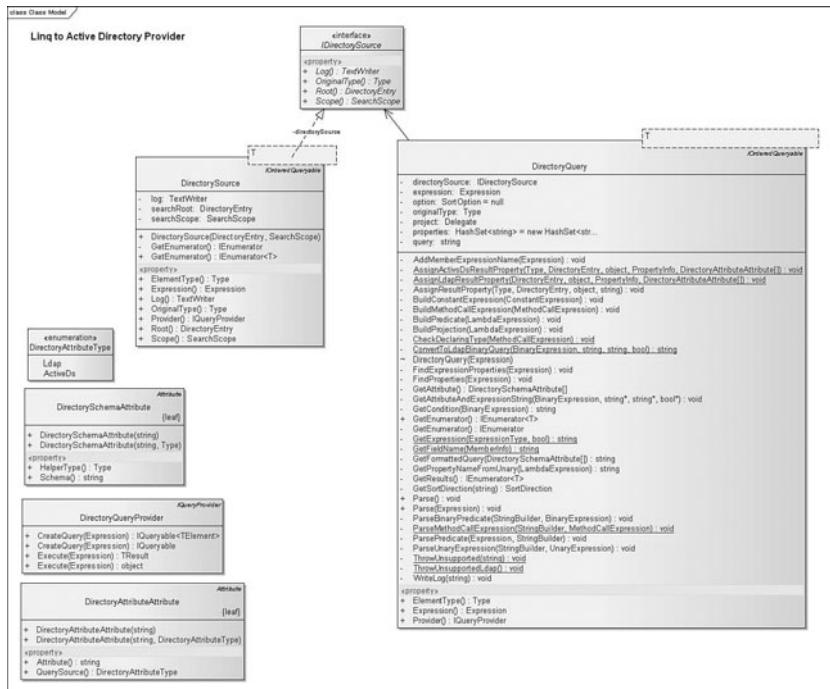


FIGURE 12.5 A class diagram depicting significant elements of the solution and how they fit together.

LISTING 12.7 Converting Where Lambda Expressions into Active Directory Search Filters

```

private void BuildPredicate(LambdaExpression lambda)
{
    StringBuilder builder = new StringBuilder();
    ParsePredicate(lambda.Body, builder);
    query = builder.ToString();
}

private void ParsePredicate(Expression expression, StringBuilder builder)
{
    builder.Append("(");
    if (expression as BinaryExpression != null)
    {
        ParseBinaryPredicate(builder, expression as BinaryExpression);
    }
    else if (expression as UnaryExpression != null)

```

LISTING 12.7 Continued

```
{  
    ParseUnaryExpression(builder, expression as UnaryExpression);  
}  
else if (expression as MethodCallExpression != null)  
{  
    ParseMethodCallExpression(builder, expression as MethodCallExpression);  
}  
else  
    throw new NotSupportedException(  
        "Unsupported query expression detected. Cannot translate to LDAP equivalent.");  
    builder.Append(")");  
}  
  
private void ParseBinaryPredicate(StringBuilder builder, BinaryExpression binary)  
{  
    switch (binary.NodeType)  
    {  
        case ExpressionType.AndAlso:  
            builder.Append("&");  
            ParsePredicate(binary.Left, builder);  
            ParsePredicate(binary.Right, builder);  
            break;  
        case ExpressionType.OrElse:  
            builder.Append(" | ");  
            ParsePredicate(binary.Left, builder);  
            ParsePredicate(binary.Right, builder);  
            break;  
        default: //E.g. Equal, NotEqual, GreaterThan  
            builder.Append(GetCondition(binary));  
            break;  
    }  
}  
  
private void ParseUnaryExpression(StringBuilder builder, UnaryExpression unary)  
{  
    if (unary.NodeType == ExpressionType.Not)  
    {  
        builder.Append("!");  
        ParsePredicate(unary.Operand, builder);  
    }  
    else  
        throw new NotSupportedException(  
            "Unsupported query operator detected: " + unary.NodeType);  
}
```

LISTING 12.8 Storing Fields from the Active Directory Query in a Hashtable to Assign to Anonymous Project Types in the select Clause

```
private void BuildProjection(LambdaExpression lambda)
{
    project = lambda.Compile();
    originalType = lambda.Parameters[0].Type;

    MemberInitExpression memberInitExpression = lambda.Body as MemberInitExpression;

    if (memberInitExpression != null)
        foreach (MemberAssignment memberAssignment in memberInitExpression.Bindings)
            FindProperties(memberAssignment.Expression);
    else
        foreach ( PropertyInfo info in originalType.GetProperties() )
            properties.Add(info.Name);
}

/// <summary>
/// Recursive helper method to finds all required properties for projection.
/// </summary>
/// <param name="expression">Expression to detect property uses for.</param>
private void FindProperties(Expression expression)
{
    if (expression.NodeType == ExpressionType.MemberAccess)
    {
        AddMemberExpressionName(expression);
    }
    else
    {
        FindExpressionProperties(expression);
    }
}
```

Implementing Helper Attributes

The two attribute classes exist to map Active Directory schema and name elements to our programmer-friendly entities and names. For example, the next section defines an entity `user` that maps to the `IADsUser` interface from `Interop.ActiveDS` to an Active Directory user using the `DirectorySchemaAttribute` and maps elements like `Dn` to the Active Directory `distinguishedName` attribute.

Listing 12.9 contains the code for `DirectorySchemaAttribute`, and Listing 12.10 contains the code for `DirectoryAttributeAttribute`.

LISTING 12.9 The `DirectorySchemaAttribute` Is Used to Indicate to Which Active Directory Schema the Entities Map

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace LinqToActiveDirectory
{
    public sealed class DirectorySchemaAttribute : Attribute
    {

        public DirectorySchemaAttribute(string schema)
        {
            Schema = schema;
        }

        public DirectorySchemaAttribute(string schema, Type type)
        {
            Schema = schema;
            HelperType = type;
        }

        public string Schema { get; private set; }
        public Type HelperType { get; set; }
    }
}
```

LISTING 12.10 `DirectoryAttributeAttribute` Is Used to Indicate to What Field an Entity Field Maps and Whether It Maps Using the IADsUser Field Names or LDAP Field Names

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace LinqToActiveDirectory
{
    public sealed class DirectoryAttributeAttribute : Attribute
    {

        public DirectoryAttributeAttribute(string attribute)
        {
            Attribute = attribute;
            QuerySource = DirectoryAttributeType.Ldap;
        }
    }
}
```

LISTING 12.10 Continued

```
}

public DirectoryAttributeAttribute(string attribute,
    DirectoryAttributeType querySource)
{
    Attribute = attribute;
    QuerySource = querySource;
}

public string Attribute { get; private set; }
public DirectoryAttributeType QuerySource { get; set; }
}
```

Defining Active Directory Schema Entities

The last element of the sample provider is to define program-specific entities—that is, classes that contain those elements we want to obtain from Active Directory's objects. Listing 12.11 defines a User class that is mapped to the `IADsUser` definition of a user and Listing 12.12 defines a Group that is mapped to the LDAP definition of a group. `IADsUser` is an interface defined as part of the Active Directory Services Interfaces (ADSI).

LISTING 12.11 The User Entity Maps to an Active Directory User

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using LinqToActiveDirectory;
using ActiveDs;

namespace QueryActiveDirectory
{
    [DirectorySchema("user", typeof(IADsUser))]
    class User
    {
        public string Name { get; set; }

        public string Description { get; set; }

        public int LogonCount { get; set; }

        [DirectoryAttribute("PasswordLastChanged", DirectoryAttributeType.ActiveDs)]
        public DateTime PasswordLastSet { get; set; }
    }
}
```

LISTING 12.11 Continued

```
[DirectoryAttribute("distinguishedName")]
public string Dn { get; set; }

[DirectoryAttribute("memberOf")]
public string[] Groups { get; set; }
}

}
```

LISTING 12.12 The Group Entity Maps to an Active Directory Group

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using LinqToActiveDirectory;

namespace QueryActiveDirectory
{
    [DirectorySchema("group")]
    class Group
    {
        public string Name { get; set; }

        [DirectoryAttribute("member")]
        public string[] Members { get; set; }
    }
}
```

The entity classes are used to construct a `DirectorySource`. The `Type` of the entity is used to determine what schema and properties should be obtained from the Active Directory query results.

Querying Active Directory with LINQ

After a considerable amount of plumbing, you can now get data from Active Directory as easily as you might from a custom object or SQL Server (using SQL for LINQ). Remember that the Active Directory LINQ provider is portable across applications so you only have to write this code once. (Well, you can just download the code.)

Listing 12.13 shows two LINQ queries. The first selects from users and returns a projection containing the name only for “Paul Kimmel” or “Guest.” The second query shows a query against groups filtering on the `Name` and sorting in descending order.

LISTING 12.13 Using the Active Directory LINQ Provider to Query Users and Groups

12

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.DirectoryServices;
using System.Collections;
using LinqToActiveDirectory;

namespace QueryActiveDirectory
{
    class Program
    {
        static void Main(string[] args)
        {
            GetUsersLinq();
        }

        private static readonly string LDAP =
            "LDAP://sci.softconcepts.com:3268/DC=softconcepts,DC=COM";
        static DirectoryEntry activeDirectory = new DirectoryEntry(LDAP);
        private static readonly string user = "xxxxxxxx";
        private static readonly string password = "xxxxxxxx";

        private static void GetUsersLinq()
        {
            var users = new DirectorySource<User>(activeDirectory, SearchScope.Subtree);
            users.Log = Console.Out;
            var groups = new DirectorySource<Group>(
                activeDirectory, SearchScope.Subtree);
            groups.Log = Console.Out;

            var results = from user in users
                         where user.Name == "Paul Kimmel" ||
                               user.Name == "Guest"
                         select new { user.Name };

            Console.WriteLine("users");
            foreach (var r in results)
                Console.WriteLine(r.Name);

            Console.ReadLine();
            Console.WriteLine(new string('-', 40));
        }
    }
}
```

LISTING 12.13 Continued

```
Console.WriteLine("groups");

var groupResults = from group1 in groups
    where group1.Name == "Users" ||
    group1.Name == "IIS_WPG"
    orderby group1.Name descending
    select group1.Name;

Array.ForEach(groupResults.ToArray(), g =>
{
    Console.WriteLine("Group: {0}", g.Name);
    Array.ForEach(g.Members.ToArray(), m => Console.WriteLine(m));
})
}
```

The prep work is that you construct a `DirectoryEntry` from `System.DirectoryServices`. The `DirectoryEntry` and `SearchScope` are used to construct an instance of a `DirectorySource`. The parameter (`User` or `Group`) defines the schema you will be querying from Active Directory and helps map the Active Directory fields to your entity fields. The `Console.Out` is assigned to facilitate printing what's going on in the provider and then you write LINQ queries.

Summary

As you can see from this chapter, LINQ isn't limited to custom objects, XML, or SQL. You can query anything that is represented as a collection, as was demonstrated by the Outlook folder query examples, and you can write an `IQueryable` provider to map LINQ to completely new data providers such as Active Directory.

You will need to download the code for this chapter. (Most of the other chapters contain the complete listing.) And, I encourage you to check Bart De Smet's example at <http://www.codeplex.com/LinqToAD>. Bart has indicated that he is committed to maintaining and extending that code.

The same capability we used to build an Active Directory provider in this chapter was used to implement support for XML and ADO/SQL. So, now you know how the "magic" works; the next part of the book explores SQL for LINQ.

PART III

LINQ for Data

IN THIS PART

CHAPTER 13	Querying Relational Data with LINQ	265
CHAPTER 14	Creating Better Entities and Mapping Inheritance and Aggregation	289
CHAPTER 15	Joining Database Tables with LINQ Queries	309
CHAPTER 16	Updating Anonymous Relational Data	349
CHAPTER 17	Introducing ADO.NET 3.0 and the Entity Framework	383

This page intentionally left blank

CHAPTER 13

Querying Relational Data with LINQ

"If we could sell our experiences for what they cost us, we'd all be millionaires."

—Abigail Van Buren

Everything you have learned in the last 12 chapters can be leveraged here—extension methods, expression trees, Lambda Expressions, queries, joins, and everything else. The biggest difference is that you will learn how to get data from a new kind of provider, SQL Server.

Recall from Chapter 12, “Querying Outlook and Active Directory,” that Language INtegrated Query (LINQ) supports custom providers and these providers convert LINQ expression trees to the language of the provider. (In Chapter 12, this was demonstrated by using Active Directory as the provider.) In this chapter, a provider for Structured Query Language (SQL) already exists and ADO.NET objects are accessible via LINQ through the `IQueryable` interface.

LINQ can be used to query `DataSets` directly or you can use the `DataContext` and `Table<T>` (which implements `ITable`, `IQueryable`, and `IEnumerable`). The basic chore is to define a standard object-relational mapping (ORM) that maps a C# entity class to a table in a database. It’s not that much work because you can use the `SqlMetal` (for more on `SqlMetal` refer to the later section “`SqlMetal: Using the Entity Class Generator Tool`”) command-line utility or LINQ to SQL class designer to perform the ORM chore for you. This chapter looks at creating the ORM mapping, using `SqlMetal`, and the LINQ to SQL Class designer. This chapter begins by manually defining an ORM to see that it’s not that much work and then progresses to using the tools and exploring additional features of LINQ to SQL. The important thing to

IN THIS CHAPTER

- ▶ Defining Table Objects
- ▶ Connecting to Relational Data with `DataContext` Objects
- ▶ Querying `DataSets`
- ▶ `SqlMetal: Using the Entity Class Generator Tool`
- ▶ Using the LINQ to SQL Class Designer

keep in mind is that LINQ is completely compatible with ADO.NET 2.0, so you can use new LINQ features with existing ADO.NET code.

Defining Table Objects

Object-relational mapping is not a new concept. Defining an ORM might sound ominous but any time you define a custom class that maps to a database table (called an entity class), you are defining an ORM. The first thing you might be interested in is how much work is involved. The answer is not much.

You need the following information to make LINQ to SQL work:

- ▶ A database—you can use your old friend Northwind
- ▶ A reference to `System.Data.Linq` and that namespace added to your code with a `using` statement
- ▶ An instance of the `DataContext`, which connects to your database with a connection string
- ▶ A class that represents your entity and the `TableAttribute`, mapping the class to the table
- ▶ The `ColumnAttribute` to map properties to columns in the table

The rest is easy. You can use automatic properties. You don't have to use any of the name fields for the attributes, and you are ready to start using LINQ with SQL. The code in Listing 13.1 is a bare-bones, hand-coded ORM and LINQ over SQL example. Two interesting features are the assignment of the `Console.Out` stream to the `DataContext`'s `Log` property and the override `ToString` method. The `Log` property lets LINQ show you the queries that it's generating from your ORM, and the `ToString` method uses reflection to dump the `Customer` class' state.

LISTING 13.1 A Bare-Bones ORM and LINQ to SQL Example

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data;
using System.Data.Linq;
using System.Data.Linq.Mapping;
using System.Data.Common;
using System.Reflection;
```

```
namespace ContextAndTable
{
    class Program
    {
```

LISTING 13.1 Continued

```
private static readonly string connectionString =
    @"Data Source=.\SQLEXPRESS;AttachDbFilename=" +
    "\"C:\Books\Sams\LINQ\Northwind\northwnd.mdf\";" +
    "Integrated Security=True;Connect Timeout=30;User Instance=True";

static void Main(string[] args)
{
    DataContext customerContext = new DataContext(connectionString);
    Table<Customer> customers = customerContext.GetTable<Customer>();

    var startsWithA = from customer in customers
                      where customer.CustomerID[0] == 'A'
                      select customer;

    customerContext.Log = Console.Out;

    Array.ForEach(startsWithA.ToArray(), c => Console.WriteLine(c));
    Console.ReadLine();
}

[Table(Name="Customers")]
public class Customer
{
    [Column()]
    public string CustomerID{ get; set; }

    [Column()]
    public string CompanyName{ get; set; }

    [Column()]
    public string ContactName{ get; set; }

    [Column()]
    public string ContactTitle{ get; set; }

    [Column()]
}
```

LISTING 13.1 Continued

```
public string Address{ get; set; }

[Column()]
public string City{ get; set; }

[Column()]
public string Region{ get; set; }

[Column()]
public string PostalCode{ get; set; }

[Column()]
public string Country{ get; set; }

[Column()]
public string Phone{ get; set; }

[Column()]
public string Fax{ get; set; }

public override string ToString()
{
    StringBuilder builder = new StringBuilder();
    PropertyInfo[] props = this.GetType().GetProperties();

    // using array for each
    Array.ForEach(props.ToArray(), prop =>
        builder.AppendFormat("{0} : {1}", prop.Name,
            prop.GetValue(this, null) == null ? "<empty>\n" :
            prop.GetValue(this, null).ToString() + "\n");

    return builder.ToString();
}
```

Notice that the `Table` attribute indicates that the `Customer` class is mapped to the `Customers` table. Further notice that none of the named arguments were needed for the `Column` attribute.

In the `Program` class, a connection string was added. You'll need to change the connection string for your database. The `DataContext` plays a role similar to the connection class, and

you declare an instance of `Table<Customer>` and get the table data from `DataContext.GetTable`. The rest is a simple LINQ query and a few lines for displaying the results. The output is shown in Figure 13.1. The first part of Figure 13.1 is the `SELECT` statement written by LINQ and the rest is the output from the `Array.ForEach` statement.

```

file:///C:/Books/Addison Wesley/LINQ/SOURCE/Chapter 13/ContextAndTable/ContextAndTable/bin...
SELECTI [t@1].[CustomerID], [t@1].[CompanyName], [t@1].[ContactName], [t@1].[ContactTitle], [t@1].[Address], [t@1].[City], [t@1].[Region], [t@1].[PostalCode], [t@1].[Country], [t@1].[Phone], [t@1].[Fax]
FROM [Customer] AS [t@1]
WHERE UNICODE(CONVERT(NChar1),SUBSTRING([t@1].[CustomerID], @p1 + 1, 1)) = @p1
-- Ep1: Input Int <Size = 0; Prec = 0; Scale = 0> [0]
-- @p1: Input Int <Size = 0; Prec = 0; Scale = 0> [65]
-- Context: SqlProvider<Sql2005> Model: AttributedMetaModel Build: 3.5.21022.8

CustomerID : ALEKU
CompanyName : Alfreds Futterkiste
ContactName : Maria Anders
ContactTitle : Sales Representative
Address : Obere Str. 57
City : Berlin
Region :
PostalCode : 12209
Country : Germany
Phone : 030-0074321
Fax : 030-0076545

CustomerID : ANATR
CompanyName : Ana Trujillo Emparedados y helados
ContactName : Ana Trujillo
ContactTitle : Owner

```

FIGURE 13.1 The output from the `DataContext.Log` property and the `Customer` object state for the resultset.

Mapping Classes to Tables

A class that maps to a database table is called an *entity*. For LINQ, entities are decorated with `TableAttribute` and `ColumnAttribute`. The `ColumnAttribute` can be applied to any field or property, public, private, or internal, but only those elements of the entity with the `ColumnAttribute` will be persisted when LINQ saves changes back to the database.

The `ColumnAttribute` supports several named arguments, including `AutoSync`, `CanBeNull`, `DbType`, `Expression`, `IsDbGenerated`, `IsDiscriminator`, `IsPrimaryKey`, `IsVersion`, `Name`, `Storage`, and `UpdateCheck`. For example, `Name` is used to indicate the table column name. `Storage` can be used to indicate the underlying field name and LINQ will write to the field rather than through the property. `DbType` is one of the supported types in the `DbType` enumeration, for example `NChar`. `Enum.Parse` appears to be used to convert the `DbType` string argument to one of the `DbType` enumeration values. Using this new information, you can define the `Customer` class more precisely using a named argument for the `ColumnAttribute`, as shown in Listing 13.2.

LISTING 13.2 Providing Values for Named Arguments of the `ColumnAttribute` Class

```

[Table(Name="Customers")]
public class Customer
{
    private string customerID;
    private string companyName;
    private string contactName;
    private string contactTitle;
    private string address;
    private string city;

```

LISTING 13.2 Continued

```
private string region;
private string postalCode;
private string country;
private string phone;
private string fax;

[Column(Name="CustomerID", Storage="customerID",
    DbType="NChar(5)", CanBeNull=false)]
public string CustomerID
{
    get { return customerID; }
    set { customerID = value; }
}

[Column(Name="CompanyName", Storage="companyName",
    DbType="NVarChar(40)", CanBeNull=true)]
public string CompanyName
{
    get { return companyName; }
    set { companyName = value; }
}

[Column(Name="ContactName", Storage="contactName",
    DbType="NVarChar(30)")]
public string ContactName
{
    get { return contactName; }
    set { contactName = value; }
}

[Column(Name="ContactTitle", Storage="contactTitle",
    DbType = "NVarChar(30)")]
public string ContactTitle
{
    get { return contactTitle; }
    set { contactTitle = value; }
}

[Column(Name="Address", Storage="address",
    DbType = "NVarChar(60)")]

```

LISTING 13.2 Continued

```
public string Address
{
    get { return address; }
    set { address = value; }
}

[Column(Name="City", Storage="city",
    DbType = "NVarChar(15)")]
public string City
{
    get { return city; }
    set { city = value; }
}

[Column(Name = "Region", Storage = "region", DbType = "NVarChar(15)")]
public string Region
{
    get { return region; }
    set { region = value; }
}

[Column(Name="PostalCode", Storage="postalCode",
    DbType = "NVarChar(10)")]
public string PostalCode
{
    get { return postalCode; }
    set { postalCode = value; }
}

[Column(Name = "Country", Storage = "country", DbType = "NVarChar(15)")]
public string Country
{
    get { return country; }
    set { country = value; }
}

[Column(Name = "Phone", Storage = "phone", DbType = "NVarChar(24)")]
public string Phone
{
    get { return phone; }
    set { phone = value; }
}
```

LISTING 13.2 Continued

```
[Column(Name = "Fax", Storage = "fax", DbType = "NVarChar(24)")]
public string Fax
{
    get { return fax; }
    set { fax = value; }
}

public override string ToString()
{
    StringBuilder builder = new StringBuilder();
    PropertyInfo[] props = this.GetType().GetProperties();

    // using array for each
    Array.ForEach(props.ToArray(), prop =>
        builder.AppendFormat("{0} : {1}\n",
            prop.Name,
            prop.GetValue(this, null) == null ? "<empty>\n" :
            prop.GetValue(this, null).ToString() + "\n"));

    return builder.ToString();
}
}
```

For the `CustomerID` `Name` argument, we indicate the underlying table column name. The `Storage` named attribute is `customerID` (note the lowercase `c`), which means that LINQ will use the underlying field to set the value of the customer ID. The `DbType` field is an `NChar` up to five characters long. (The `DbType` string argument is not case sensitive.) And, finally, the `CanBeNull` argument is associated with the `NOT NULL` capability of SQL databases.

Table 13.1 contains a brief description for (or references to locations in this book for examples of) uses for the other named arguments of `ColumnAttribute`.

TABLE 13.1 Named Arguments for `ColumnAttribute`

Named Property	Description
<code>AutoSync</code>	Indicates when to synchronize this column with the database; for example, <code>AutoSync.OnInsert</code>
<code>CanBeNull</code>	Indicates whether a column can or cannot contain null values
<code>DbType</code>	Represents one of the <code>DbType</code> enumeration values
<code>Expression</code>	Describes how a computed column is derived, such as “ <code>UnitPrice + 1.0</code> ”

TABLE 13.1 Continued

IsDbGenerated	Indicates whether the column is auto-generated, such as auto-generated primary key columns marked as IDENTITY in the database
IsDiscriminator	Maps out inheritance hierarchies (refer to Chapter 14, “Creating Better Entities and Mapping Inheritance and Aggregation”)
IsPrimaryKey	Indicates if the column is a primary key
IsVersion	Indicates if the column is a database time stamp or version number
Name	Displays the column name
Storage	Identifies the field used to store the column/field value in the entity class
UpdateCheck	Indicates how to detect concurrency conflicts

Viewing the Query Text Generated by LINQ

If a `TextWriter` is assigned to the `DataContext.Log`, the `DataContext` will display information as the LINQ to SQL provider is doing its job. If you want to see the SQL text specifically, it can be requested from the `DataContext.GetCommand()` method. `GetCommand` returns a `DBCommand`, from which the SQL can be requested. Listing 13.3 contains a sample that uses LINQ to SQL for the Northwind.Order Details table and requests the `DBCommand.CommandText`.

LISTING 13.3 Requesting the Underlying Command Generated By the LINQ to SQL Provider

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data.Linq;
using System.Data.Linq.Mapping;
using System.Reflection;

namespace GetCommandDemo
{
    class Program
    {

        private static readonly string connectionString =
            "Data Source=BUTLER;Initial Catalog=Northwind;Integrated Security=True";

        static void Main(string[] args)
        {
            DataContext context = new DataContext(connectionString);
    }
}

```

LISTING 13.3 Continued

```
Table<OrderDetail> details = context.GetTable<OrderDetail>();  
  
Console.WriteLine("SELECT: {0}", context.GetCommand(details).CommandText);  
Console.WriteLine();  
  
var results = from detail in details  
              where detail.OrderID == 10248  
              select detail;  
  
  
Array.ForEach(results.ToArray(), d=>Console.WriteLine(d));  
Console.ReadLine();  
}  
}  
  
[Table(Name="Order Details")]  
public class OrderDetail  
{  
    [Column()]  
    public int OrderID{get; set;}  
    [Column()]  
    public int ProductID{get; set;}  
  
    [Column()]  
    public decimal UnitPrice{get; set;}  
    [Column()]  
    public Int16 Quantity{get; set;}  
    [Column()]  
    public float Discount{get; set;}  
  
  
    public override string ToString()  
{  
        StringBuilder builder = new StringBuilder();  
        PropertyInfo[] props = this.GetType().GetProperties();  
  
        // using array for each  
        Array.ForEach(props.ToArray(), prop =>  
            builder.AppendFormat("{0} : {1}", prop.Name,  
                prop.GetValue(this, null) == null ? "<empty>\n" :
```

LISTING 13.3 Continued

```
prop.GetValue(this, null).ToString() + "\n"));

return builder.ToString();
}
}

}
```

The statement

```
Console.WriteLine("SELECT: {0}", context.GetCommand(details).CommandText);
```

returns the SQL SELECT generated by the LINQ to SQL provider. The results from the sample instance of the code run produces the following output:

```
SELECT [t0].[OrderID], [t0].[ProductID], [t0].[UnitPrice], [t0].[Quantity],
[t0].[Discount]
FROM [Order Details] AS [t0]
```

13

Connecting to Relational Data with DataContext Objects

So far, the examples have used the `DataContext` class directly. You can inherit from `DataContext` and embed information such as the connection string into the new class. This approach is a recommended practice and makes LINQ to SQL even easier to use. Listing 13.4 is a revision of Listing 13.3. In Listing 13.4, a generalized `DataContext` for the Northwind database has been added.

LISTING 13.4 Inheriting from DataContext and Embedding the Connection String, Resulting in a Strongly Typed DataContext (In This Example, the Northwind Database)

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data.Linq;
using System.Data.Linq.Mapping;
using System.Reflection;

namespace DataContextChildClass
{
    public class Northwind : DataContext
    {
```

LISTING 13.4 Continued

```
private static readonly string connectionString =
    "Data Source=BUTLER;Initial Catalog=Northwind;Integrated Security=True";
public Northwind() : base(connectionString)
{
}
}

class Program
{
    static void Main(string[] args)
    {
        Northwind context = new Northwind();
        Table<OrderDetail> details = context.GetTable<OrderDetail>();

        Console.WriteLine("SELECT: {0}", context.GetCommand(details).CommandText);
        Console.WriteLine();

        var results = from detail in details
                     where detail.OrderID == 10248
                     select detail;

        Array.ForEach(results.ToArray(), d=>Console.WriteLine(d));
        Console.ReadLine();
    }
}

[Table(Name="Order Details")]
public class OrderDetail
{
    [Column()]
    public int OrderID{get; set;}
    [Column()]
    public int ProductID{get; set;}

    [Column()]
    public decimal UnitPrice{get; set;}
    [Column()]
    public Int16 Quantity{get; set;}
    [Column()]
    public float Discount{get; set;}
```

LISTING 13.4 Continued

```
public override string ToString()
{
    StringBuilder builder = new StringBuilder();
    PropertyInfo[] props = this.GetType().GetProperties();

    // using array for each
    Array.ForEach(props.ToArray(), prop =>
        builder.AppendFormat("{0} : {1}", prop.Name,
            prop.GetValue(this, null) == null ? "<empty>\n" :
            prop.GetValue(this, null).ToString() + "\n"));
    return builder.ToString();
}
```

In the revised Listing 13.4, there is a new class `Northwind`. `Northwind` inherits from `DataContext` with the connection string embedded in the `Northwind` class, resulting in a named, strongly typed `DataContext`.

The `DataContext` is the starting point for LINQ to SQL. One instance of a `DataContext` represents a “unit of work” for related operations. The `DataContext` is a lightweight class and is usually used at the method scope level, rather than creating one instance of the `DataContext` and reusing it.

A connection can be reused in a `DataContext`, for example, when using transactions. Simply inherit from the `DataContext` and embed a `SqlConnection` object in the class as well as the connection string. Then initialize the base class—the `DataContext`—with the `SqlConnection` object instead of the connection string.

Querying DataSets

Visual Studio and C# provide compile-time syntax checking, static typing, and IntelliSense support. By writing queries in LINQ/C# instead of another query language such as SQL, you get the support of all of these Integrated Development Environment (IDE) and language features. You can still write stored procedures and use your favorite query editor, but you also have the choice of writing the queries against DataSets in your C# code.

LINQ to DataSet is not intended to replace ADO.NET 2.0 code. Rather, the LINQ to DataSet code sits on top of and works in conjunction with the ADO.NET 2.0 code and might be useful in scenarios where data is coming from multiple data providers, is querying locally, and is performing reporting and analysis. Figure 13.2 illustrates the relationship between LINQ to DataSet and ADO.NET.

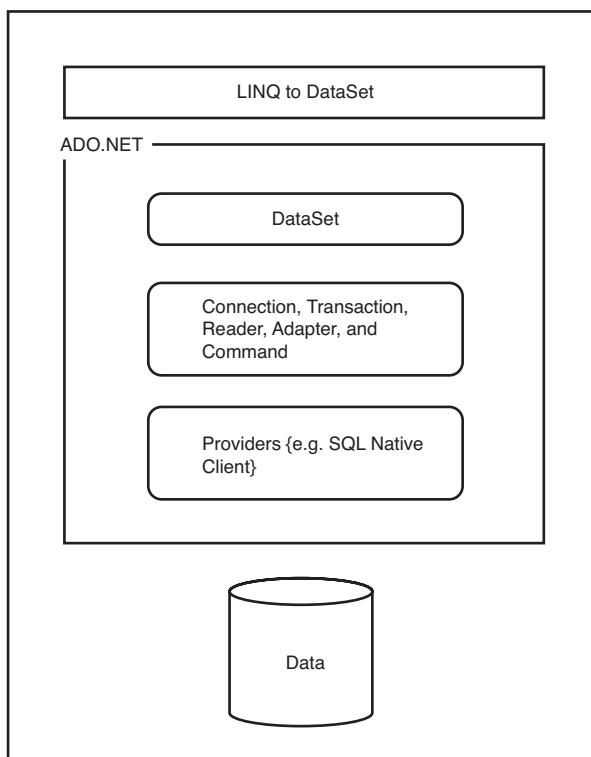


FIGURE 13.2 The LINQ to DataSet technology stack.

Support for DataSets with LINQ is exposed primarily through the `DataRowExtensions` and the `DataTableExtensions` classes. `DataRowExtensions` introduces the extension methods `Field` and `SetField`, and the `DataTableExtensions` class introduces the extension methods `AsEnumerable`, `AsQueryable`, and `CopyToDataTable`.

Selecting Data from a DataTable

The key to LINQ to DataSet is using the `DataTableExtension` and `DataRowExtension` methods. Table extension methods yield a queryable sequence, and row extensions provide access to field-level data. Listing 13.5 demonstrates plain vanilla ADO.NET followed by a LINQ to DataSet query that uses the `AsEnumerable` extension method. (The LINQ query is shown in bold font.)

LISTING 13.5 LINQ to DataSet, Querying a DataTable via the AsEnumerable Extension Method

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
```

LISTING 13.5 Continued

```
using System.Data;
using System.Data.SqlClient;

namespace LINQToDataSetSimple
{
    class Program
    {
        private static readonly string connectionString =
            "Data Source=BUTLER;Initial Catalog=Northwind;Integrated Security=True";

        static void Main(string[] args)
        {
            DataSet data = new DataSet();
            using(SqlConnection connection = new SqlConnection(connectionString))
            {
                connection.Open();
                SqlCommand command =
                    new SqlCommand("SELECT * FROM Suppliers", connection);
                command.CommandType = CommandType.Text;
                SqlDataAdapter adapter = new SqlDataAdapter(command);
                adapter.Fill(data, "Suppliers");
            }

            DataTable supplierTable = data.Tables["Suppliers"];

            IEnumerable<DataRow> suppliers =
                from supplier in supplierTable.AsEnumerable()
                select supplier;

            Console.WriteLine("Supplier Information");
            foreach(DataRow row in suppliers)
            {
                Console.WriteLine(row.Field<string>("CompanyName"));
                Console.WriteLine(row.Field<string>("City"));
                Console.WriteLine();
            }
        }

        Console.ReadLine();
    }
}
```

LISTING 13.5 Continued

```
    }  
}  
}
```

After the DataSet is populated using ADO.NET code, a single table is obtained from the DataSet's Tables collection. The type that stores the query results can be defined as an anonymous type or the type it is, `IEnumerable<DataRow>`. The `from` clause includes a range—supplier here—and the sequence that is obtained from the `AsEnumerable` extension method of the `DataTable`. In a nutshell for the basic query syntax, use `AsEnumerable` to get a sequence that LINQ can work with.

The rest of the example displays some of the results of the query. Notice that the data is obtained from the fields using the `Field` generic extension method.

Querying the DataTable with a Where Clause

The fundamental LINQ grammar of LINQ queries doesn't change because the source of data is an ADO.NET DataSet. The only change relative to DataSets is that you call extension methods to get at the underlying data. For example, to use a field value in a `where` clause, you need to use the `Field` extension method to request the field value. Listing 13.6 uses plain vanilla ADO.NET code to get data from the Northwind Order Details and Products table with a join. The LINQ query uses the resulting view via the DataSet's `DataTable` and uses the `Field` extension method on the range variable and compares the `Discount` with 0.10 (or Discounts greater than 10%). (To help you spot the LINQ query in the example, a bold font is used.)

LISTING 13.6 Using the Field Extension Method on the Detail Range Variable to Filter By Discounts > 10%

```
using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Text;  
using System.Data;  
using System.Data.SqlClient;  
  
namespace LinqToDataSetWithWhere  
{  
    class Program  
    {  
        private static readonly string connectionString =  
            "Data Source=BUTLER;Initial Catalog=Northwind;Integrated Security=True";  
    }  
}
```

LISTING 13.6 Continued

```
static void Main(string[] args)
{
    string sql = "SELECT * FROM [Order Details] od " +
        "INNER JOIN Products p on od.ProductID = od.ProductID";

    DataSet data = new DataSet();
    using(SqlConnection connection = new SqlConnection(connectionString))
    {
        connection.Open();
        SqlCommand command =
            new SqlCommand(sql, connection);
        command.CommandType = CommandType.Text;
        SqlDataAdapter adapter = new SqlDataAdapter(command);
        adapter.Fill(data);
    }

    DataTable detailTable = data.Tables[0];

    IEnumerable<DataRow> details =
        from detail in detailTable.AsEnumerable()
        where detail.Field<float>("Discount") > 0.10f
        select detail;

    Console.WriteLine("Big-discount orders");
    foreach(DataRow row in details)
    {
        Console.WriteLine(row.Field<int>("OrderID"));
        Console.WriteLine(row.Field<string>("ProductName"));
        Console.WriteLine(row.Field<decimal>("UnitPrice"));
        Console.WriteLine(row.Field<float>("Discount"));
        Console.WriteLine();
    }

    Console.ReadLine();
}
}
```

Using Partitioning Methods

The partitioning methods, `Skip` and `Take`, work directly on sequences as described in Chapter 6, “Using Standard Query Operators,” in the section “Partitioning with `Skip` and `Take`.” For example, if you add the following line

```
details = details.Take(25);
```

after the LINQ query in Listing 13.6, then (per the fragment) you can partition the result-set by grabbing the first five elements in the sequence.

Rather than going through all of the basic capabilities again, it is worth noting that the LINQ capabilities can all be used with LINQ to DataSets, too. Aggregation, partitioning, filtering, generation operators, concatenation, and quantifiers are all supported.

Sorting Against DataTables

To sort a query, you add the `orderby` clause as before. The only change for sorting, when working with DataSets, is that you have to use the extension method `Field` on the range variable to obtain the field value to sort by. The following fragment can be plugged in to Listing 13.6 to sort the `DataTable`’s data by the `ProductID`.

```
IEnumerable<DataRow> details =
    from detail in detailTable.AsEnumerable()
    where detail.Field<float>("Discount") > 0.10f
    orderby detail.Field<int>("ProductID")
    select detail;
```

Defining a Join with DataSets

Joins, of course, are supported by LINQ to DataSets, but you need to use table and data row extensions in the query. You can use the data table extensions to get the sequence and the data row extensions to get the fields (because the correlation occurs at the field level).

A brand-new listing, Listing 13.7, was created to demonstrate the join and blend in a few techniques you might find useful. Listing 13.7 shows that you can send multiple `select` statements to SQL Server in one call by separating the queries with a semicolon. (This is also how malicious SQL injection attacks work.) The LINQ join and a projection create a new type by shaping elements from each of the source sequences.

LISTING 13.7 A LINQ to DataSet Example That Demonstrates Multiple SQL Statements in One ADO.NET Call, Join for DataSets, and a Projection Using Elements from Both Sequences in the Join

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data;
using System.Data.SqlClient;

namespace LinqToDataSetWithJoin
{
    class Program
    {
        private static readonly string connectionString =
            "Data Source=BUTLER;Initial Catalog=Northwind;Integrated Security=True";

        static void Main(string[] args)
        {
            const string sql = "SELECT * FROM Orders;" +
                "SELECT * FROM [Order Details];";

            DataSet data = new DataSet();
            using(SqlConnection connection = new SqlConnection(connectionString))
            {
                connection.Open();
                SqlCommand command =
                    new SqlCommand(sql, connection);
                command.CommandType = CommandType.Text;
                SqlDataAdapter adapter = new SqlDataAdapter(command);
                adapter.Fill(data);
            }

            DataTable orders = data.Tables[0];
            DataTable orderDetails = data.Tables[1];

            var orderResults =
                from order in orders.AsEnumerable()
                join detail in orderDetails.AsEnumerable()
```

LISTING 13.7 Continued

```
on order.Field<int>"("OrderID")
equals detail.Field<int>"("OrderID")
select new {
    OrderID=order.Field<int>"("OrderID"),
    CustomerID=order.Field<string>"("CustomerID"),
    ProductID=detail.Field<int>"("ProductID"),
    UnitPrice=detail.Field<decimal>"("UnitPrice"),
    Quantity=detail.Field<Int16>"("Quantity"),
    Discount=detail.Field<float>"("Discount")};

Console.WriteLine("Orders & Details");
foreach(var result in orderResults)
{
    Console.WriteLine("Order ID: {0}", result.OrderID);
    Console.WriteLine("Customer ID: {0}", result.CustomerID);
    Console.WriteLine("Product ID: {0}", result.ProductID);
    Console.WriteLine("Unit Price: {0}", result.UnitPrice);
    Console.WriteLine("Quantity: {0}", result.Quantity);
    Console.WriteLine("Discount: {0}", result.Discount);
    Console.WriteLine();
}

Console.ReadLine();
}
}
}
```

Because the LINQ query defines a projection, the return type is an `IEnumerable<T>`, where `T` is an anonymous type derived from the projection not `IEnumerable<DataRow>`. The `from` clause defines the first range (`order`) and the `join` clause defines the second range (`detail`) and the correlation on `OrderID` by using the `Field` extension method. The example uses an equijoin—`join..in..on..equals`—but you can derive a nonequijoin by correlating data with a `where` clause. Finally, the anonymous type is a projection composed of `Orders.OrderID`, `Orders.CustomerID`, and the remainder of the fields from the Order Details table.

For more information on joins in general, check out Chapters 15 and 16, “Joining Database Tables with LINQ Queries” and “Updating Anonymous Relational Data,” respectively, which cover stored procedures and transactions used in conjunction with LINQ.

SqlMetal: Using the Entity Class Generator Tool

SqlMetal is an external Microsoft utility installed with Visual Studio 2008 in [drive:\]Program Files\Microsoft SDKs\Windows\v6.0A\bin that is used to generate .dbml (*database markup language*) files or ORM source files for LINQ to SQL. .dbml files are Extensible Markup Language (XML) files that contain information describing the schema of the thing defined. (The XML context in the .dbml file looks a lot like the ColumnAttributes we added to the entity classes earlier.)

The following command line uses SqlMetal.exe to generate a DataContext and entity classes for the entire Northwind database, generating a DataContext class and an entity class for each of the tables in the database.

```
SqlMetal /server:butler /database:Northwind /code:northwind.cs
```

There are several options for SqlMetal. The server, database, username, password, connection string, and timeout values can be expressed as command-line options. There are also command-line switches for extracting views, functions, and stored procedures. Output can be created as a map, dbml, or source code file. You can express the generated code language or let the tool infer the language from the extension of the source code file (the /code:*filename.ext parameter*). In addition to the language switch, a namespace, the name of the data context class, an entity base class (to inherit from), pluralization of entity class names, and whether the classes are serializable can all be expressed on the command line.

The previous SqlMetal statement refers to the server “butler,” the Northwind database, and generates all of the output as C# in a file named northwind.cs. As is, the statement will generate a DataContext called Northwind and an entity (or table-mapped) class for every table in the Northwind database but nothing for the functions, views, or stored procedures (or “sprocs,” in technical jargon).

Using the LINQ to SQL Class Designer

The LINQ to SQL Class designer is an integrated aspect of Visual Studio. The class designer is a designer in Visual Studio. A designer is some code that has the ability to interact with Visual Studio in some specific way. In this instance, the designer has the ability to drag and drop tables, stored procedures, and functions to a graphical user interface (GUI) representation of a Database Markup Language (DBML) file and have those elements converted to XML and source code. The class designer is like SqlMetal integrated in a visual way into Visual Studio.

For example, if you create a new project and add a new “LINQ to SQL Classes” item from the Add New Item dialog box (see Figure 13.3), then Visual Studio generates a .dbml file, a .cs file, a .dbml.layout file, and a .designer.cs file to your project. These files represent the designer arrangement, XML describing the schema of the elements dragged to the designer, and the code that contains the DataContext and entity classes of the elements on the designer.

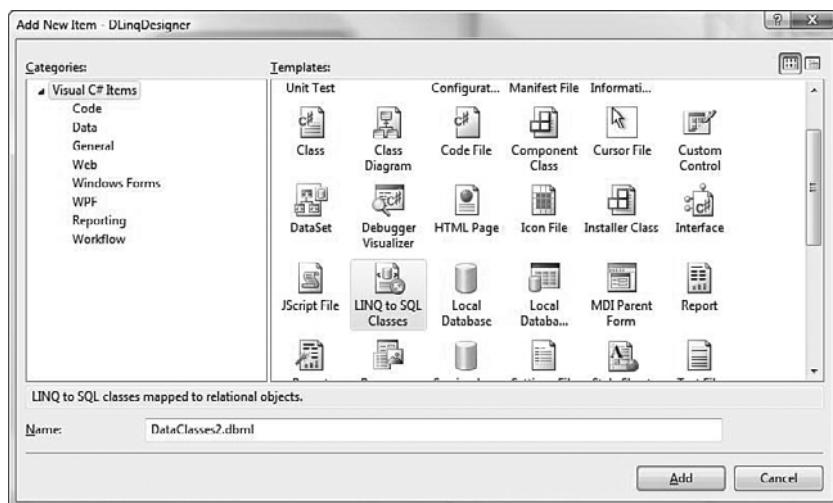


FIGURE 13.3 Add a “LINQ to SQL Classes” element to your project to code generate the **DataContext** and entity class (or classes) to your project.

Figure 13.4 illustrates what the LINQ to SQL Class designer will look like as elements are added. Adding elements to the designer generates a **DataContext** and entities for the items added to the designer, and the designer supports adding functions and stored procedures, which are represented as independent classes with properties representing the resultset of the function or query.

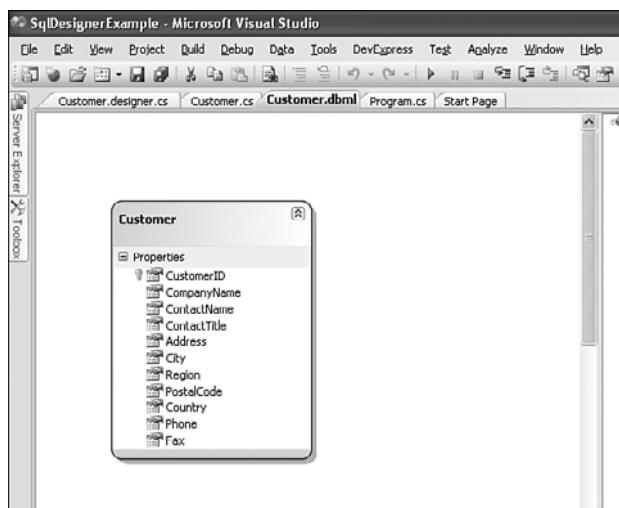


FIGURE 13.4 A representation of the LINQ to SQL Class designer in Visual Studio.

Summary

LINQ for data is supported in two basic ways. You can use LINQ to DataSets, which uses extension methods to make table data and rows and fields accessible in LINQ queries. No preparation work is necessary—this is pure ADO.NET coding. The other way to use LINQ with data is to use LINQ to SQL. LINQ to SQL requires that you create an ORM. In this context, an ORM is a `DataContext` class representing the database and classes decorated with `TableAttribute` and the elements that map to database columns decorated with `ColumnAttribute`.

Using LINQ to SQL results in cleaner code, and you can use SqlMetal or the LINQ to SQL Class designer to generate the ORM plumbing. (In addition, a new version of ADO.NET is coming, the ADO.NET Entity Framework. Because the entity framework is based on LINQ, it is introduced in Chapter 17, “Introducing ADO.NET 3.0 and the Entity Framework.”)

This page intentionally left blank

CHAPTER 14

Creating Better Entities and Mapping Inheritance and Aggregation

“The man who has no imagination has no wings.”

—Muhammad Ali

New technologies that solve everyday problems are sometimes suspect. New technologies that are found to address everyday problems superficially are useless except to hobbyists. LINQ to SQL supports everyday problems like reading and updating data as well as expressing inheritance hierarchies and aggregational and compositional relationships.

In this chapter, you learn to use nullable types for columns that allow null values. You will learn how to map inheritance hierarchies with LINQ to SQL. You will read about and learn how to represent master-detail (one-to-one and one-to-many) relationships, and you will learn how to use LINQ to SQL to generate databases on the fly and populate tables in these databases. (Combined with Chapter 16, “Updating Anonymous Relational Data,” which includes stored procedures, transactions, modifications, deletes, and concurrency resolution, you will have everything you need to manage databases with LINQ.)

Defining Better Entities with Nullable Types

Everyday plumbing code is dull. It’s dull to read and dull to write. The solution to avoid writing dull parts of code is: Don’t write it (at least any more than is necessary). There are all kinds of tricks inside (and outside of) Visual Studio that help avoid writing plumbing code. Property getters and setters are mostly plumbing. Use automatic properties if

IN THIS CHAPTER

- ▶ Defining Better Entities with Nullable Types
- ▶ Mapping Inheritance Hierarchies for LINQ to SQL
- ▶ Adding EntitySet Classes as Properties
- ▶ Creating Databases with LINQ to SQL

your properties don't need any extra lines of code. Use code snippets for routine copy-and-paste code. Consider CodeRush and Refactor—metaprogramming tools from DevExpress—that write code for you with just one or two keystrokes or menu clicks. In addition, you can write macros using EnvDTE namespace, create wizards, and project templates, and you can use code generators and emitters. (Many of these topics are covered in books, including some of my other titles and columns, and by other authors. You can check out *Visual Basic .NET Power Coding* by me from Addison Wesley or my columns on codeguru.com, devsource.com, or informit.com.)

One kind of plumbing code that is everywhere is the infamous check for `DBNull` and null when reading database columns. This is dull plumbing code. You can eliminate the need to write checks for null and `DBNull` by using nullable types. Nullable types are explicitly defined with the generic type `Nullable<T>` where `T` is the underlying data type or using the convenience notation `?` as a suffix to the data type. For instance, “`int?`” defines a nullable integer type. Only value types can be defined as nullable types. Reference types like strings are already nullable.

When you define a type as nullable, it can be assigned a null or any other proper value for that type. For instance, a nullable integer can be null or any valid whole number plus or minus about 2.1 billion. Nullable types can be checked with the null coalescing operator `??`. For example,

```
int? x = y ?? 0;
```

means that `x` can be assigned the value of `y`, but if `y` is null, `x` will be assigned the value `0`.

By using nullable types, you can assign a null field value from the database directly to the nullable field or property in the entity class. The minor sticking point is how to know whether to make an entity nullable. The easy answer is that all value type class fields and properties can be made nullable, or you can look in the database schema. If `IS NULL` or `Allow Nulls` is indicated for a column in the database table (see Figure 14.1), the correlating field in an entity class needs to be null, too. Of course, if any fields are missed, you are likely to see the exception shown in Figure 14.2 (`InvalidOperationException`) when LINQ attempts to assign a null value to a non-nullable property or field.

Listing 14.1 demonstrates how you can match nullable properties to nullable columns in the `Orders` table of the Northwind database. Notice that both the `Nullable<T>` type and the shorthand variant (with `type?`) are demonstrated. In addition, notice that the `ColumnAttribute`'s named argument `CanBeNull` is set to `true` for nullable types.

LISTING 14.1 Demonstrating Nullable Types for LINQ to SQL Entity Classes

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data.Linq;
using System.Data.Linq.Mapping;
```

LISTING 14.1 Continued

```
using System.Reflection;

namespace NullableClassElements
{
    class Program
    {
        static void Main(string[] args)
        {
            Northwind northwind = new Northwind();
            Table<Order> orders = northwind.GetTable<Order>();

            northwind.Log = Console.Out;
            Array.ForEach(orders.ToArray(), t=>Console.WriteLine(t));
            Console.ReadLine();
        }
    }

    [Database(Name = "Northwind")]
    public class Northwind : DataContext
    {

        private static readonly string connectionString =
            @"Data Source=.\SQLEXPRESS;AttachDbFilename=" +
            "\"C:\Books\Sams\LINQ\Northwind\northwnd.mdf\";" +
            "Integrated Security=True;Connect Timeout=30;User Instance=True";

        public Northwind() : base(connectionString) { }

    }

    [Table(Name="Orders")]
    public class Order
    {
        [Column(IsPrimaryKey=true)]
        public int OrderID{get; set;}

        [Column()]
        public string CustomerID{get; set;}

        [Column(CanBeNull=true)]
        public int? EmployeeID{get; set;}

        [Column(CanBeNull = true)]
        public Nullable<DateTime> OrderDate{get; set;}
    }
}
```

LISTING 14.1 Continued

```
[Column(CanBeNull = true)]
public DateTime? RequiredDate{get; set;}

[Column(CanBeNull = true)]
public DateTime? ShippedDate{get; set;}

[Column(CanBeNull = true)]
public int? ShipVia{get; set;}

[Column(CanBeNull = true)]
public decimal? Freight{get; set;}

[Column(CanBeNull = true)]
public string ShipName{get; set;}

[Column(CanBeNull = true)]
public string ShipAddress{get; set;}

[Column(CanBeNull = true)]
public string ShipCity{get; set;}

[Column(CanBeNull = true)]
public string ShipRegion{get; set;}

[Column(CanBeNull = true)]
public string ShipPostalCode{get; set;}

[Column(CanBeNull = true)]
public string ShipCountry{get; set;}

public override string ToString()
{
    StringBuilder builder = new StringBuilder();
    PropertyInfo[] props = this.GetType().GetProperties();

    // using array for each
    Array.ForEach(props.ToArray(), prop =>
        builder.AppendFormat("{0} : {1}", prop.Name,
            prop.GetValue(this, null) == null ? "<empty>\n" :
            prop.GetValue(this, null).ToString() + "\n"));
    return builder.ToString();
}
```

Column Name	Data type	Allow Nulls
OrderID	int	<input type="checkbox"/>
CustomerID	nchar(5)	<input checked="" type="checkbox"/>
EmployeeID	int	<input checked="" type="checkbox"/>
OrderDate	datetime	<input checked="" type="checkbox"/>
RequiredDate	datetime	<input checked="" type="checkbox"/>
ShippedDate	datetime	<input checked="" type="checkbox"/>
ShipVia	int	<input checked="" type="checkbox"/>
Freight	money	<input checked="" type="checkbox"/>
ShipName	nvarchar(40)	<input checked="" type="checkbox"/>
ShipAddress	nvarchar(60)	<input checked="" type="checkbox"/>
ShipCity	nvarchar(15)	<input checked="" type="checkbox"/>
ShipRegion	nvarchar(15)	<input checked="" type="checkbox"/>
ShipPostalCode	nvarchar(10)	<input checked="" type="checkbox"/>
ShipCountry	nvarchar(15)	<input checked="" type="checkbox"/>

FIGURE 14.1 Elements of any schema that permits null values should be defined as nullable types in the correlating entity class.

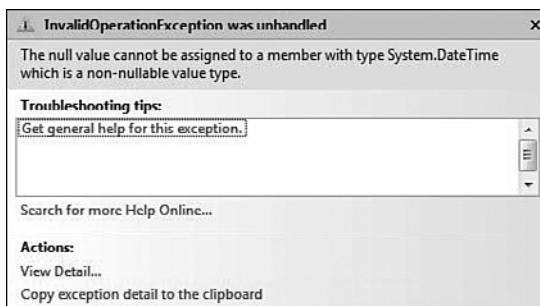


FIGURE 14.2 If LINQ attempts to assign a null value to any field that does not permit nulls, an InvalidOperationException will be thrown by the framework.

If you want to see the more verbose version of the Orders class, you can run SqlMetal against the Northwind database. The following command is another variation of the SqlMetal command line and it will generate the proper nullable types and attribute applications for you:

```
sqlmetal "C:\Books\Sams\LINQ\Northwind\northwnd.mdf" /code:c:\temp\ wnd.cs
```

The preceding SqlMetal command uses the .mdf file—the literal database file—as the input and generates code for the `DataContext` and all of the tables, including tables with null columns.

Mapping Inheritance Hierarchies for LINQ to SQL

Inheritance is one facet of the object-oriented world that is easy to overdo. Deep inheritance hierarchies make a framework hard to understand and even harder to use. There is no perfect rule for how much inheritance is too much but experienced developers know it when they see it.

One kind of inheritance that isn't bandied about too much is how inheritance hierarchies are represented in a persisted state. For our purposes, let's focus on how LINQ to SQL manages inheritance.

LINQ to SQL supports the flattened, single-table mapping approach. Assume you have one base class and two subclasses. *In toto*, single-table mapping means that one table has the union of all of the persisted properties for all three classes without repetition. To facilitate identifying which class is represented, some kind of key is used. To handle elements not used by a particular class, that column contains a null. Single-table mapping has good performance heuristics because it is not necessary to perform joins or multitable scans to get all of the data.

Suppose, for instance, that you want to define a hierarchy of kinds of people for a physician. You might define a Person, Patient, and VendorContact class. (You could also define Physician, PhysicianAssistant, MedicalAssistant, OfficeWorker, and whatever classes make sense, but three classes are enough to illustrate inheritance mapping.) Further, assume that Person contains a unique identifier and first and last names. The Patient class might contain gender, age, and date-of-last-visit, and the VendorContact might contain title and company ID. A class diagram can be depicted, as shown in Figure 14.3, and the singly mapped table would contain a column for each of the datum. (Listing 14.2 contains the schema definition copied from the Table Designer in Visual Studio 2008.)

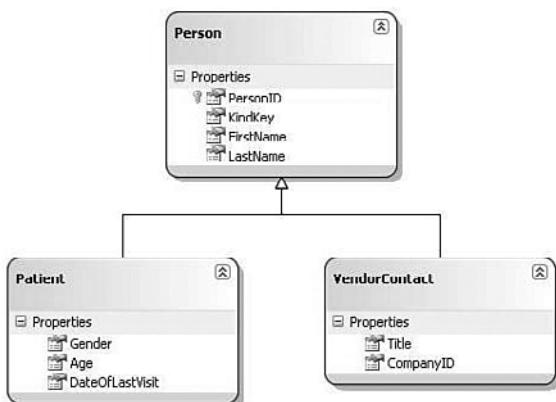


FIGURE 14.3 A visual depiction of the class relationship between the base class Person and subclasses Patient and VendorContact.

LISTING 14.2 The Schema Definition (for Person) of a Singly Mapped Table Describing the Persistence Data for the Inheritance Hierarchy of Person, Patient, and VendorContact

```
PersonID      int      Unchecked
KindKey       char(1)   Unchecked
FirstName     nvarchar(15)  Unchecked
LastName      nvarchar(15)  Unchecked
Gender        nchar(1)   Checked
Age           int      Checked
DateOfBirth   datetime  Checked
Title         nvarchar(50)  Checked
CompanyID    int      Checked
```

The Checked/Unchecked value is indicative of the Allow nulls column in the designer. Notice that everything except the columns representing the base class permits nulls. The reason for this is that, depending on which class a row represents, it may or may not make sense to use some columns, so you are hedging for flexibility here. Finally, the KindKey column is used as a sort of type identifier to indicate the class type that the row-datum represents.

Listing 14.3 demonstrates inheritance mapping. The `InheritanceMappingAttribute` is added to the base class along with the `TableAttribute`. The `InheritanceMappingAttribute` indicates all of the types that might be represented by each data row. The code-named arguments are mapped to a value in the base class that is the discriminator (`[Column(IsDiscriminator=true)]`). The discriminator is used to indicate the type of object represented by the data row.

Each of the subclasses inherits the attributes from the base class. The `ColumnAttribute` is used to map the underlying columns and you only need to have a column appear one time in just one of the classes in the hierarchy. When you request the table from the `DataContext`, request the singly mapped table, in this case, the `Person` table described in Listing 14.2. LINQ will read the `Person` table and the discriminator and figure out the exact type in the hierarchy of `Person`, `Patient`, and `VendorContact` to add to the collection of rows. That is, you request `Table<Person>` from the `DataContext` and LINQ figures out from the discriminator if it should create `Person`, `Patient`, or `VendorContact` objects. Here is Listing 14.3.

LISTING 14.3 Mapping Inheritance Hierarchy with a Single Table and a Discriminator Column That Tells LINQ What Kind of Parent or Child Object to Create

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data.Linq;
using System.Data.Linq.Mapping;
using System.Reflection;
```

LISTING 14.3 Continued

```
namespace InheritanceHierarchy
{
    class Program
    {
        static void Main(string[] args)
        {
            Inheritance inherit = new Inheritance();
            Table<Person> people = inherit.GetTable<Person>();

            Array.ForEach(people.Where(
                person => person.GetType()
                == typeof(Patient)).ToArray(), r => Console.WriteLine(r));
            Console.ReadLine();
        }
    }

    [Database(Name = "Inheritance")]
    public class Inheritance : DataContext
    {
        private static readonly string connectionString =
            "Data Source=.\SQLExpress;Initial Catalog=Inheritance;" +
            "Integrated Security=True;Pooling=False";

        public Inheritance()
            : base(connectionString)
        {
            this.Log = Console.Out;
        }

        public Table<Patient> GetPatients()
        {
            return this.GetTable<Patient>();
        }

        public Table<VendorContact> GetVendors()
        {
            return this.GetTable<VendorContact>();
        }
    }

    [Table(Name="Person")]
    [InheritanceMapping(Code = "P", Type = typeof(Person), IsDefault = true)]
    [InheritanceMapping(Code = "A", Type = typeof(Patient))]
    [InheritanceMapping(Code = "V", Type = typeof(VendorContact))]
    public class Person
```

LISTING 14.3 Continued

```
{  
    [Column(IsPrimaryKey = true)]  
    public int PersonID { get; set; }  
  
    [Column(IsDiscriminator = true)]  
    public char KindKey { get; set; }  
  
    [Column()]  
    public string FirstName { get; set; }  
  
    [Column()]  
    public string LastName { get; set; }  
  
    public override string ToString()  
    {  
        StringBuilder builder = new StringBuilder();  
        builder.AppendFormat("Type: {0}\n", this.GetType().Name);  
  
        PropertyInfo[] props = this.GetType().GetProperties();  
  
        // using array for each  
        Array.ForEach(props.ToArray(), prop =>  
            builder.AppendFormat("{0} : {1}", prop.Name,  
                prop.GetValue(this, null) == null ? "<empty>\n" :  
                prop.GetValue(this, null).ToString() + "\n"));  
  
        return builder.ToString();  
    }  
}  
  
public class Patient : Person  
{  
    [Column()]  
    public char Gender { get; set; }  
  
    [Column()]  
    public int Age { get; set; }  
  
    [Column(CanBeNull = true)]  
    public DateTime? DateOfLastVisit { get; set; }  
}  
  
public class VendorContact : Person  
{  
    [Column()]
```

LISTING 14.3 Continued

```

public string Title { get; set; }

[Column()]
public int CompanyID { get; set; }
}

}

```

Creating Inheritance Mappings with the LINQ to SQL Designer

If you find creating inheritance hierarchies a little confusing at first, you can use the LINQ to SQL Object Relational Designer (see Chapter 13, “Querying Relational Data with LINQ,” for more information) to map the hierarchy. To begin, assume you have a single table where each row might represent one of several classes that are all related to each other through inheritance. Further, assume one column in the table contains a value that indicates what kind of object the row represents. Then, you can use the designer by following these steps:

1. Assuming you have a project, add the LINQ to SQL Classes item template from the Add New Item dialog. (This step adds a .dbml file, a .dbml.layout file, a .designer.cs file, and a .cs file to your project.)
2. Drag the table containing the singly mapped persisted objects to the left side of the designer.
3. Drag the same table to the designer *n* more times, an additional time for each child type.
4. Rename all but one copy to the class name you’d like the designer to generate.
5. Open the Toolbox and click the Inheritance toolbox item (see Figure 14.4), and then click the subclass and connect it to the base class (for example, Patient to Person).
6. In each class, delete all of the columns that you don’t want to be defined in that class. Each column should appear only once between parent and child (or base and sub, if you prefer) because child classes inherit parent class members.
7. Click on the inheritance line and open the Properties window. Specify the Discriminator in the properties window.
8. Set the Derived Class Discriminator Value. This is something you pick and should be one of the values in the actual table. (In the example, the discriminator is the KindKey column.)
9. Set the Base Class Discriminator Value (see Figure 14.5).
10. Specify the Inheritance Default, which is used when loading rows that do not match any defined code.
11. Repeat the steps for all subclasses in the hierarchy.

The designer adds the generated `DataContext` and class definitions with `TableAttributes` and `ColumnAttributes` to the `.designer.cs` file as partial classes. You can add custom code to the `.cs` file containing additional partial classes.

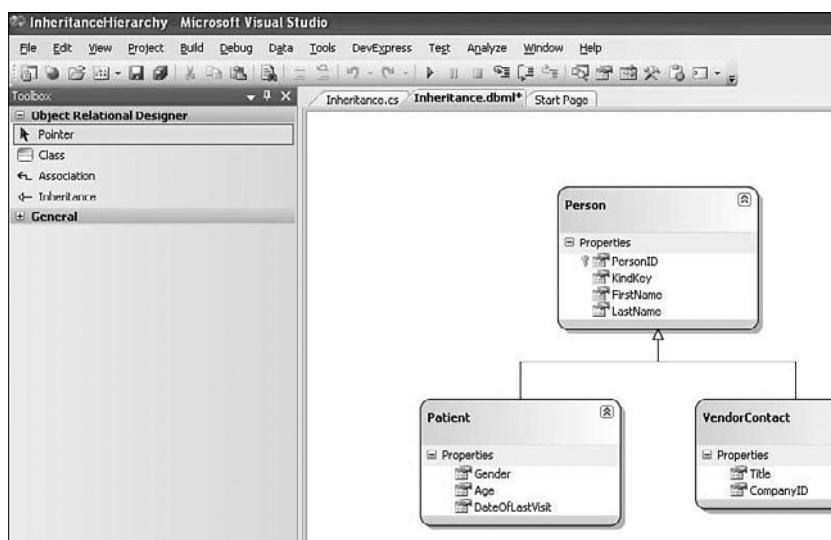


FIGURE 14.4 Define inheritance visually from the Toolbox.

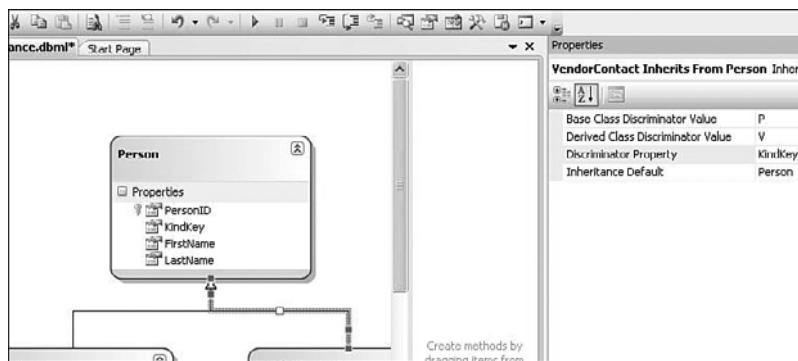


FIGURE 14.5 Set the derived type and base type discriminator values in the Properties window.

Customizing Classes Created with the LINQ to SQL Designer

If you want to customize the classes generated by the LINQ to SQL Object Relational Designer (or O/R Designer), open the generated .cs file (not the designer.cs file). For example, assume you want to add the reflection tool for dumping an object's state to the generated Person entity class. Further assume that the .dbml class was named Inheritance. We can open the Inheritance.cs file and add the partial class definition for Person to the file, rounding out the .cs file, as shown in Listing 14.4.

LISTING 14.4 To Extend LINQ to SQL Designer-generated Classes, Use the Partial Modifier

```

using System.Text;
using System.Reflection;
using System;
using System.Linq;

namespace Inheritance
{
    partial class InheritanceDataContext
    {
    }

    partial class Person
    {
        public override string ToString()
        {
            StringBuilder builder = new StringBuilder();
            builder.Append(this.GetType().Name + "\n");
            PropertyInfo[] props = this.GetType().GetProperties();

            // using array for each
            Array.ForEach(props.ToArray(), prop =>
                builder.AppendFormat("{0} : {1}", prop.Name,
                    prop.GetValue(this, null) == null ? "<empty>\n" :
                    prop.GetValue(this, null).ToString() + "\n"));

            return builder.ToString();
        }
    }
}

```

Adding EntitySet Classes as Properties

In object-oriented parlance, association means when class A refers to class B but doesn't control its lifetime. In C#, every property using an intrinsic type like `int` or `string` can be called an association. It is possible to argue that because C# is garbage collected, nothing is owned and, thus, everything that is not inheritance is an association. For our purposes, because C# can create instances of classes, we call a class that has a *class-member* an example of aggregation by composition.

An `EntitySet<T>` is a generic type, where `T` is an instance of an entity class that is loaded on demand. `EntitySets` are used to represent composition where one class owns an instance of one or more downstream classes. For example, customers could be said to own their orders, so you could define a property whose type is `EntitySet<Order>`. When you access the `EntitySet` property, LINQ loads the data for the `EntitySet`. (You can also load the data explicitly by calling `EntitySet.Load()`.)

Listing 14.5 contains a typed `DataContext` for the Northwind database and `Table` and `Column` attributes for the `Customers` table. It also introduces the `AssociationAttribute` for detail `EntitySet` objects. The named argument `Storage` indicates the field containing the `EntitySet` data, and the named argument `OtherKey` indicates the foreign key that correlates parent and child rows in the relationship (in this case, `Customers` and `Orders`). Finally, (referring to the `Orders` property) the property type is an `EntitySet<T>`, where `T` is the `Order` entity class (also defined in the listing).

LISTING 14.5 A One-to-Many or One-to-One Relationship Can Be Expressed Using the `AssociationAttribute` and the `EntitySet` Generic Class, as Shown in the Example for the Customers and Orders Tables

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data.Linq;
using System.Data.Linq.Mapping;
using System.Reflection;
using System.Collections;

namespace AddingEntityClassProperties
{
    class Program
    {
        static void Main(string[] args)
        {
            MasterDetailContext context = new MasterDetailContext();
            Table<Customer> customersWithOrders = context.GetCustomers();

            Array.ForEach(customersWithOrders.ToArray(),
                r=>
                {
                    Console.WriteLine(r);
                });
            Console.ReadLine();
        }
    }

    public class MasterDetailContext : DataContext
    {
        public static readonly string connectionString =
            "Data Source=BUTLER;Initial Catalog=Northwind;Integrated Security=True";

        public MasterDetailContext() : base(connectionString)
```

LISTING 14.5 Continued

```
{w
    this.Log = Console.Out;
}

public Table<Customer> GetCustomers()
{
    return this.GetTable<Customer>();
}
}

[Table(Name="Customers")]
public class Customer
{
    [Column(IsPrimaryKey=true)]
    public string CustomerID{ get; set; }

    [Column()]
    public string CompanyName{get; set; }

    [Column()]
    public string ContactName{get; set; }

    [Column()]
    public string ContactTitle{get; set; }

    [Column()]
    public string Address{get; set; }

    [Column()]
    public string City{get; set; }

    [Column()]
    public string Region{get; set; }

    [Column()]
    public string PostalCode{get; set; }

    [Column()]
    public string Country{get; set; }

    [Column()]
    public string Phone{get; set; }

    [Column()]
    public string Fax{get; set; }
```

LISTING 14.5 Continued

```
private EntitySet<Order> orders;

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

public override string ToString()
{
    StringBuilder builder = new StringBuilder();
    PropertyInfo[] props = this.GetType().GetProperties();

    // using array for each
    Array.ForEach(props.ToArray(), prop =>
    {
        object obj = prop.GetValue(this, null);
        if(obj is IList)
            (obj as IList).AppendEntitySet(builder);
        else
        {
            builder.AppendFormat("{0} : {1}",
                prop.Name,
                prop.GetValue(this, null) == null ? "<empty>\n" :
                prop.GetValue(this, null).ToString() + "\n");
        });
    });

    return builder.ToString();
}

public static class ExtendsList
{
    public static void AppendEntitySet(this IList entity,
        StringBuilder builder)
    {
        foreach(var obj in entity)
        {
            builder.AppendFormat("{0}\n", obj.ToString());
        }
    }
}

[Table(Name="Orders")]
public class Order
```

LISTING 14.5 Continued

```
{  
    [Column(IsPrimaryKey=true)]  
    public int OrderID{ get; set; }  
  
    [Column()]  
    public string CustomerID{ get; set; }  
  
    [Column(CanBeNull=true)]  
    public int? EmployeeID{get; set;}  
  
    [Column(CanBeNull = true)]  
    public DateTime? OrderDate{ get; set; }  
  
    [Column(CanBeNull = true)]  
    public DateTime? RequiredDate{ get; set; }  
  
    [Column(CanBeNull = true)]  
    public DateTime? ShippedDate{ get; set; }  
  
    [Column(CanBeNull = true)]  
    public int? ShipVia{ get; set; }  
  
    [Column(CanBeNull = true)]  
    public decimal? Freight{ get; set; }  
  
    [Column()]  
    public string ShipName{ get; set; }  
  
    [Column()]  
    public string ShipAddress{ get; set; }  
  
    [Column()]  
    public string ShipCity{ get; set; }  
  
    [Column()]  
    public string ShipRegion{ get; set; }  
  
    [Column()]  
    public string ShipPostalCode{ get; set; }  
  
    [Column()]  
    public string ShipCountry{ get; set; }  
  
    public override string ToString()  
    {  
        StringBuilder builder = new StringBuilder();
```

LISTING 14.5 Continued

```
PropertyInfo[] props = this.GetType().GetProperties();  
  
        // using array for each  
        Array.ForEach(props.ToArray(), prop =>  
            builder.AppendFormat("\t{0} : {1}", prop.Name,  
                prop.GetValue(this, null) == null ? "<empty>\n" :  
                prop.GetValue(this, null).ToString() + "\n"));  
  
    return builder.ToString();  
}  
}  
}
```



Creating Databases with LINQ to SQL

Many common scenarios depict an existing database and read and write to that database. More advanced scenarios create databases and tables at runtime. For example, tools like Microsoft SourceSafe create databases on the fly based on a common template, and tools like ACT! by Sage permit the user to define the schema and extend and create a database.

LINQ to SQL supports database creation through the `DataContext` class. The way it works is that you define a `DataContext` with a connection string. You define entity classes with the `Table` and `Column` attributes, and when you call `DataContext.CreateDatabase`, LINQ generates the SQL calls for you. Listing 14.6 demonstrates how to delete an existing database, create a new database, and add some data to the `Routes` table. The database is named `Planner` and the single table is named `Routes`. Figure 14.6 shows some of the SQL text generated and executed to perform the create and insert operations. (`DeleteDatabase` performs a `DROP DATABASE [PLANNER]` against the master table in the SQL Server instance.)

```
file:///C:/books/Addison Wesley/INQ/Source/Chapter 14/CreateDatabase1.inqToSql/Create... - x
on disk 'PLANNER'.
The CREATE DATABASE process is allocating 0.49 MB on disk 'PLANNER_log'...
.Net SqlClient Data Provider: Changed database context to 'PLANNER'..
SET QUOTED_IDENTIFIER ON
CREATE TABLE [Routes] (
    [ID] Int NOT NULL,
    [StartAirport] NChar(5),
    [EndAirport] NChar(5),
    [TimeInRoute] float,
    [GroundDistance] float,
    CONSTRAINT [PK_Routes] PRIMARY KEY ([ID])
)
INSERT INTO [Routes]([ID], [StartAirport], [EndAirport], [TimeInRoute], [GroundDistance])
VALUES (<Op0>, <Op1>, <Op2>, <Op3>, <Op4>)
-- <Op0>: Input Int {Size = 0; Prec = 0; Scale = 0} [0]
-- <Op1>: Input NChar {Size = 5; Prec = 0; Scale = 0} [KTEU]
-- <Op2>: Input NChar {Size = 5; Prec = 0; Scale = 0} [KLAN]
-- <Op3>: Input Float {Size = 0; Prec = 0; Scale = 0} [6000000000]
-- <Op4>: Input Float {Size = 0; Prec = 0; Scale = 0} [10]
-- Context: SqlProviderForSql2008 Model: AttrributedMetaModel Build: 3.5.21022.8
```

FIGURE 14.6 LINQ to SQL supports deleting and creating databases and managing tables and table data on the run.

LISTING 14.6 Deleting and Creating Databases and Inserting Data with LINQ to SQL

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data.Linq;
using System.Data.Linq.Mapping;

namespace CreateDatabaseLinqToSql
{
    class Program
    {
        static void Main(string[] args)
        {
            const string connectionString =
                "Data Source=BUTLER;Initial Catalog=PLANNER;" +
                "Integrated Security=True;Pooling=False";

            RoutePlanner planner =
                new RoutePlanner(connectionString);

            // create the database
            if(planner.DatabaseExists())
                planner.DeleteDatabase();

            planner.CreateDatabase();

            Table<Route> routes = planner.Routes;

            Route route = new Route
            {
                StartAirport="KTEW",
                EndAirport="KLAN",
                TimeInRoute=TimeSpan.FromMinutes(10).Ticks,
                GroundDistance=10
            };

            routes.InsertOnSubmit(route);
            planner.SubmitChanges();
        }
    }

    public class RoutePlanner : DataContext
```

LISTING 14.6 Continued

```
{  
    public Table<Route> Routes;  
    public RoutePlanner(string connectionString) : base(connectionString)  
    {}  
}  
  
[Table(Name="Routes")]  
public class Route  
{  
    [Column(IsPrimaryKey=true)]  
    public int ID{ get; set; }  
  
    [Column(DbType="NChar(5)")]  
    public string StartAirport{ get; set; }  
  
    [Column(DbType="NChar(5)")]  
    public string EndAirport{ get; set; }  
  
    [Column(DbType="float", CanBeNull=true)]  
    public float? TimeInRoute{ get; set; }  
  
    [Column(DbType="float", CanBeNull=true)]  
    public float? GroundDistance{ get; set; }  
}  
}
```

14

Notice that most of the code in Listing 14.6 is consistent with code seen previously in this chapter. New code includes the `DataContext.DatabaseExists()`, `DataContext.DeleteDatabase`, and `DataContext.DeleteDatabase` calls. In addition, the code that begins with "Table<Router> routes =" requests the collection of Routes from the typed `DataContext`. After the Table of Route objects is created, a new Route is created using named arguments; this object is added to the collection using the `InsertOnSubmit` method and `DataContext.SubmitChanges` writes the new row to the database. (You can use the SQL query designer in Visual Studio to verify the insert operation. Figure 14.6 also shows the generated `INSERT` statement generated by LINQ to SQL.)

For more information on deletes, updates, and inserts, refer to Chapter 16.

Summary

LINQ is a fully formed language feature. In this chapter, you learned that LINQ to SQL supports entities that are inline with SQL's ability to store null fields in traditionally non-null kinds of fields like integers. You also learned how inheritance hierarchies can be mapped with the `InheritanceMappingAttribute` and associations can be mapped with the `AssociationAttribute` and the `EntityType<T>` generic class. Many of these advanced features can be generated visually with the LINQ to SQL Object Relational Designer.

Finally, the last section demonstrated that you can create and delete databases dynamically with LINQ and easily manage the tables in these databases. You'll build on these skills by exploring aggregation and join relationships in the next two chapters (Chapter 15, "Joining Database Tables with LINQ Queries") and table modifications, stored procedures, functions, and transactions in Chapter 16.

CHAPTER 15

Joining Database Tables with LINQ Queries

“In dreams begins responsibility.”

—William Butler Yeats

There is a small likelihood that you will read this chapter title and reflect that it is similar to Chapter 11, “Joining Query Results.” In fact, this chapter and Chapter 11 (also on joins) do share fundamental underpinnings; both chapters cover LINQ joins. The difference is that Chapter 11 covered joins for custom classes, and this chapter covers joins for ADO.NET objects and entity classes. There are differences.

For example, this chapter looks at joins for LINQ to DataSets and joins for LINQ to SQL. To that end, we have to write the joins to deal with nullable fields, how to provide a default type for the `DefaultIfEmpty` argument of a left join, LINQ to SQL for Views, and databinding.

Rather than assume you could very quickly make the leap from LINQ for objects to LINQ for data by mentally filling in these gaps, this chapter fills in the gaps for you, offers additional sample types, and includes new material for views and databinding. (Check out the section on “Defining Joins with LINQ to SQL,” which provides an example based on the sample Pubs database. This section demonstrates how to convert programmer-unfriendly column names to more readable Entity property names using the `Name` argument of the `ColumnAttribute`.)

IN THIS CHAPTER

- ▶ Defining Joins with LINQ to DataSet
- ▶ Defining Joins with LINQ to SQL
- ▶ Querying Views with LINQ
- ▶ Databinding with LINQ to SQL

Defining Joins with LINQ to DataSet

LINQ for data is supported in two basic ways. You can write LINQ to ADO.NET code that provides for querying tables in DataSets, and you can define entities and use LINQ to SQL to query entities—classes that map to tables. Although you can choose either approach as it suits your needs, in general, consider using LINQ to DataSets only if you are incorporating LINQ into existing code that uses ADO.NET DataSets and tables. (Coverage of that style of joins is covered in this section.) If you are writing new code, consider using LINQ to SQL and object-relational maps entities mapped to SQL tables. (Joins for LINQ to SQL are covered beginning in the section titled “Defining Joins with LINQ to SQL.”)

All the join types, such as cross joins and group joins, are not covered for both types of LINQ for data, but together most of the join types are covered in this chapter. After you go over the basic and most common join types in this chapter, you will be able to extrapolate the less common right and cross joins from the material in Chapter 11.

Coding Equijoins

Fundamentally, an equijoin for LINQ to DataSets is the same as equijoins for LINQ to objects. The key difference is that you are dealing with `DataTable` and `DataRow` objects and you will have to add the code to query the table and express the join by requesting a `DataRow`'s `Field` value. (An explanation of how to query a `DataTable` and request `Field` data is covered in Chapter 13, “Querying Relational Data with LINQ.”) The code in Listing 15.1 demonstrates how to build on the information in Chapter 13 to express an equijoin for LINQ to DataSets.

LISTING 15.1 Defining an Equijoin Using the `join...in...on...equals` Clause for LINQ to DataSets

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data.SqlClient;
using System.Data;

namespace LinqToDataSetEquijoin
{
    class Program
    {
        static void Main(string[] args)
        {
            const string connectionString =
                "Data Source=.\\SQLEXPRESS;AttachDbFilename=" +
                "\\\"C:\\Books\\Sams\\LINQ\\Northwind\\northwnd.mdf\\\";" +
                ";Integrated Security=True;Connect Timeout=30;User Instance=True";
```

LISTING 15.1 Continued

```
DataSet dataset = new DataSet();
using (SqlConnection connection = new SqlConnection(connectionString))
{
    connection.Open();
    const string SQL = "SELECT * FROM Products;SELECT * FROM SUPPLIERS";
    SqlDataAdapter adapter = new SqlDataAdapter(SQL, connection);
    adapter.Fill(dataset);
}

DataTable products = dataset.Tables[0];
DataTable suppliers = dataset.Tables[1];

var inventory = from product in products.AsEnumerable()
                join supplier in suppliers.AsEnumerable()
                on product.Field<int>("SupplierID")
                equals supplier.Field<int>("SupplierID")
                orderby product.Field<string>("ProductName")
                select new
                {
                    Company=supplier.Field<string>("CompanyName"),
                    City=supplier.Field<string>("City"),
                    Phone=supplier.Field<string>("Phone"),
                    Product=product.Field<string>("ProductName"),
                    Price=product.Field<decimal>("UnitPrice"),
                    InStock=product.Field<Int16>("UnitsInStock"),
                    Discontinued=product.Field<bool>("Discontinued")
                };

string line = new string('-', 40);
foreach(var item in inventory)
{
    Console.WriteLine("Company: {0}", item.Company);
    Console.WriteLine("City: {0}", item.City);
    Console.WriteLine("Phone: {0}", item.Phone);
    Console.WriteLine("Product: {0}", item.Product);
    Console.WriteLine("Price: {0:C}", item.Price);
    Console.WriteLine("Quantity on hand: {0}", item.InStock);
    Console.WriteLine("Discontinued: {0}", item.Discontinued);
    Console.WriteLine(line);
    Console.WriteLine();
}

Console.ReadLine();
}
```

In Listing 15.1, both queries are sent to the Northwind database via ADO.NET as a single, semicolon-delimited string. The equijoin is constructed by defining the first range element, `product`, from the `products` `DataTable` using the `AsEnumerable` extension method, and the second join is `supplier` from the `suppliers` `DataTable` using `AsEnumerable` to facilitate querying. The equijoin is `product.SupplierID equals supplier.SupplierID`. The `SupplierID` is extracted from the range values using the `Field` extension method properties and the `equals` clause. The rest of the code is boilerplate code that demonstrates a complete functional example.

Coding Nonequijoins

A nonequijoin is a join that uses inequality. These joins have to be expressed using a `where` clause and can include one or more predicates. Listing 15.2 is an excerpt that can be plugged into Listing 15.1 (with minor changes to display the output). Listing 15.2 uses a `where` clause and an equality predicate (`==`) based on the `SupplierID` and looks for discontinued products in the second predicate for `where`.

LISTING 15.2 A Nonequijoin Based on Two Predicates and a where Clause

```
var discontinued = from product in products.AsEnumerable()
                  from supplier in suppliers.AsEnumerable()
                  where product.Field<int>("SupplierID")
                  == supplier.Field<int>("SupplierID")
                  &&
                  product.Field<bool>("Discontinued") == true
                  select new
                  {
                      Company=supplier.Field<string>("CompanyName"),
                      City=supplier.Field<string>("City"),
                      Phone=supplier.Field<string>("Phone"),
                      Product=product.Field<string>("ProductName"),
                      Price=product.Field<decimal>("UnitPrice"),
                      InStock=product.Field<Int16>("UnitsInStock"),
                      Discontinued=product.Field<bool>("Discontinued")
                  };

```

A classic nonequijoin using not equals (`!=` in LINQ and `<>` in SQL) are seldom used but can be employed to define a self-join. Listing 15.3 shows a self-join on the `Northwind.Products` table based on `ProductName` equality and `SupplierID` inequality. (You could use a key in a real-world example if the key for products were not unique as it is in the Northwind database.)

LISTING 15.3 A Nonequijoin Used to Contrive a Self-Join to See If There Are Two Suppliers Providing the Same Product (By Name) at Different Prices

```
var productsFromMultipleSuppliers =
    from product1 in products.AsEnumerable()
    from product2 in products.AsEnumerable()
    where product1.Field<string>("ProductName") ==
        product2.Field<string>("ProductName")
    && product1.Field<int>("SupplierID") !=
        product2.Field<int>("SupplierID")
    select new
    {
        ProductName = product1.Field<string>("ProductName"),
        Supplier1 = product1.Field<int>("SupplierID"),
        Supplier1Price = product1.Field<decimal>("UnitPrice"),
        Supplier2 = product2.Field<int>("SupplierID"),
        Supplier2Price = product1.Field<decimal>("UnitPrice")
    };

```

In a good relational database with constraints and unique keys, duplicates by key are less likely to occur. Self-joins are usually more prevalent when porting old, legacy database from nonrelational systems where the data technology is not enforcing uniqueness and data integrity.

5

Defining a Left Join and a Word about Right Joins

The left join and right join are not supported by keywords in LINQ. (And, according to a group blog from the VB team—<http://blogs.msdn.com/vbteam/archive/2008/01/31/converting-sql-to-linq-part-8-left-right-outer-join-bill-horst.aspx>—these features are not planned in current or near-future releases.) However, as demonstrated in Chapter 11, the left join (or left outer join) can be constructed by flattening the group join with the `DefaultIfEmpty` extension method.

In this section, a left join for LINQ to DataSets is demonstrated followed by a brief discussion on the infrequently used right join and how that can be constructed as well.

Defining a Left Outer Join with LINQ to DataSets

As demonstrated in Chapter 11, a left outer join is constructed with two sequences: a group join and the `DefaultIfEmpty` extension method. To implement a left join in LINQ to DataSets, you need to address the minor challenge of handling the `DefaultIfEmpty` method.

In short, the left join needs an empty child `DataRow` in the event there is no matching child row. You can handle `DefaultIfEmpty` by calling this method with the `DataTable.NewRow` method and defining the project—in `select new`—to use nullable types

for child data. The latter step (shown in Listing 15.4) is necessary because empty new rows for missing children will be initialized with nulls.

LISTING 15.4 Defining a Left Outer Join in LINQ to DataSets Depends on Calling DataTable.NewRow to Initialize Empty, Missing Child Data Rows and Nullable Types in the Projection Created in the select Clause

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data.SqlClient;
using System.Data;

namespace LinqToDataSetLeftJoin
{
    class Program
    {
        static void Main(string[] args)
        {
            const string connectionString =
                "Data Source=.\SQLExpress;AttachDbFilename=" +
                "\"C:\\Books\\Sams\\LINQ\\Northwind\\northwnd.mdf\";" +
                ";Integrated Security=True;Connect Timeout=30;User Instance=True";

            DataSet dataset = new DataSet();
            using (SqlConnection connection = new SqlConnection(connectionString))
            {
                connection.Open();
                const string SQL = "SELECT * FROM CUSTOMERS;SELECT * FROM ORDERS";
                SqlDataAdapter adapter = new SqlDataAdapter(SQL, connection);
                adapter.Fill(dataset);
            }

            DataTable customers = dataset.Tables[0];
            DataTable orders = dataset.Tables[1];

            var customersWithoutOrders = from customer in customers.AsEnumerable()
                                         join order in orders.AsEnumerable()
                                         on customer.Field<string>("CustomerID") equals
                                         order.Field<string>("CustomerID") into children
                                         from child in children.DefaultIfEmpty(orders.NewRow())
                                         let OrderID = child.Field<int?>("OrderID")
                                         where OrderID == null
                                         select new
                                         {
                                             CustomerID = customer.Field<string>("CustomerID"),
                                             CompanyName = customer.Field<string>("CompanyName"),
                                             ContactName = customer.Field<string>("ContactName"),
                                             ContactTitle = customer.Field<string>("ContactTitle"),
                                             Address = customer.Field<string>("Address"),
                                             City = customer.Field<string>("City"),
                                             PostalCode = customer.Field<string>("PostalCode"),
                                             Country = customer.Field<string>("Country"),
                                             Phone = customer.Field<string>("Phone"),
                                             Fax = customer.Field<string>("Fax"),
                                             Email = customer.Field<string>("Email"),
                                             OrderCount = (int?)order.Count
                                         };
        }
    }
}
```

LISTING 15.4 Continued

```

        Company=customer.Field<string>("CompanyName"),
        City=customer.Field<string>("City"),
        Phone=customer.Field<string>("Phone"),
        OrderID=child.Field<int?>("OrderID"),
        OrderDate=child.Field<DateTime?>("OrderDate"),
        RequiredDate=child.Field<DateTime?>("RequiredDate"),
        ShipCity=child.Field<string>("ShipCity")
    };

    string line = new string('-', 40);
    foreach(var item in customersWithoutOrders)
    {
        Console.WriteLine("Company: {0}", item.Company);
        Console.WriteLine("City: {0}", item.City);
        Console.WriteLine("Phone: {0}", item.Phone);
        Console.WriteLine("Order ID: {0}", item.OrderID);
        Console.WriteLine("Order Date: {0}", item.OrderDate);
        Console.WriteLine("Required Date: {0}", item.RequiredDate);
        Console.WriteLine("Ship to: {0}", item.ShipCity);
        Console.WriteLine(line);
        Console.WriteLine();
    }

    Console.ReadLine();
}
}
}

```

The into...from...in clause flattens a group join into a left join. If a right `DataRow` isn't available, invoke `NewRow` to initialize the empty, or missing, child row, and define the projection to support nullable types for the missing child row data (shown in the `select new` clause in Listing 15.4). For instance, `child.Field<int?>("OrderID")` makes it permissible to initialize the projection without an actual `OrderID` in the `OrderDetails` data row.

Considering Right Joins

The right outer join is not supported in LINQ to DataSets or LINQ to SQL. A right outer join returns all of the child rows and only the parent rows that have correlated data. A right join will find children with parents and orphaned children. (Orphaned children occur when the database designer or SQL author forgets to cascade deletes.)

You can construct a right join by reversing the order of the range arguments in a LINQ query. For example, if you switch the order of the range variables `customer` and `order` in Listing 15.4, the result is a left join from `Orders` to `Customers`, which yields the same

result as a right join. The implication is also that the grouping behavior occurs on Customers and not Orders.

Right joins are pretty rare. Again, you are likely to see orphaned children in legacy data where cascaded deletes were unsupported. If you remember to enable cascaded deletes in your SQL databases and literally delete children with LINQ, orphaned data should not exist. Listing 15.5 shows a RIGHT OUTER JOIN in SQL, and Listing 15.6 shows an implicit right join constructed with LINQ by reversing the order of the range and sequence values from Listing 15.4.

LISTING 15.5 A RIGHT OUTER JOIN in SQL

```
SELECT      C.CustomerID, C.CompanyName, C.ContactName, C.ContactTitle,
C.Address, C.City,
C.Region, C.PostalCode, C.Country, C.Phone, C.Fax, O.OrderID,
O.CustomerID AS Expr1, O.EmployeeID, O.OrderDate, O.RequiredDate, O.ShippedDate,
O.ShipVia,
O.Freight, O.ShipName, O.ShipAddress, O.ShipCity, O.ShipRegion, O.ShipPostalCode,
O.ShipCountry
FROM Customers AS C RIGHT OUTER JOIN Orders AS O ON C.CustomerID = O.CustomerID
```

LISTING 15.6 A Modified Version of Listing 15.4 That Reverses the Order of the Customers and Orders Clauses and Groups on Customers Rather Than Orders, Yielding the Same Result as a RIGHT OUTER JOIN

```
var orphanedOrders = from order in orders.AsEnumerable()
                      join customer in customers.AsEnumerable()
                        on order.Field<string>("CustomerID") equals
                        customer.Field<string>("CustomerID")
                      into parent
                      from p in parent.DefaultIfEmpty(customers.NewRow())
                      select new
                      {
                          CustomerID = p.Field<string>("CustomerID"),
                          Company = p.Field<string>("CompanyName"),
                          City = p.Field<string>("City"),
                          Phone = p.Field<string>("Phone"),
                          OrderID = order.Field<int?>("OrderID"),
                          OrderDate = order.Field<DateTime?>("OrderDate"),
                          RequiredDate = order.Field<DateTime?>("RequiredDate"),
                          ShipCity = order.Field<string>("ShipCity")
                      };

```

NOTE

Although there is a foreign key constraint on `orders` and `customers`, when I added an order, I put the `Orders.CustomerID` in mixed case while the `Customers.CustomerID` was in all uppercase. The result was SQL Server respected the constraint on `CustomerID` but LINQ did not, resulting in a query that looked like orphaned orders exist.

The key is that well-constructed databases shouldn't create orphans. For example, if a customer is deleted, the customer's orders should be deleted too. Legacy database (and data) often don't support features found in modern relational systems; however, clearly it is possible to get a result that looks like orphans exist as demonstrated by primary key strings mismatched by case.

Defining Joins with LINQ to SQL

If you have a lot of existing ADO.NET code that uses `DataSets` and `DataTables`, clearly you can use LINQ without adding a lot of entities mapped to tables. If you are defining new systems, consider defining object-relational mappings, mapping entity classes to database tables. The result is much cleaner code, simpler LINQ expressions, and classes that are independent of ADO.NET types.

This part of the chapter demonstrates the most common joins, including the equijoin, group join, and left join for LINQ to SQL. Clearly, you can implement the other joins shown earlier in this chapter and in Chapter 11, but it isn't necessary to cover each kind of join again once you have a basic understanding of the differences between LINQ to SQL joins and joins on other kinds of objects.

Coding Equijoins

The equijoin is the most common join. Equijoins are also directly supported. Because that is the case, this section covers an equijoin and a new database is included to keep the material fresh as well as to demonstrate how to correct poorly named database columns.

The example in Listing 15.7 demonstrates an equijoin on the `pubs` database. `pubs` is a sample database that contains publishers, authors, books, and royalty information but uses abbreviations to excess. In Listing 15.7, the abbreviated column names are corrected with the `Name` argument to the `ColumnAttribute` and null fields are handled with nullable types in the entity definition. (Nullable types for entities are introduced in Chapter 16, "Updating Anonymous Relational Data.")

LISTING 15.7 An Equijoin on the `Titles` Table, the `TitleAuthor` Linking Table—to Support Multiple Book Authors—and the `Authors` Table of the `pubs` Sample Database

```
using System;
using System.Collections.Generic;
using System.Linq;
```

LISTING 15.7 Continued

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

namespace LinqToSqlEquijoin
{
    class Program
    {
        static void Main(string[] args)
        {
            Publishers publishers = new Publishers();

            var titles = from title in publishers.Titles
                         join titleAuthor in publishers.TitleAuthors on
                             title.TitleID equals titleAuthor.TitleID
                         join author in publishers.Authors on
                             titleAuthor.AuthorID equals author.AuthorID
                         select new
                         {
                             Author=author.FirstName + ' ' + author.LastName,
                             Book=title.BookTitle
                         };

            Array.ForEach(titles.ToArray(), b=>
                Console.WriteLine("Author: {0}, Book: {1}", b.Author, b.Book));

            Console.ReadLine();
        }
    }

    public class Publishers : DataContext
    {
        private static readonly string connectionString =
            "Data Source=BUTLER;Initial Catalog=pubs;Integrated Security=True";

        public Publishers() : base(connectionString)
        {
            Log = Console.Out;
        }

        public Table<Author> Authors
        {
            get{ return this.GetTable<Author>(); }
        }

        public Table<TitleAuthor> TitleAuthors
```

LISTING 15.7 Continued

```
{  
    get{ return this.GetTable<TitleAuthor>(); }  
}  
  
public Table<Title> Titles  
{  
    get{ return this.GetTable<Title>(); }  
}  
}  
  
[Table(Name="authors")]  
public class Author  
{  
    [Column(Name="au_id", IsPrimaryKey=true)]  
    public int AuthorID{ get; set; }  
  
    [Column(Name="au_lname")]  
    public string LastName{ get; set; }  
  
    [Column(Name="au_fname")]  
    public string FirstName{ get; set; }  
  
    [Column(Name="phone")]  
    public string Phone{ get; set; }  
  
    [Column(Name="address")]  
    public string Address{ get; set; }  
  
    [Column(Name="city")]  
    public string City{ get; set; }  
  
    [Column(Name="state")]  
    public string State{ get; set; }  
  
    [Column(Name="zip")]  
    public string ZipCode{ get; set; }  
  
    [Column(Name="contract")]  
    public bool? Contract{ get; set; }  
}  
  
[Table(Name="titleauthor")]  
public class TitleAuthor  
{  
    [Column(Name="au_id")]  
    public int? AuthorID{ get; set; }
```

LISTING 15.7 Continued

```
[Column(Name="title_id")]
public int? TitleID{ get; set; }

[Column(Name="au_ord")]
public Int16? AuthorOrder{ get; set; }

[Column(Name="royaltyper")]
public int? RoyaltyPercentage{ get; set; }
}

[Table(Name="titles")]
public class Title
{
    [Column(Name="title_id", IsPrimaryKey=true)]
    public int? TitleID{ get; set; }

    [Column(Name="title")]
    public string BookTitle{ get; set; }

    [Column(Name="type")]
    public string Type{ get; set; }

    [Column(Name="pub_id")]
    public string PublisherID{ get; set; }

    [Column(Name="price")]
    public decimal? Price{ get; set; }

    [Column(Name="advance")]
    public decimal? Advance{ get; set; }

    [Column(Name="royalty")]
    public int? Royalty{ get; set; }

    [Column(Name="ytd_sales")]
    public int? YearToDateSales{ get; set; }

    [Column(Name="notes")]
    public string Notes{ get; set; }

    [Column(Name="pubdate")]
    public DateTime? PublicationDate{ get; set; }
}
```

The LINQ query in Listing 15.7 clearly demonstrates that you can define an equijoin on multiple tables using additional join clauses. (Pubs uses a linking table TitleAuthor to map multiple authors to a single title.) The projection defines a composite field Author based on FirstName and LastName and nullable types for value types in the entity classes. (Nullable types are defined with the question mark (?) after the data type.)

Poorly named columns like ytd_sales are corrected by using the Name argument to the ColumnAttribute. For example, ytd_sales is mapped to YearToDateSales in the Title class.

Implementing the Group Join

There is actually a GroupJoin extension method. In VB .NET the GroupJoin is explicit. In C#, the group join is emitted to (Microsoft Intermediate Language) MSIL when you use the join...on...equals...into keywords collectively.

A group join is a master-detail relationship where for each element in the master sequence there is a contained association that includes the child objects. Listing 15.8 shows a group join on the Northwind Orders and Order Details tables, and a nested array displaying each element. Listing 15.9 shows the disassembled code showing how the LINQ statement is expressed as an extension method, clearly showing the GroupJoin (and how much more complicated it would be to use extension methods instead of queries had Microsoft stopped at extension methods).

LISTING 15.8 A Group Join Uses the join...on...equals...into and the GroupJoin Extension Method Is Emitted

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data.Linq;
using System.Data.Linq.Mapping;

namespace LinqToSqlGroupJoin
{
    class Program
    {
        static void Main(string[] args)
        {
            Northwind northwind = new Northwind();

            var orderInformation = from order in northwind.Orders
                                  join detail in northwind.Details
                                  on order.OrderID equals detail.OrderID
                                  into orderDetails
                                  select new
                                  {
                                      OrderID = order.OrderID,
                                      CompanyName = order.Customer.CompanyName,
                                      OrderDate = order.OrderDate,
                                      Details = orderDetails
                                  };
        }
    }
}
```

LISTING 15.8 Continued

```

        into children
        select new
        {
            CustomerID = order.CustomerID,
            OrderDate = order.OrderDate,
            RequiredDate = order.RequiredDate,
            Details = children
        };
    }

    string line = new string('-', 40);
    Array.ForEach(orderInformation.ToArray(), r =>
    {
        Console.WriteLine("Customer ID: {0}", r.CustomerID);
        Console.WriteLine("Order Date: {0}", r.OrderDate
            .GetValueOrDefault().ToShortDateString());
        Console.WriteLine("Required Date: {0}", r.RequiredDate
            .GetValueOrDefault().ToShortDateString());

        Console.WriteLine("-----Order Details-----");
        Array.ForEach(r.Details.ToArray(), d =>
        {
            Console.WriteLine("Product ID: {0}", d.ProductID);
            Console.WriteLine("Unit Price: {0}", d.UnitPrice);
            Console.WriteLine("Quantity: {0}", d.Quantity);
            Console.WriteLine();
        });
        Console.WriteLine(line);
        Console.WriteLine();
    });
}

Console.ReadLine();
}
}

public class Northwind : DataContext
{
    private static readonly string connectionString =
        "Data Source=.\SQLExpress;AttachDbFilename=\"C:\\Books\\Sams\\\" +
        "LINQ\\Northwind\\northwnd.mdf\";" +
        "Integrated Security=True;Connect Timeout=30;User Instance=True";

    // "Data Source=.\SQLExpress;AttachDbFilename=c:\\temp\\northwnd.mdf;" +
    // "Integrated Security=True;Connect Timeout=30;User Instance=True";
}

```

LISTING 15.8 Continued

```
public Northwind()
    : base(connectionString)
{
    Log = Console.Out;
}

public Table<Order> Orders
{
    get{ return this.GetTable<Order>(); }
}

public Table<OrderDetail> Details
{
    get{ return GetTable<OrderDetail>(); }
}

[Table(Name = "dbo.Order Details")]
public partial class OrderDetail
{
    private int _OrderID;
    private int _ProductID;
    private decimal _UnitPrice;
    private short _Quantity;
    private float _Discount;

    public OrderDetail()
    {
    }

    [Column(Storage = "_OrderID", DbType = "Int NOT NULL", IsPrimaryKey = true)]
    public int OrderID
    {
        get
        {
            return this._OrderID;
        }
        set
        {
            this._OrderID = value;
        }
    }
}
```

LISTING 15.8 Continued

```
[Column(Storage = "_ProductID", DbType = "Int NOT NULL", IsPrimaryKey = true)]
public int ProductID
{
    get
    {
        return this._ProductID;
    }
    set
    {
        this._ProductID = value;
    }
}

[Column(Storage = "_UnitPrice", DbType = "Money NOT NULL")]
public decimal UnitPrice
{
    get
    {
        return this._UnitPrice;
    }
    set
    {
        this._UnitPrice = value;
    }
}

[Column(Storage = "_Quantity", DbType = "SmallInt NOT NULL")]
public short Quantity
{
    get
    {
        return this._Quantity;
    }
    set
    {
        this._Quantity = value;
    }
}

[Column(Storage = "_Discount", DbType = "Real NOT NULL")]
public float Discount
{
    get
    {
```

LISTING 15.8 Continued

```
        return this._Discount;
    }
    set
    {
        this._Discount = value;
    }
}

[Table(Name = "dbo.Orders")]
public partial class Order
{

    private int _OrderID;

    private string _CustomerID;

    private System.Nullable<int> _EmployeeID;

    private System.Nullable<System.DateTime> _OrderDate;

    private System.Nullable<System.DateTime> _RequiredDate;

    private System.Nullable<System.DateTime> _ShippedDate;

    private System.Nullable<int> _ShipVia;

    private System.Nullable<decimal> _Freight;

    private string _ShipName;

    private string _ShipAddress;

    private string _ShipCity;

    private string _ShipRegion;

    private string _ShipPostalCode;

    private string _ShipCountry;

    public Order()
    {

    }

[Column(Storage = "_OrderID", AutoSync = AutoSync.OnInsert,
DbType = "Int NOT NULL IDENTITY", IsPrimaryKey = true, IsDbGenerated = true)]
```

LISTING 15.8 Continued

```
public int OrderID
{
    get
    {
        return this._OrderID;
    }
    set
    {
        this._OrderID = value;
    }
}

[Column(Storage = "_CustomerID", DbType = "NChar(5)")]
public string CustomerID
{
    get
    {
        return this._CustomerID;
    }
    set
    {
        this._CustomerID = value;
    }
}

[Column(Storage = "_EmployeeID", DbType = "Int")]
public System.Nullable<int> EmployeeID
{
    get
    {
        return this._EmployeeID;
    }
    set
    {
        this._EmployeeID = value;
    }
}

[Column(Storage = "_OrderDate", DbType = "DateTime")]
public System.Nullable<System.DateTime> OrderDate
{
    get
    {
        return this._OrderDate;
    }
}
```

LISTING 15.8 Continued

```
set
{
    this._OrderDate = value;
}
}

[Column(Storage = "_RequiredDate", DbType = "DateTime")]
public System.Nullable<System.DateTime> RequiredDate
{
    get
    {
        return this._RequiredDate;
    }
    set
    {
        this._RequiredDate = value;
    }
}

[Column(Storage = "_ShippedDate", DbType = "DateTime")]
public System.Nullable<System.DateTime> ShippedDate
{
    get
    {
        return this._ShippedDate;
    }
    set
    {
        this._ShippedDate = value;
    }
}

[Column(Storage = "_ShipVia", DbType = "Int")]
public System.Nullable<int> ShipVia
{
    get
    {
        return this._ShipVia;
    }
    set
    {
        this._ShipVia = value;
    }
}

[Column(Storage = "_Freight", DbType = "Money")]

15
```

LISTING 15.8 Continued

```
public System.Nullable<decimal> Freight
{
    get
    {
        return this._Freight;
    }
    set
    {
        this._Freight = value;
    }
}

[Column(Storage = "_ShipName", DbType = "NVarChar(40)")]
public string ShipName
{
    get
    {
        return this._ShipName;
    }
    set
    {
        this._ShipName = value;
    }
}

[Column(Storage = "_ShipAddress", DbType = "NVarChar(60)")]
public string ShipAddress
{
    get
    {
        return this._ShipAddress;
    }
    set
    {
        this._ShipAddress = value;
    }
}

[Column(Storage = "_ShipCity", DbType = "NVarChar(15)")]
public string ShipCity
{
    get
    {
        return this._ShipCity;
    }
    set
```

LISTING 15.8 Continued

```
{  
    this._ShipCity = value;  
}  
}  
  
[Column(Storage = "_ShipRegion", DbType = "NVarChar(15)")]  
public string ShipRegion  
{  
    get  
    {  
        return this._ShipRegion;  
    }  
    set  
    {  
        this._ShipRegion = value;  
    }  
}  
  
[Column(Storage = "_ShipPostalCode", DbType = "NVarChar(10)")]  
public string ShipPostalCode  
{  
    get  
    {  
        return this._ShipPostalCode;  
    }  
    set  
    {  
        this._ShipPostalCode = value;  
    }  
}  
  
[Column(Storage = "_ShipCountry", DbType = "NVarChar(15)")]  
public string ShipCountry  
{  
    get  
    {  
        return this._ShipCountry;  
    }  
    set  
    {  
        this._ShipCountry = value;  
    }  
}
```

LISTING 15.9 The Disassembled Code (from Reflector) Showing the Very Long Chain of Extension Methods, Including the GroupJoin That Ultimately Provides the Behavior of the LINQ Query

```

private static void Main(string[] args)
{
    ParameterExpression CS$0$0000;
    ParameterExpression CS$0$0002;
    Northwind northwind = new Northwind();
    var orderInformation = northwind.Orders.GroupJoin(northwind.Details,
        Expression.Lambda<Func<Order, int>>(Expression.Property(CS$0$0000 =
            Expression.Parameter(typeof(Order), "order"), (MethodInfo)
            methodof(Order.get_OrderID)), new ParameterExpression[] { CS$0$0000 }), 
        Expression.Lambda<Func<OrderDetail, int>>(Expression.Property(CS$0$0000 =
            Expression.Parameter(typeof(OrderDetail), "detail"), (MethodInfo)
            methodof(OrderDetail.get_OrderID)), new ParameterExpression[] { CS$0$0000 }), 
        Expression.Lambda(Expression.New((ConstructorInfo) methodof
            =>(<>f__AnonymousType0<string,
            DateTime?, DateTime?, IEnumerable<OrderDetail>>..ctor,
            =><>f__AnonymousType0<string,
            DateTime?, DateTime?, IEnumerable<OrderDetail>>), new Expression[] {
            Expression.Property(CS$0$0000 = Expression.Parameter(typeof(Order), "order"),
                (MethodInfo) methodof(Order.get_CustomerID)), Expression.Property(CS$0$0000,
                (MethodInfo) methodof(Order.get_OrderDate)), Expression.Property(CS$0$0000,
                (MethodInfo) methodof(Order.get_RequiredDate)), CS$0$0002 =
                    Expression.Parameter(typeof(IEnumerable<OrderDetail>), "children") },
            new MethodInfo[])
        { (MethodInfo) methodof(<>f__AnonymousType0<string, DateTime?, DateTime?,
            IEnumerable<OrderDetail>>.get_CustomerID, <>f__AnonymousType0<string,
            =>DateTime?,
            DateTime?, IEnumerable<OrderDetail>>), (MethodInfo)
            methodof(<>f__AnonymousType0<string, DateTime?, DateTime?,
            IEnumerable<OrderDetail>>.get_OrderDate, <>f__AnonymousType0<string, DateTime?,
            DateTime?, IEnumerable<OrderDetail>>), (MethodInfo)
            methodof(<>f__AnonymousType0<string, DateTime?, DateTime?,
            IEnumerable<OrderDetail>>.get_RequiredDate, <>f__AnonymousType0<string,
            =>DateTime?,
            DateTime?, IEnumerable<OrderDetail>>), (MethodInfo)
            methodof(<>f__AnonymousType0<string, DateTime?, DateTime?,
            IEnumerable<OrderDetail>>.get_Details, <>f__AnonymousType0<string, DateTime?,
            DateTime?, IEnumerable<OrderDetail>>) }, new ParameterExpression[] {
            =>CS$0$0000,
            CS$0$0002 }));
    string line = new string('-', 40);
    Array.ForEach(orderInformation.ToArray(), delegate (<>f__
```

LISTING 15.9 Continued

```
AnonymousType0<string, DateTime?, DateTime?, IEnumerable<OrderDetail>> r) {
    Console.WriteLine("Customer ID: {0}", r.CustomerID);
    Console.WriteLine("Order Date: {0}",
        r.OrderDate.GetValueOrDefault().ToShortDateString());
    Console.WriteLine("Required Date: {0}",
        r.RequiredDate.GetValueOrDefault().ToShortDateString());
    Console.WriteLine("-----Order Details-----");
    if (<>c__DisplayClass3.CS$<>9__CachedAnonymousMethodDelegate5 == null)
    {
        <>c__DisplayClass3.CS$<>9__CachedAnonymousMethodDelegate5 = delegate
(OrderDetail d) {
            Console.WriteLine("Product ID: {0}", d.ProductID);
            Console.WriteLine("Unit Price: {0}", d.UnitPrice);
            Console.WriteLine("Quantity: {0}", d.Quantity);
            Console.WriteLine();
        };
    }
    Array.ForEach<OrderDetail>(r.Details.ToArray<OrderDetail>(),
        <>c__DisplayClass3.CS$<>9__CachedAnonymousMethodDelegate5);
    Console.WriteLine(line);
    Console.WriteLine();
});
Console.ReadLine();
}
```

15

Implementing a Left Join

Because we are using entity classes with LINQ to SQL, constructing the LEFT JOIN uses the same code as constructing a left join in Chapter 11. To convert the group join to a left join, you add an additional *from* clause and range variable after the *into* clause and invoke the *DefaultIfEmpty* method in the group sequence. Listing 15.10 just contains the LINQ statement with the *DefaultIfEmpty* called on the children group. Remember with the left join, all of the parent information is repeated for each child, so a nested array is no longer needed.

LISTING 15.10 The Group Join Is Easily Converted to a Left Join Construction By Adding an Additional *from* Clause on the Group Sequence and Calling *DefaultIfEmpty*

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data.Linq;
using System.Data.Linq.Mapping;
```

LISTING 15.10 Continued

```
namespace LinqToSqlLeftJoin
{
    class Program
    {
        static void Main(string[] args)
        {
            Northwind northwind = new Northwind();

            var orderInformation = from order in northwind.Orders
                                   join detail in northwind.Details
                                   on order.OrderID equals detail.OrderID
                                   into children
                                   from child in children.DefaultIfEmpty()
                                   select new
                                   {
                                       order.CustomerID,
                                       order.OrderDate,
                                       order.RequiredDate,
                                       child.ProductID,
                                       child.UnitPrice,
                                       child.Quantity,
                                       child.Discount
                                   };
        }

        string line = new string('-', 40);
        Array.ForEach(orderInformation.ToArray(), r =>
        {
            Console.WriteLine("Customer ID: {0}", r.CustomerID);
            Console.WriteLine("Order Date: {0}", r.OrderDate
                .GetValueOrDefault().ToShortDateString());
            Console.WriteLine("Required Date: {0}", r.RequiredDate
                .GetValueOrDefault().ToShortDateString());

            Console.WriteLine("Product ID: {0}", r.ProductID);
            Console.WriteLine("Unit Price: {0:C}", r.UnitPrice);
            Console.WriteLine("Quantity: {0}", r.Quantity);
            Console.WriteLine("Discount: {0}", r.Discount);
            Console.WriteLine();
        });

        Console.ReadLine();
    }
}
```

LISTING 15.10 Continued

```
public class Northwind : DataContext
{
    private static readonly string connectionString =
        "Data Source=.\SQLExpress;AttachDbFilename=\"C:\\Books\\Sams\\\\\" +
        "LINQ\\Northwind\\northwnd.mdf\";" +
        "Integrated Security=True;Connect Timeout=30;User Instance=True";

    public Northwind()
        : base(connectionString)
    {
        Log = Console.Out;
    }

    public Table<Order> Orders
    {
        get{ return this.GetTable<Order>(); }
    }

    public Table<OrderDetail> Details
    {
        get{ return GetTable<OrderDetail>(); }
    }
}

[Table(Name = "dbo.Order Details")]
public partial class OrderDetail
{
    private int? _OrderID;
    private int? _ProductID;
    private decimal? _UnitPrice;
    private short? _Quantity;
    private float? _Discount;

    public OrderDetail()
    {
    }

    [Column(Storage = "_OrderID", DbType = "Int NOT NULL", IsPrimaryKey = true)]
    public int? OrderID
    {
        get
    }
}
```

LISTING 15.10 Continued

```
{  
    return this._OrderID;  
}  
set  
{  
    this._OrderID = value;  
}  
}  
  
[Column(Storage = "_ProductID", DbType = "Int NOT NULL", IsPrimaryKey = true)]  
public int? ProductID  
{  
    get  
    {  
        return this._ProductID;  
    }  
    set  
    {  
        this._ProductID = value;  
    }  
}  
  
[Column(Storage = "_UnitPrice", DbType = "Money NOT NULL")]  
public decimal? UnitPrice  
{  
    get  
    {  
        return this._UnitPrice;  
    }  
    set  
    {  
        this._UnitPrice = value;  
    }  
}  
  
[Column(Storage = "_Quantity", DbType = "SmallInt NOT NULL")]  
public short? Quantity  
{  
    get  
    {  
        return this._Quantity;  
    }  
    set  
    {  
        this._Quantity = value;  
    }  
}
```

LISTING 15.10 Continued

```
        }
    }

[Column(Storage = "_Discount", DbType = "Real NOT NULL")]
public float? Discount
{
    get
    {
        return this._Discount;
    }
    set
    {
        this._Discount = value;
    }
}

[Table(Name = "dbo.Orders")]
public partial class Order
{
    private int _OrderID;
    private string _CustomerID;
    private System.Nullable<int> _EmployeeID;
    private System.Nullable<System.DateTime> _OrderDate;
    private System.Nullable<System.DateTime> _RequiredDate;
    private System.Nullable<System.DateTime> _ShippedDate;
    private System.Nullable<int> _ShipVia;
    private System.Nullable<decimal> _Freight;
    private string _ShipName;
    private string _ShipAddress;
    private string _ShipCity;
    private string _ShipRegion;
    private string _ShipPostalCode;
    private string _ShipCountry;
    public Order()
}
```

LISTING 15.10 Continued

```
{  
}  
  
[Column(Storage = "_OrderID", AutoSync = AutoSync.OnInsert,  
    DbType = "Int NOT NULL IDENTITY", IsPrimaryKey = true, IsDbGenerated = true)]  
public int OrderID  
{  
    get  
    {  
        return this._OrderID;  
    }  
    set  
    {  
        this._OrderID = value;  
    }  
}  
  
[Column(Storage = "_CustomerID", DbType = "NChar(5)")]  
public string CustomerID  
{  
    get  
    {  
        return this._CustomerID;  
    }  
    set  
    {  
        this._CustomerID = value;  
    }  
}  
  
[Column(Storage = "_EmployeeID", DbType = "Int")]  
public System.Nullable<int> EmployeeID  
{  
    get  
    {  
        return this._EmployeeID;  
    }  
    set  
    {  
        this._EmployeeID = value;  
    }  
}  
  
[Column(Storage = "_OrderDate", DbType = "DateTime")]
```

LISTING 15.10 Continued

```
public System.Nullable<System.DateTime> OrderDate
{
    get
    {
        return this._OrderDate;
    }
    set
    {
        this._OrderDate = value;
    }
}

[Column(Storage = "_RequiredDate", DbType = "DateTime")]
public System.Nullable<System.DateTime> RequiredDate
{
    get
    {
        return this._RequiredDate;
    }
    set
    {
        this._RequiredDate = value;
    }
}

[Column(Storage = "_ShippedDate", DbType = "DateTime")]
public System.Nullable<System.DateTime> ShippedDate
{
    get
    {
        return this._ShippedDate;
    }
    set
    {
        this._ShippedDate = value;
    }
}

[Column(Storage = "_ShipVia", DbType = "Int")]
public System.Nullable<int> ShipVia
{
    get
    {
        return this._ShipVia;
    }
    set
```

LISTING 15.10 Continued

```
{  
    this._ShipVia = value;  
}  
}  
  
[Column(Storage = "_Freight", DbType = "Money")]  
public System.Nullable<decimal> Freight  
{  
    get  
    {  
        return this._Freight;  
    }  
    set  
    {  
        this._Freight = value;  
    }  
}  
  
[Column(Storage = "_ShipName", DbType = "NVarChar(40)")]  
public string ShipName  
{  
    get  
    {  
        return this._ShipName;  
    }  
    set  
    {  
        this._ShipName = value;  
    }  
}  
  
[Column(Storage = "_ShipAddress", DbType = "NVarChar(60)")]  
public string ShipAddress  
{  
    get  
    {  
        return this._ShipAddress;  
    }  
    set  
    {  
        this._ShipAddress = value;  
    }  
}  
  
[Column(Storage = "_ShipCity", DbType = "NVarChar(15)")]
```

LISTING 15.10 Continued

```
public string ShipCity
{
    get
    {
        return this._ShipCity;
    }
    set
    {
        this._ShipCity = value;
    }
}

[Column(Storage = "_ShipRegion", DbType = "NVarChar(15)")]
public string ShipRegion
{
    get
    {
        return this._ShipRegion;
    }
    set
    {
        this._ShipRegion = value;
    }
}

[Column(Storage = "_ShipPostalCode", DbType = "NVarChar(10)")]
public string ShipPostalCode
{
    get
    {
        return this._ShipPostalCode;
    }
    set
    {
        this._ShipPostalCode = value;
    }
}

[Column(Storage = "_ShipCountry", DbType = "NVarChar(15)")]
public string ShipCountry
{
    get
    {
        return this._ShipCountry;
    }
}
```

LISTING 15.10 Continued

```
    }
    set
    {
        this._ShipCountry = value;
    }
}
}
```

Note that because we are using mapped entity classes, yet really just classes, the LINQ statement doesn't make any special allowances for the data coming from the database. However, the entity classes are written a little differently. In addition to using the `TableAttribute` and `ColumnAttribute` in the entity classes, you have to allow for nulls in the underlying table itself. For this reason, use nullable types in your entity classes for value types, like `int` and `DateTime`.

Querying Views with LINQ

Views don't seem to be used in sample code on the Internet, in articles, or even in books that much. However, in real-world applications, the "view" is a very useful concept.

A view is a stored SQL statement that represents a snapshot of data. Generally, a view is created to capture some coherent representation of data, such as detailed customer order information. Often, views are thought of as read-only snapshots of data representing a join on one or more tables. The SQL view itself is read-only, but you can write code that gets data from a view and then separate code that uses that information to update the associated tables independently.

The first part of this section walks you through designing a view in Microsoft Visual Studio (which you can skip if you are comfortable with using Visual studio to design views), and the second part of this section demonstrates that a view can be mapped to an entity in C# and then queried using LINQ to SQL much as you would a table.

Building a View in SQL Server

To create a view based on two tables, you can start in Server Explorer in Visual Studio. To demonstrate, let's use the `Orders` and `Order Details` table of the Northwind database. Here are the numbered steps:

1. In Visual Studio, select View, Server Explorer.
2. Expand the Data Connections table (see Figure 15.1) and right-click on the Views node of the Northwind database.

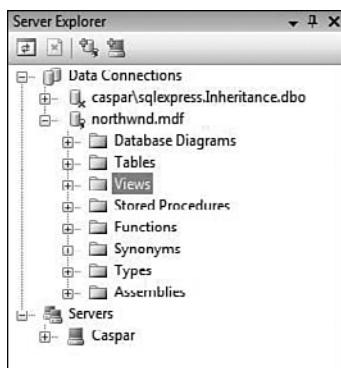


FIGURE 15.1 Right-click the Views node of Server Explorer and select Add View to design a view in Visual Studio.

3. The Add Table dialog box is displayed. Click the Orders table and the Order Details table, and click the Add button in the dialog box (refer to Figure 15.2). Close the Add Table dialog box.

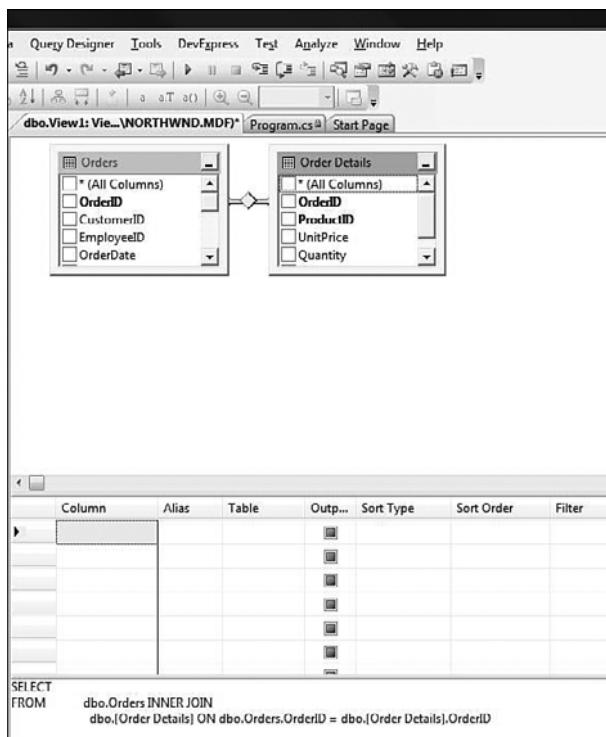


FIGURE 15.2 After the Orders and Order Details tables have been added to the diagram pane of the visual designer (for views).

4. When you have added the tables, the Visual Studio designer expresses the relationships based on natural primary and foreign key relationships. (You can use the automatically added relationship or define others by dragging and dropping columns between tables.)
5. Next, select the columns to be part of the resultset. For our example, select all columns from Orders and every column except the OrderID from Order Details.
6. Save and close the view, providing a meaningful name.

Listing 15.11 shows the SQL representing the view created in the preceding set of numbered steps.

LISTING 15.11 The SQL Representing the View Based on Northwind's Orders and Order Details as Generated from the Preceding Set of Numbered Steps

```

SELECT
    dbo.Orders.*,
    dbo.[Order Details].ProductID,
    dbo.[Order Details].UnitPrice,
    dbo.[Order Details].Quantity,
    dbo.[Order Details].Discount
FROM
    dbo.Orders
INNER JOIN
    dbo.[Order Details] ON dbo.Orders.OrderID = dbo.[Order Details].OrderID

```

Querying a View with LINQ to SQL

After defining a view, or from an existing view, you can write SELECT statements against the view as if it were a table. By defining an object-relational mapping, you can write LINQ queries against the view, too. The key here is to use the view name for the Name argument of the `TableAttribute`. Writing a LINQ select statement against an ORM, mapped to a view, looks just like the code for querying against a table. The example coded against the “Alphabetical list of products” view in the Northwind database is shown in Listing 15.12.

LISTING 15.12 Using the `TableAttribute` and `ColumnAttribute` as You Would for a Database Table to Map an Entity Class to a View in Your Database

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data.Linq;
using System.Data.Linq.Mapping;
using System.Reflection;

```

LISTING 15.12 Continued

```
namespace LinqtoSqlView
{
    class Program
    {
        static void Main(string[] args)
        {
            Northwind northwind = new Northwind();

            var products = from product in northwind.Products
                           select product;

            string line = new string('-', 40);
            Array.ForEach(products.ToArray(), r =>
            {
                Console.WriteLine(r);
                Console.WriteLine(line);
                Console.WriteLine();
            });

            Console.ReadLine();
        }
    }

    public class Northwind : DataContext
    {
        private static readonly string connectionString =
            "Data Source=.\SQLEXPRESS;AttachDbFilename=\""
            + "C:\\Books\\Sams\\LINQ\\Northwind\\northwnd.mdf\"; "
            + "Integrated Security=True;Connect Timeout=30;User Instance=True";

        public Northwind()
            : base(connectionString)
        {
            Log = Console.Out;
        }

        public Table<ProductList> Products
        {
            get { return this.GetTable<ProductList>(); }
        }
    }

    [Table(Name = "Alphabetical list of products")]
    public class ProductList
    {
```

LISTING 15.12 Continued

```
[Column()]
public int ProductID { get; set; }

[Column()]
public string ProductName{ get; set; }

[Column()]
public int SupplierID { get; set; }

[Column()]
public int CategoryID { get; set; }

[Column()]
public string QuantityPerUnit { get; set; }

[Column()]
public decimal UnitPrice { get; set; }

[Column()]
public Int16 UnitsInStock { get; set; }

[Column()]
public Int16 UnitsOnOrder { get; set; }

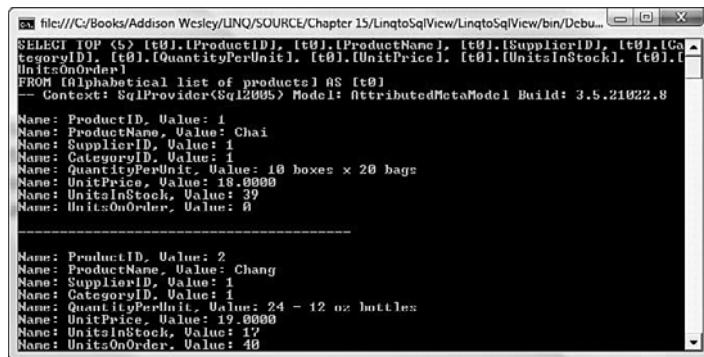
public override string ToString()
{
    StringBuilder builder = new StringBuilder();
    PropertyInfo[] info = this.GetType().GetProperties();

    Array.ForEach(info, i =>
    {
        Console.WriteLine("Name: {0}, Value: {1}",
            i.Name,
            i.GetValue(this, null) == null ? "none" :
            i.GetValue(this, null));
    });
}

return builder.ToString();
}
```

LINQ to SQL can determine by the `Name` argument of the `TableAttribute` that it is working with a view, and LINQ generates the code to query the view. The output from the

code, including the LINQ output sent to the console, shows the query treats the view as it would a table by placing the view name in the `from` clause of the generated SQL (refer to Figure 15.3).



The screenshot shows a Windows Task Manager window with the title bar "file:///C:/Books/Addison Wesley/LINQ/SOURCE/Chapter 15/LinqtoSqlView/LinqtoSqlView/bin/Debug/AlphabeticalList.exe". The main content area displays the following SQL query and its execution results:

```

SELECT TOP (5) [t0].[ProductID], [t0].[ProductName], [t0].[SupplierID], [t0].[CategoryID], [t0].[QuantityPerUnit], [t0].[UnitPrice], [t0].[UnitsInStock], [t0].[UnitsOnOrder]
FROM (Alphabetical list of products) AS [t0]
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build: 3.5.21022.8
Name: ProductID, Value: 4
Name: ProductName, Value: Chai
Name: SupplierID, Value: 1
Name: CategoryID, Value: 1
Name: QuantityPerUnit, Value: 10 boxes x 20 bags
Name: UnitPrice, Value: 18.0000
Name: UnitsInStock, Value: 39
Name: UnitsOnOrder, Value: 0

Name: ProductID, Value: 2
Name: ProductName, Value: Chang
Name: SupplierID, Value: 1
Name: CategoryID, Value: 1
Name: QuantityPerUnit, Value: 24 - 12 oz bottles
Name: UnitPrice, Value: 19.0000
Name: UnitsInStock, Value: 17
Name: UnitsOnOrder, Value: 40

```

FIGURE 15.3 The LINQ to SQL provider properly uses the view as the source for the `from` clause of the SQL statement.

Databinding with LINQ to SQL

The key to databinding is `IEnumerable`. If you trace through the .NET source or disassembled .NET assemblies far enough, you will see that binding uses `IEnumerable` and reflection to read public property names and bind these and their values to bindable controls, like a `GridView`.

Bindability is also supported by classes that implement `IBindingList`. `IBindingList`, in turn, implements the `IEnumerable` interface. For instance, LINQ sequences return `IEnumerable<T>` objects, so these are automatically bindable. `EntitySets`, as described in Chapter 14, “Creating Better Entities and Mapping Inheritance and Aggregation,” are used for properties of entities that themselves represent mapped relationships. `EntitySets` also implement `IEnumerable`, so these are bindable, too.

Listing 15.13 represents the code behind of a web page. Most of the code is an ORM representing the “Alphabetical list of products” view, and the `Page_Load` method shows that with the `DataSource` and `DataBind` properties of bindable controls (`GridView`, in the example), you can bind the output from a LINQ query right to a bindable control. (To implement the example, simply place a `GridView` on a web page and add the code from Listing 15.13.)

LISTING 15.13 Bindability Is Supported By `IEnumerable`; `IEnumerable` Is at the Heart of All of the Sequences Returned from LINQ So Binding Is Directly Supported Without Any Fanfare

```

using System;
using System.Collections;
using System.Configuration;

```

LISTING 15.13 Continued

```
using System.Data;
using System.Linq;
using System.Web;
using System.Web.Security;
using System.Web.UI;
using System.Web.UI.HtmlControls;
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;
using System.Xml.Linq;
using System.Data.Linq;
using System.Data.Linq.Mapping;

namespace LinqToSqlDatabinding
{
    public partial class _Default : System.Web.UI.Page
    {
        protected void Page_Load(object sender, EventArgs e)
        {
            Northwind northwind = new Northwind();

            var products = from product in northwind.Products
                           select product;

            GridView1.DataSource = products;
            GridView1.DataBind();
        }
    }

    public class Northwind : DataContext
    {
        private static readonly string connectionString =
            "Data Source=.\SQLExpress;AttachDbFilename=c:\\temp\\northwnd.mdf;" +
            "Integrated Security=True;Connect Timeout=30;User Instance=True";

        public Northwind()
            : base(connectionString)
        {
            Log = Console.Out;
        }

        public Table<ProductList> Products
        {
            get { return this.GetTable<ProductList>(); }
        }
    }
}
```

LISTING 15.13 Continued

```
[Table(Name = "Alphabetical list of products")]
public class ProductList
{
    [Column()]
    public int ProductID { get; set; }

    [Column()]
    public string ProductName{ get; set; }

    [Column()]
    public int SupplierID { get; set; }

    [Column()]
    public int CategoryID { get; set; }

    [Column()]
    public string QuantityPerUnit { get; set; }

    [Column()]
    public decimal UnitPrice { get; set; }

    [Column()]
    public Int16 UnitsInStock { get; set; }

    [Column()]
    public Int16 UnitsOnOrder { get; set; }

}
}
```

15

Summary

There are two ways to use LINQ with ADO.NET (so far). You can use LINQ to DataSets and query DataTables. This approach is great for existing code and works because ADO.NET DataTables are associated with extension methods that provide access to enumerability for DataTables and underlying field values. After you have enumerability and access to field values, you can define join relationships. The second way ADO.NET is supported is through LINQ to SQL. LINQ to SQL depends on mapping entity classes to database tables, but after being mapped, these entity classes are just classes with a persistence layer and LINQ behaviors are supported behaviors like joins.

The biggest difference where LINQ, ADO.NET, and joins are concerned is that the column in databases can contain nulls. To account for the possibility of a null, define your entities as nullable types. (Also, define fields in projections as nullable types.) After you have

dealt with nullable fields, querying databases via LINQ is consistent with querying custom objects.

The key to success is not to rewrite existing code to support LINQ; rather use LINQ to work in conjunction with the style of code you have or will have. And, you always have the option to define joins in SQL, too. The choice is not an either/or choice. Choosing LINQ joins or SQL joins is a matter of suitability to the problem at hand, which, in turn, is a matter of preference.

CHAPTER 16

Updating Anonymous Relational Data

“A prudent question is one-half of wisdom.”

—Francis Bacon

There are a couple of challenges when updating data. The data has to be inserted, deleted, or modified. Related data has to be managed as an atomic operation, called atomicity, and often multiple users might update the same data at the same time—concurrency—and you have to figure out what to do when updates are in conflict.

This chapter demonstrates how to update data directly, replace default modifications with user-defined modifications and stored procedures, how to provide for atomicity—multiple updates treated as one—and concurrent update conflicts. These are all supported by LINQ.

Adding and Removing Data

The simplest applications simply make modifications to an underlying persistence layer, the database. Data is read and/or inserted and data is modified or deleted. Referred to in a general sense as changing stored data, this section shows you how to read data and change or delete it and how to insert new data. After this section, you will build this knowledge by exploring support for more challenging kinds of changes to an underlying data store.

Inserting Data with LINQ to SQL

LINQ to SQL is designed so that you define entity classes, manually, with SqlMetal, or the Visual Designer, and then

IN THIS CHAPTER

- ▶ Adding and Removing Data
- ▶ Calling User-Defined Functions
- ▶ Using Transactions
- ▶ Understanding Conflict Resolution
- ▶ N-Tier Applications and LINQ to SQL

interact with these mapped entities like regular classes. To that end, inserting data becomes a process of initializing a new object and inserting it into the data store using methods provided as part of the LINQ infrastructure.

The central player in submitting changes back to a database is the `DataContext.SubmitChanges` method. Listing 16.1 contains an entity class mapped to the Northwind *Customers* table. The code constructs a new customer and inserts it into the database. All of the underlying plumbing for the insert is automatically defined for you by LINQ.

LISTING 16.1 Inserting a New Customer into the Customers Table

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data.Linq;
using System.Data.Linq.Mapping;

namespace LinqToSqlAddRemoveData
{
    class Program
    {
        static void Main(string[] args)
        {
            Northwind northwind = new Northwind();
            Table<Customer> customers = northwind.GetTable<Customer>();

            // insert a new customer
            Customer customer = new Customer();
            customer.CustomerID = "DUSTY";
            customer.CompanyName = "Dusty's Cellars";
            customer.ContactName = "Dena Swanson";
            customer.ContactTitle = "Catering Manager";
            customer.Address = "1839 Grand River Avenue";
            customer.City = "Okemos";
            customer.Region = "MI";
            customer.PostalCode = "48864";
            customer.Country = "US";
            customer.Phone = "(517) 349-5150";
            customer.Fax = "(517) 349-8416";

            customers.InsertOnSubmit(customer);

            northwind.SubmitChanges();
        }
    }
}
```

LISTING 16.1 Continued

```
public class Northwind : DataContext
{
    private static readonly string connectionString =
        "Data Source=.\SQLExpress;AttachDbFilename="" +
        "C:\Books\Sams\LINQ\Northwind\northwnd.mdf"" +
        ";Integrated Security=True;Connect Timeout=30;User Instance=True";

    public Northwind() : base(connectionString)
    {
        Log = Console.Out;
    }
}

[Table(Name="Customers")]
public class Customer
{
    [Column(IsPrimaryKey=true)]
    public string CustomerID{ get; set; }

    [Column()]
    public string CompanyName{ get; set; }

    [Column()]
    public string ContactName{ get; set; }

    [Column()]
    public string ContactTitle{ get; set; }

    [Column()]
    public string Address{ get; set; }

    [Column()]
    public string City{ get; set; }

    [Column()]
    public string Region{ get; set; }

    [Column()]
    public string PostalCode{ get; set; }

    [Column()]
    public string Country{ get; set; }

    [Column()]
    public string Phone{ get; set; }

    [Column()]
    public string Fax{ get; set; }
}
```

In Listing 16.1, a *Customer* entity is defined and mapped to the *Customers* table. The *Main* function constructs a custom *DataContext*, named *Northwind*, and requests the *Customers*

data from the database. A new `Customer` object is constructed, initialized, and added to the `customers` sequence with `InsertOnSubmit`. Calling `DataContext.SubmitChanges()` writes the changes back to the database.

In this scenario, you have some overhead because `GetTable` returns the `Customers` data, which you don't need to explicitly do if you just want to insert. You also have the option of calling `DataContext.ExecuteCommand`, passing the name of a stored procedure and the parameters to `ExecuteCommand`.

In this scenario, the `Console` stream is mapped to the `DataContext.Log` property, and you will see that LINQ can easily generate the appropriate SQL statement to perform the insert.

Deleting Data with LINQ to SQL

Data can be deleted from a sequence by calling `Table<T>.DeleteOnSubmit`. (The pattern of submitting changes is `operationOnSubmit` or `operationAllOnSubmit`.) In Listing 16.2, again you use a mapped entity, a `DataContext`, and a sequence of `Table` objects. From the sequence, request the objects you want to delete. Call `DeleteOnSubmit` or `DeleteAllOnSubmit` and `SubmitChanges`.

In Listing 16.2, the generic method `Single` is used to return a single `Customer` object. If `Single` returns no data, an exception is thrown; this is why you need to use the exception handler. Listing 16.2 uses the same mapped entity and `DataContext` as Listing 16.1.

LISTING 16.2 Selecting a Single Object from the Customers Table and Deleting It; Protecting Calls to Single with an Exception Handler in Case the Resultset Is Empty

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data.Linq;
using System.Data.Linq.Mapping;

namespace LINQtoSQLDeleteData
{
    class Program
    {
        static void Main(string[] args)
        {
            Northwind northwind = new Northwind();
            Table<Customer> customers = northwind.GetTable<Customer>();

            try
            {
                var remove = customers.Single(cust => cust.CustomerID == "DUSTY");
                customers.DeleteOnSubmit(remove);
                northwind.SubmitChanges();
            }
        }
    }
}
```

LISTING 16.2 Continued

```
        }
    catch
    {
        Console.WriteLine("Nothing to do.");
    }
}

public class Northwind : DataContext
{
    private static readonly string connectionString =
        "Data Source=.\SQLExpress;AttachDbFilename=\"\" +
        "C:\Books\Sams\LINQ\Northwind\northwnd.mdf\"\" +
        ";Integrated Security=True;Connect Timeout=30;User Instance=True";

    public Northwind() : base(connectionString)
    {
        Log = Console.Out;
    }
}

[Table(Name="Customers")]
public class Customer
{
    [Column(IsPrimaryKey=true)]
    public string CustomerID{ get; set; }
    [Column()]
    public string CompanyName{ get; set; }
    [Column()]
    public string ContactName{ get; set; }
    [Column()]
    public string ContactTitle{ get; set; }
    [Column()]
    public string Address{ get; set; }
    [Column()]
    public string City{ get; set; }
    [Column()]
    public string Region{ get; set; }
    [Column()]
    public string PostalCode{ get; set; }
    [Column()]
    public string Country{ get; set; }
    [Column()]
    public string Phone{ get; set; }
```

LISTING 16.2 Continued

```
[Column()]
public string Fax{ get; set; }
}
```

In case you skipped the chapter on Lambda Expressions (Chapter 5, “Understanding Lambda Expressions and Closures”), the argument `cust => cust.CustomerID == "DUSTY"` is a Lambda Expression—think very terse function notation where `cust` is the input parameter and `cust.CustomerID == "DUSTY"` is equivalent to a `return` statement that returns a Boolean result for the test equals “DUSTY”.

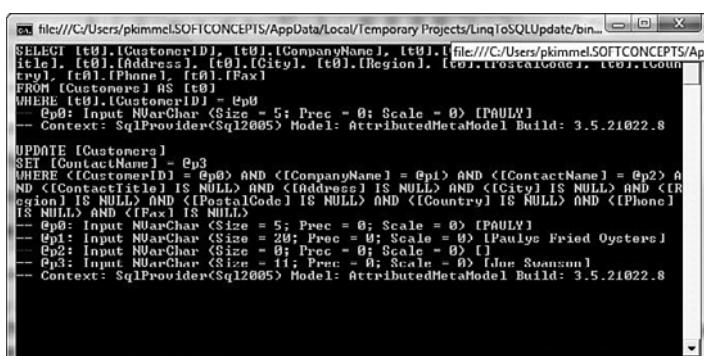
NOTE

By the way, Dusty’s Cellar is an exceptional fine foods shop and restaurant in Okemos, Michigan. (I buy my cigars from Dusty’s, which is my reward while writing.) See, we try to educate and inform here.

It really doesn’t matter how you get the objects you want to delete. You can use any of the query variations presented in this book to get the data. The interchange is the same: Call `DeleteOnSubmit` for the `Table<T>` collection and `SubmitOnChanges`.

Updating Data with LINQ to SQL

Using the elements from Listing 16.1 and Listing 16.2, you can modify the `Main` function to update the database. In Listing 16.3, the Lambda Expression was changed to select `CustomerID "PAULY"` and modify the `ContactName`. Wild man Joe Swanson is the contact for Pauly’s Fried Oysters. As shown in Figure 16.1, LINQ generates the `select` statement for the `Single` method and the update, which is implicit because the `ContactName` is modified.



```
file:///C:/Users/pkimmel/SOFTCONCEPTS/AppData/Local/Temporary Projects/LinqToSQLUpdate/bin... X
SELECT [t0].[CustomerID], [t0].[CompanyName], [t0].[file:///C:/Users/pkimmel/SOFTCONCEPTS/App
title], [t0].[Address], [t0].[City], [t0].[Region], [t0].[PostalCode], [t0].[Country]
FROM [Customers] AS [t0]
WHERE [t0].[CustomerID] = @p0
-- @p0: Input NVarChar <Size: 5; Prec: 0; Scale: 0> [PAULY]
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build: 3.5.21022.8

UPDATE [Customers]
SET [ContactName] = @p3
WHERE <[CustomerID] = @p1> AND <[CompanyName] = @p1> AND <[ContactName] = @p2>
AND <[CompanyName] IS NULL> AND <[Address] IS NULL> AND <[City] IS NULL> AND <[Region]
IS NULL> AND <[PostalCode] IS NULL> AND <[Country] IS NULL> AND <[Phone]
IS NULL> AND <[Fax] IS NULL>
-- @p0: Input NVarChar <Size: 5; Prec: 0; Scale: 0> [PAULY]
-- @p1: Input NVarChar <Size: 20; Prec: 0; Scale: 0> [Paulys Fried Oysters]
-- @p2: Input NVarChar <Size: 0; Prec: 0; Scale: 0> []
-- @p3: Input NVarChar <Size: 11; Prec: 0; Scale: 0> [Joe Swanson]
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build: 3.5.21022.8
```

FIGURE 16.1 LINQ generates the default SELECT and UPDATE SQL statements.

LISTING 16.3 Modifying Data Is as Simple as Changing an Object and Calling SubmitChanges

```
static void Main(string[] args)
{
    Northwind northwind = new Northwind();
    Table<Customer> customers = northwind.GetTable<Customer>();

    try
    {
        var changeOne = customers.Single(cust => cust.CustomerID == "PAULY");
        changeOne.ContactName = "Joe Swanson";
        northwind.SubmitChanges();
    }
    catch
    {
        Console.WriteLine("Nothing to do.");
    }
}
```

If you run this query multiple times, LINQ is smart enough to know that the `ContactName` has been modified already and it won't run the `UPDATE` statement subsequent times (unless the field value is actually different).

Using Stored Procedures

Just a few short years ago, it was considered a best practice to use stored procedures rather than dynamic queries in your code. At every conference that includes discussions on SQL, Microsoft and noted experts repeat the message that dynamic queries and stored procedures heuristics (performance characteristics) are close enough that you can use either programming style. I generally preferred stored procedures because it is a natural division-point of labor: Programmers write C# or VB .NET and great SQL programmers write optimized stored procedures.

As demonstrated and illustrated in Figure 16.1, LINQ clearly writes dynamic queries for you. You can elect to replace these default procedures with stored procedures by providing `Insert`, `Update`, and `Delete` methods in your custom data context. For example, to use your stored procedure instead of the default generated SQL, follow these steps:

1. Define a stored procedure or use an existing one. (For this exercise, you can use `UpdateCustomer`, which is defined in the Northwind database.)
2. Define a private method `UpdateCustomer` in the custom `DataContext` class that accepts a `Customer` object.
3. Define a public procedure `UpdateCustomer` in the `DataContext` class that contains the parameters needed to perform the update.
4. Call the second public `UpdateCustomer` from the first.

5. Annotate the verbose `UpdateCustomer` method with the `FunctionAttribute` and the `Name` argument, indicating the procedure to use.
6. Annotate each of the parameters with the `ParameterAttribute` defining the column name and `DbType`.
7. Compile and run the code.

You will see that the additional code in `DataContext` is run instead of a dynamic query. Using the update example from Listing 16.3, Listing 16.4 shows the modified `DataContext` class with the two new methods.

LISTING 16.4 Adding an `UpdateObject`—For Example, the `UpdateCustomer` Method—to the `DataContext` Class to Override the Default SQL Generation Behavior and Use Your Stored Procedures Instead

```
public class Northwind : DataContext
{
    private static readonly string connectionString =
        "Data Source=.\SQLExpress;AttachDbFilename=\" + "
        "C:\\Books\\Sams\\LINQ\\Northwind\\northwnd.mdf\" + "
        ";Integrated Security=True;Connect Timeout=30;User Instance=True";

    public Northwind() : base(connectionString)
    {
        Log = Console.Out;
    }

    private void UpdateCustomer(Customer obj)
    {
        this.UpdateCustomer(obj.CustomerID, obj.CompanyName, obj.ContactName,
            obj.ContactTitle,
            obj.Address, obj.City, obj.Region, obj.PostalCode,
            obj.Country, obj.Phone, obj.Fax);
    }

    [Function(Name="dbo.UpdateCustomer")]
    public int UpdateCustomer(
        [Parameter(Name="CustomerID", DbType="NChar(5)")]
        string customerID,
        [Parameter(Name="CompanyName", DbType="NVarChar(40)")]
        string companyName,
        [Parameter(Name="ContactName", DbType="NVarChar(30)")]
        string contactName,
        [Parameter(Name="ContactTitle", DbType="NVarChar(30)")]
        string contactTitle,
        [Parameter(Name="Address", DbType="NVarChar(60)")]
        string address,
        [Parameter(Name="City", DbType="NVarChar(15)")]
        string city,
```

LISTING 16.4 Continued

```
[Parameter(Name="Region", DbType="NVarChar(15)")] string region,
[Parameter(Name="PostalCode", DbType="NVarChar(10)")] string postalCode,
[Parameter(Name="Country", DbType="NVarChar(15)")] string country,
[Parameter(Name="Phone", DbType="NVarChar(24)")] string phone,
[Parameter(Name="Fax", DbType="NVarChar(24)")] string fax)
{
    IExecuteResult result = this.ExecuteMethodCall(this,
        ((MethodInfo)(MethodInfo.GetCurrentMethod())), customerID,
        companyName, contactName,
        contactTitle, address, city, region, postalCode, country, phone, fax);
    return ((int)(result.ReturnValue));
}
}
```

As you can see from the example, the `FunctionAttribute` names the stored procedure and the `ParameterAttributes` describe and map the arguments to table columns. The function body of `UpdateCustomer` actually uses reflection—`MethodInfo.GetCurrentMethod`—to figure out the method to call. If the stored procedure returns a value, then it obtained it from the `IExecuteResult` interface. Listing 16.5 shows the stored procedure from the Northwind database.

LISTING 16.5 The `UpdateCustomer` Stored Procedure That Is Called Based on the Code in Listing 16.4

```
ALTER PROCEDURE dbo.UpdateCustomer
(
    @CustomerID nchar(5),
    @CompanyName nvarchar(40),
    @ContactName nvarchar(30),
    @ContactTitle nvarchar(30),
    @Address nvarchar(60),
    @City nvarchar(15),
    @Region nvarchar(15),
    @PostalCode nvarchar(10),
    @Country nvarchar(15),
    @Phone nvarchar(24),
    @Fax nvarchar(24)
)
AS
UPDATE CUSTOMERS
SET
    CompanyName = @CompanyName,
    ContactName = @ContactName,
```

LISTING 16.5 Continued

```
Address = @Address,
City = @City,
Region = @Region,
PostalCode = @PostalCode,
Country = @Country,
Phone = @Phone,
Fax = @Fax
WHERE CustomerID = @CustomerID

RETURN @@ROWCOUNT
```

Defining a Custom Stored Procedure

To some extent, it is assumed most readers will have a basic knowledge of things like defining stored procedures. SQL is its own language, but if you are comfortable with stored procedures, you can skip to the next section “Mapping a Stored Procedure with the LINQ to SQL Visual Designer.” If you need a quick refresher, read on.

A SQL procedure is very similar in structure to a function. The keyword ALTER or CREATE followed by PROCEDURE and a stored procedure name represents the procedure header. In parentheses, as is true with C# functions, are the comma-delimited parameter names and types. Remember to use SQL types like nvarchar for a flexible string. SQL stored procedure parameters are prefixed with the @ character. Following the parenthetical parameters is the AS keyword and then basically it’s a SQL query after that. SQL does support local variables and statements, but that’s a book all its own.

To define a procedure that inserts a *Customer* object, you need the parameters for a *Customer* row and an *insert* statement. The difference between dynamic SQL and a procedure is that you can get a bit fancier in the stored procedure, at least usually a bit fancier than any one tries with inline SQL text.

Listing 16.6 is a modified stored procedure from the Northwind database. The procedure is basically an *insert* statement with a little wiggle code. A while loop is used to avoid *CustomerID* key collisions by adding a number from 1 to 9 as the last character in the ID if a key collision occurs. A call to *UPPER* is used to ensure that *CustomerIDs* are all uppercase. You can get the parameter types from the table definition (or the Properties window itself).

LISTING 16.6 A SQL Stored Procedure That Tries to Ensure a Unique CustomerID and One That Is All Uppercase

```
ALTER PROCEDURE dbo.InsertCustomer
(
    @CustomerID      nchar(5) OUTPUT,
```

LISTING 16.6 Continued

```
@CompanyName      nvarchar(40),
@ContactName     nvarchar(30),
@ContactTitle    nvarchar(30),
@Address        nvarchar(60),
@City           nvarchar(15),
@Region         nvarchar(15),
@PostalCode     nvarchar(10),
@Country        nvarchar(15),
@Phone          nvarchar(24),
@Fax            nvarchar(24)
)
AS
DECLARE @COUNTER AS INT
SET @COUNTER = 1
SET @CustomerID = LEFT(@CompanyName, 5)
WHILE(@COUNTER < 10)
BEGIN
    IF NOT EXISTS (SELECT CustomerID
                    FROM CUSTOMERS WHERE CustomerID = @CustomerID)
        BREAK
    SET @CustomerID = LEFT(@CompanyName, 4) +
                      CAST(@COUNTER As NVarChar(10))
    SET @COUNTER = @COUNTER + 1
END

IF(@COUNTER > 9)
BEGIN
    RAISERROR('Error generating a unique customer id', 16, 1)
END

SET @CustomerID = UPPER(@CustomerID)

INSERT INTO CUSTOMERS
(
    CustomerID,
    CompanyName,
    ContactName,
    ContactTitle,
    Address,
    City,
    Region,
    PostalCode,
```

LISTING 16.6 Continued

```
Country,
Phone,
Fax
)
VALUES
(
@CustomerID,
@CompanyName,
@ContactName,
@ContactTitle,
@Address,
@City,
@Region,
@PostalCode,
@Country,
@Phone,
@Fax
)

```

```
RETURN @@ROWCOUNT
```

To define stored procedures in Microsoft Visual Studio, you need a Professional or Enterprise Edition of Visual Studio. Open Server Explorer and expand the database and Stored Procedures nodes. Right-click the Stored Procedures node and select Add New Stored Procedure (or perform the same function through the Data, Add New, Stored Procedure menu item); refer to Figure 16.2. The designer will stub out the stored procedure, as illustrated in Figure 16.2.

Next, let's use the designer to do the heavy lifting of mapping the procedure to the `DataContext`'s insert behavior.

Mapping a Stored Procedure with the LINQ to SQL Visual Designer

Designers and integrated tasks are designed to make detailed tasks easier to perform. For example, the Visual LINQ to SQL designer generates a custom `DataContext` and object-relational models (ORMs) for you. However, tools also have to be written in a general way and often produce very verbose results. That's okay if you want verbose results.

Consider using designers to learn how the plumbing of technologies work or to sanity check the code you have written. Knowing, however, that some people prefer designers, this example shows how you can map stored procedures to override LINQ-generated behavior. (For a first look at the LINQ to SQL designer, refer to Chapter 14, "Creating Better Entities and Mapping Inheritance and Aggregation.")

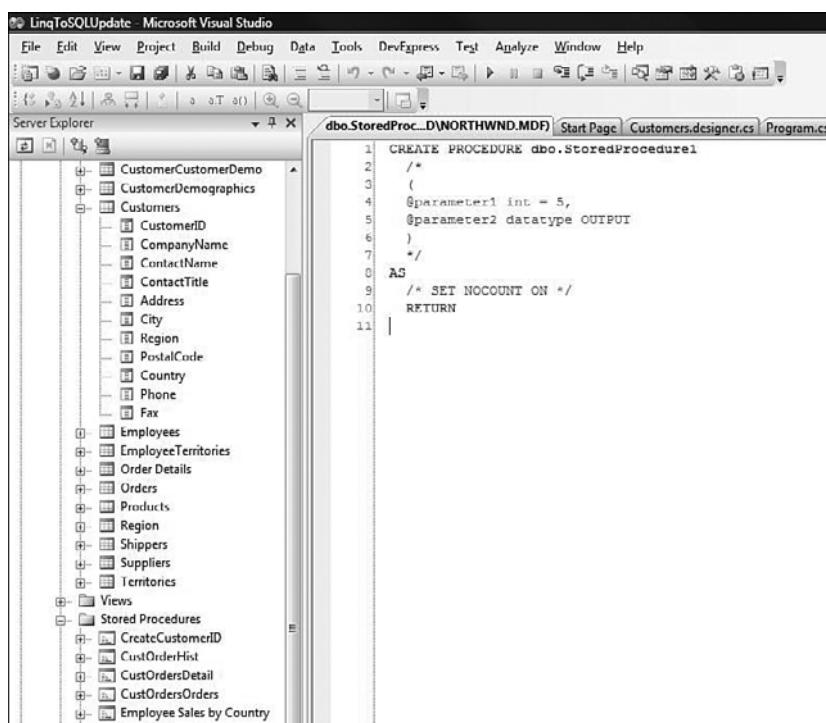


FIGURE 16.2 Visual Studio has excellent integrated support for SQL support, such as coding and executing stored procedures.

16

To map a stored procedure to override the default behavior with the LINQ to SQL designer—which is the visual way to complete the same task as described in the last section—follow these steps:

1. Add a .dbml file to your project by selecting the LINQ to SQL Classes template from the Add New Item dialog box.
2. In the designer, drag and drop a table (use *Customers* from Northwind) from Server Explorer to the surface of the designer.
3. Expand the Stored Procedures node and drag and drop the *InsertCustomers* sproc (pronounced *sproc*, short for stored procedure) to the methods pane of the designer (on the right side, by default).
4. Right-click the *Customers* table in the designer and select Configure Behavior.
5. In the Configure Behavior dialog box, make sure the *Customer* class is selected.
6. In the Behavior drop-down, make sure Insert is selected and choose the Customize radio button (see Figure 16.3).

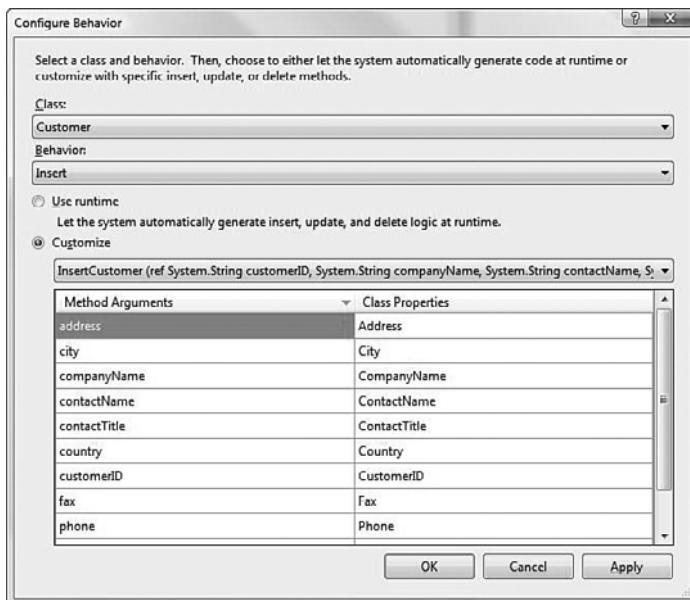


FIGURE 16.3 The Configure Behavior dialog box is used to map custom stored procedures to behaviors like insert that would otherwise be dynamically generated by the LINQ infrastructure.

7. Select the stored procedure (from Listing 16.6) that you just added to the methods pane.
8. Click OK.

After you have added the stored procedure to the desired behavior, the designer fills in the `DataContext` methods. In this instance, two `InsertCustomer` methods are added. The `InsertCustomer` method-pair works together just like `UpdateCustomer` from the previous section. Listing 16.7 shows how easy it is to use the customer stored procedure with LINQ to SQL.

LISTING 16.7 After Mapping a Custom Stored Procedure to a Behavior You Do Not Need to Change Your LINQ to SQL Code; LINQ Knows Which Method to Call By the Presence of the Configured (or Written) Code in the DataContext

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace LinqToSQLUsingStoredProcedure
{
    class Program
    {

```

LISTING 16.7 Continued

```
static void Main(string[] args)
{
    CustomersDataContext context = new CustomersDataContext();

    Customer customer = new Customer();
    customer.CustomerID = "temp";
    customer.CompanyName = "Pauly's Fried Oysters";

    context.Customers.InsertOnSubmit(customer);
    context.SubmitChanges();

    Console.WriteLine("Insert: {0}", customer.CustomerID);
    Console.ReadLine();

}
}
```

Calling User-Defined Functions

16

I didn't know until recently that Truman Capote wrote *Breakfast at Tiffany's*. Liking the ditzy character of Holly Golightly, the song *Moon River*, the mad reds, Sally Tomato, the martini parties, and the song of the same name from Blues Traveler, I am reminded of the scene where George Peppard's character takes Audrey Hepburn's character to Tiffany's to buy something. Fred (Peppard) and Holly (Hepburn) are looking for something under \$10.

The gist of the scene is that Holly "doesn't like diamonds" and she only has \$10 to spend. The salesman at Tiffany's carefully and tactfully gets to the subject of "budget." The clerk offers a sterling silver telephone dialer for \$6.75. Finally, Fred and Holly resolve on having a prize (a ring, if memory serves) from a Cracker Jack box engraved. (The writing is truly brilliant.)

So, for Holly and Fred and Sally Tomato, I will demonstrate calling a user-defined function by using the `Northwind.ProductsUnderThisUnitPrice` function. (Had PCs and SQL user functions existed in the 1960s, perhaps the clerk could have offered better alternatives to a sterling phone dialer and an engraved Cracker Jack prize.)

The key to calling a user function is to map a function in the `DataContext` subclass—in our example, the `Northwind` class—to the user function in the database. Attribute the function name with the `FunctionAttribute` and name of the SQL user-defined function, and attribute the arguments with the `ParameterAttribute`. This is the same way you mapped stored procedures in Listing 16.4. Listing 16.8 demonstrates mapping the `Products` table and invoking the `ProductsUnderThisUnitPrice` user function.

LISTING 16.8 SQL User-Defined Functions Can Be Mapped Using the FunctionAttribute and ParameterAttribute and Then Called Like Any Other Function

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data.Linq;
using System.Data.Linq.Mapping;
using System.Reflection;

namespace LinqtoSQLUserFunction
{
    class Program
    {
        static void Main(string[] args)
        {
            Northwind northwind = new Northwind();

            var under10 =
                from product in northwind.ProductsUnderThisUnitPrice(10)
                select product;

            Array.ForEach(under10.ToArray(), r =>
            {
                Console.WriteLine(r);
                Console.WriteLine(new string('-', 40));
                Console.WriteLine();
            });
            Console.ReadLine();
        }
    }

    public class Northwind : DataContext
    {
        private static readonly string connectionString =
            "Data Source=.\SQLExpress;AttachDbFilename=" +
            "c:\temp\northwnd.mdf;Integrated Security=True;" +
            "Connect Timeout=30;User Instance=True";

        public Northwind() : base(connectionString) { }

        [Function(Name="dbo.ProductsUnderThisUnitPrice", IsComposable=true)]
        public IQueryable<Product> ProductsUnderThisUnitPrice(
            [Parameter(DbType="Money")] System.Nullable<decimal> price)
        {
            return this.CreateMethodCallQuery<Product>(this,
```

LISTING 16.8 Continued

```
((MethodInfo)(MethodInfo.GetCurrentMethod()), price);  
}  
}  
  
[Table(Name="Products")]  
public class Product  
{  
    [Column()]  
    public int ProductID{ get; set; }  
    [Column()]  
    public string ProductName{ get; set; }  
    [Column()]  
    public int? SupplierID{ get; set; }  
    [Column()]  
    public int? CategoryID{ get; set; }  
    [Column()]  
    public string QuantityPerUnit{ get; set; }  
    [Column()]  
    public decimal? UnitPrice{ get; set; }  
    [Column()]  
    public short? UnitsInStock{ get; set; }  
    [Column()]  
    public short? UnitsOnOrder{ get; set; }  
    [Column()]  
    public short? ReorderLevel{ get; set; }  
    [Column()]  
    public bool? Discontinued { get; set; }  
  
    public override string ToString()  
{  
        StringBuilder builder = new StringBuilder();  
        PropertyInfo[] info = this.GetType().GetProperties();  
  
        Array.ForEach(info, i=>  
        {  
            builder.AppendFormat("Name: {0}, Value: {1}\n",  
                i.Name,  
                i.GetValue(this, null) == null ? "none" :  
                i.GetValue(this, null));  
        });  
  
        return builder.ToString();  
    }  
}
```

If you use the SQL to Classes designer to map SQL user-defined functions to C# functions, then understand that the designer will add a class named after the user-function. The additional class will contain properties that map to the resultset and the mapped function will be placed in the new class. Because `ProductsUnderThisUnitPrice` actually returns a sequence of `Products` (and you are coding the entity class), you can just place the mapped function in the `DataContext` class and use the product entity class as the parameterized type of the function's return value (as shown in Listing 16.8).

Using Transactions

In all of my books and articles, I strongly encourage readers to keep up with innovations in technology and programming languages. Like building an onion from the inside out, new technologies build on existing technologies. So, someone who has skipped anonymous delegates is going to have a harder time making the leap from delegates to Lambda Expressions than someone who has already mastered anonymous delegates. Someone who has already mastered generics will understand generic extension methods and generic delegates more readily than someone trying to get generics for the first time.

ADO.NET 1.0 and ADO.NET 1.1 supported transactions through the `Transaction` class. ADO.NET introduced the `TransactionScope`. `TransactionScope` has a broadened definition that enlists anything that can be enlisted in a transaction. This capability extends across technologies like SQL Server and COM+.

The basic way to use a `TransactionScope` is to create the object in a `using` block. Write the code that is part of the transaction such as calls to SQL Server or, in this case, LINQ to SQL code that implicitly calls SQL Server, and before the `using` statement is exited, call `TransactionScope.Complete`. If an unhandled exception occurs, the `using` block exits before `Commit` is called and the transaction is rolled back. Using the `TransactionScope` results in fewer lines of code than using the `Transaction` class. (To use the `TransactionScope`, make sure you reference the `System.Transaction` assembly.)

Listing 16.9 deletes all customers and orders with a specific `CustomerID`. Notice that the child—orders—data are deleted first, changes are submitted, and then the parent customer data are deleted, and changes are submitted. Notice that deletes in this example use `DeleteAllOnSubmit` and all of the code is wrapped up in a `TransactionScope` block (see Listing 16.9).

LISTING 16.9 Manually Deleting Parent and Child Rows as Part of a Single Transaction

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data.Linq;
using System.Data.Linq.Mapping;
```

LISTING 16.9 Continued

```
using System.Transactions;

namespace LinqToSQLUsingTransactionScope
{
    class Program
    {
        static void Main(string[] args)
        {
            Northwind northwind = new Northwind();
            Table<Customer> customers = northwind.GetTable<Customer>();
            Table<Order> orders = northwind.GetTable<Order>();

            using (TransactionScope scope = new TransactionScope())
            {
                var removeCustomers =
                    from customer in customers
                    where customer.CustomerID.ToString().ToUpper() == "PAUL1"
                    select customer;

                var removeOrders = from order in orders
                                   where order.CustomerID.ToString().ToUpper() == "PAUL1"
                                   select order;

                if (removeOrders.Any())
                {
                    orders.DeleteAllOnSubmit(removeOrders);
                    northwind.SubmitChanges();
                }

                customers.DeleteAllOnSubmit(removeCustomers);
                northwind.SubmitChanges();
                scope.Complete();
            }
        }

        public class Northwind : DataContext
        {
            private static readonly string connectionString =
                "Data Source=.\SQLExpress;AttachDbFilename="" +
                "C:\Books\Sams\LINQ\Northwind\northwnd.mdf"" +
                ";Integrated Security=True;Connect Timeout=30;User Instance=True";

            public Northwind()
                : base(connectionString)
            {

```

LISTING 16.9 Continued

```
    Log = Console.Out;
}
}

[Table(Name = "Customers")]
public class Customer
{
    [Column(IsPrimaryKey = true)]
    public string CustomerID { get; set; }

}

[Table(Name = "Orders")]
public class Order
{
    [Column(IsPrimaryKey = true)]
    public int OrderID { get; set; }
    [Column()]
    public string CustomerID { get; set; }
}

}
```

Understanding Conflict Resolution

Conflicts occur. In families, conflicts occur over laundry and money. In software, database conflicts occur when more than one agent changes data at a time. An agent can be people or subprograms.

Some terms you should be familiar with include concurrency, concurrency conflict, concurrency control, optimistic concurrency, pessimistic concurrency, and conflict resolution. Concurrency refers to simultaneous updates. Concurrency conflict refers to simultaneous changes that are in conflict. Concurrency control refers to the technique used to resolve concurrency. Optimistic concurrency is a consideration for the likelihood of conflicting, concurrent changes. Pessimistic concurrency uses lock to prohibit conflicting, concurrent changes. And, conflict resolution is the process of resolving conflicts.

Pessimistic record locking clearly solves the problem because it blocks concurrency. The problem with pessimistic concurrency is that it doesn't scale very well. Record locks are expensive, connections—a finite resource—are held on to, and in the Internet world of potentially massive numbers of users, locking records just doesn't scale very well.

The alternative is that you then have to use optimistic concurrency, which is to permit simultaneous changes and then resolve conflicts when or if they occur. LINQ supports optimistic concurrency.

The way that LINQ supports concurrency is through the exception mechanism. LINQ supports mapping which columns are checked for concurrent updates via the `UpdateCheck` parameter of the `ColumnAttribute`. When two agents have changed the same data in conflicting ways, LINQ throws an exception and then it's the programmer's job to figure out how to resolve the conflicting data. A usual means of resolving conflicts is to show both versions to a user and let the user decide which values to keep.

In the following subsections, you get a chance to explore all of the ways in which LINQ supports concurrency conflict and resolution.

Indicating the Conflict Handling Mode for `SubmitChanges`

The `SubmitChanges` method accepts a `ConflictMode` argument. `ConflictMode` is an enum that can be `FailOnFirstConflict` or `ContinueOnConflict`. If the argument is `FailOnFirstConflict`, the update stops as soon as two values are in conflict. If the value is `ContinueOnConflict`, all of the conflicts are accumulated and the update stops after all conflicts are returned at the end of the process.

Regardless of the `ConflictMode` enumeration chosen, a concurrency conflict throws a `ChangeConflictException`. Listing 16.10 shows a sample console application that uses the `BackgroundWorker` component to simulate simultaneous updates and causes a `ChangeConflictException`. In the next section, you get to doctor-up the code to resolve the conflict.

LISTING 16.10 A Concurrency Conflict Occurs when Two Users Make Conflicting Changes to the Same Record as Demonstrated Here via a `BackgroundWorker` and Two Threads Making Conflicting Changes to the Shippers Table

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data.Linq;
using System.Data.Linq.Mapping;
using System.ComponentModel;

namespace LINQToSQLOConflictHandlingMode
{
    class Program
    {
        static void Main(string[] args)
        {
            BackgroundWorker worker1 = new BackgroundWorker();
            worker1.DoWork += new DoWorkEventHandler(worker1_DoWork);
            BackgroundWorker worker2 = new BackgroundWorker();
            worker2.DoWork += new DoWorkEventHandler(worker1_DoWork);
            worker1.RunWorkerAsync("Speedy Distressed");
        }

        static void worker1_DoWork(object sender, DoWorkEventArgs e)
        {
            string name = e.Argument as string;
            using (NorthwindDataContext db = new NorthwindDataContext())
            {
                var query = from s in db.Shippers
                           where s.CompanyName == "Speedy
                           Distressed"
                           select s;
                if (query.Count() > 0)
                {
                    Shipper s1 = query.First();
                    s1.Fax = "123-4567";
                    db.SubmitChanges();
                }
            }
        }
    }
}
```

LISTING 16.10 Continued

```
    worker2.RunWorkerAsync("Speedy Xpress");
    Console.ReadLine();
}

static void worker1_DoWork(object sender, DoWorkEventArgs e)
{
    Northwind northwind = new Northwind();
    var one = northwind.Shippers.Single(r => r.ShipperID == 1);

    // Speedy Express
    one.CompanyName = (string)e.Argument;

    System.Threading.Thread.Sleep(2500);
    northwind.SubmitChanges(ConflictMode.FailOnFirstConflict);
}
}

public class Northwind : DataContext
{
    private static readonly string connectionString =
        "Data Source=.\SQLExpress;AttachDbFilename=" +
        "c:\temp\northwnd.mdf;Integrated Security=True;" +
        "Connect Timeout=30;User Instance=True";

    public Northwind() : base(connectionString)
    {
        Log = Console.Out;
    }

    public Table<Shipper> Shippers
    {
        get { return this.GetTable<Shipper>(); }
    }
}

[Table(Name = "Shippers")]
public class Shipper
{
    [Column(IsPrimaryKey=true)]
    public int ShipperID { get; set; }
    [Column()]
    public string CompanyName { get; set; }
    [Column()]
}
```

LISTING 16.10 Continued

```
    public string Phone { get; set; }  
}  
}
```

Catching and Resolving Concurrency Conflicts

Single user applications or applications that will only rarely have multiple users updating the same record can use pessimistic concurrency. Simply think *last person to update the record has the right information*. If multiple users will be updating records, you will need to handle conflicts by catching the `ChangeConflictException`. Again, it might be sufficient to permit some columns to be updated without a conflict check.

Revisions to the code in Listing 16.10 will start with code for ignoring entity fields/database columns during concurrency checks and progress to examining conflicts and resolving those conflicts. (Going forward, all of the remaining code listings in the following subsections are revisions to Listing 16.10.)

Ignoring Some Columns for Conflict Checks

The `ColumnAttribute` supports a named `UpdateCheck`. If you add `UpdateCheck`, LINQ uses that value to determine how to treat a column during conflict checks. The values for `UpdateCheck` are `Always`, `Never`, and `WhenChanged`.

Listing 16.11 shows a modified `Shipper` class that applies the `UpdateCheck` to `CompanyName` and `Phone`. The `CompanyName` is checked for conflicts when it is modified, but the `Phone` (number) is never checked. These values should be set to whatever makes sense for your solution.

LISTING 16.11 An Excerpt from Listing 16.10 That Applies the `UpdateCheck` Named Argument to the `ColumnAttribute` to Guide How LINQ Performs Concurrency Conflict Checks

```
[Table(Name = "Shippers")]  
public class Shipper  
{  
    [Column(IsPrimaryKey=true)]  
    public int ShipperID { get; set; }  
  
    [Column(UpdateCheck=UpdateCheck.WhenChanged)]  
    public string CompanyName { get; set; }  
  
    [Column(UpdateCheck=UpdateCheck.Never)]  
    public string Phone { get; set; }  
}
```

Retrieving Conflict Information

To catch change conflicts, add a try...catch block with `ChangeConflictException` in the catch clause. Listing 16.12 is an excerpt from the worker method showing how to apply the exception handler. In the example—until the next section—the exception handler simply writes the exception message to the console.

LISTING 16.12 Adding the `ChangeConflictException` Handler Around the `SubmitChanges` Call

```
static void worker1_DoWork(object sender, DoWorkEventArgs e)
{
    Northwind northwind = new Northwind();
    var one = northwind.Shippers.Single(r => r.ShipperID == 1);

    // Speedy Express
    one.CompanyName = (string)e.Argument;

    System.Threading.Thread.Sleep(2500);
    try
    {
        northwind.SubmitChanges(ConflictMode.FailOnFirstConflict);
    }
    catch(ChangeConflictException ex)
    {
        Console.WriteLine(ex.Message);
    }
    Console.ReadLine();
}
```

Determining the Entity and Table Associated with the Conflict

Two primary things are associated with the conflict: the table and the entity object. This information is derived from the `DataContext.ChangeConflicts` collection and the `DataContext.Mapping` collection.

Listing 16.13 extends Listing 16.10 to demonstrate how to iterate through the `ChangeConflicts` collection (in bold in the listing), which yields each object related to the conflict, a `ChangeObjectConflict`. And, Listing 16.13 demonstrates how to use the `DataContext.Mapping` property, an instance of the `MetaModel` class, to determine the `MetaTable`. The `ChangeObjectConflict` represents an update attempt with one or more conflicts, and the `MetaTable` is an abstraction of the underlying table or view.

LISTING 16.13 Examining the Entities in Conflict and the MetaTable, Which Is an Abstraction of the Underlying Table or View in Conflict

```

static void worker1_DoWork(object sender, DoWorkEventArgs e)
{
    Northwind northwind = new Northwind();
    var one = northwind.Shippers.Single(r => r.ShipperID == 1);

    // Speedy Express
    one.CompanyName = (string)e.Argument;

    System.Threading.Thread.Sleep(2500);
    try
    {
        northwind.SubmitChanges(ConflictMode.FailOnFirstConflict);
    }
    catch(ChangeConflictException ex)
    {
        Console.WriteLine(ex.Message);
        foreach(ObjectChangeConflict conflict in northwind.ChangeConflicts)
        {
            // entity in conflict
            MetaTable meta = northwind.Mapping.GetTable(conflict.Object.GetType());
            Console.WriteLine("Table: {0}", meta.TableName);
            Console.WriteLine("Object:\n {0}", conflict.Object);
        }
    }
    Console.ReadLine();
}

```

16

The `foreach` statement—added to Listing 16.12—iterates over each conflict and shows the `MetaTable` and entity related to the conflict. To resolve detailed elements of the conflict, you need to drill down into member elements of the entities.

Comparing Various States of Member Conflicts

A common way to show a user member conflict is to display the various values, perhaps in a grid. In the simplified solution in Listing 16.14, the three states are simply written to the console. The three member states are what is in the database, the original value, and the entity's current value.

Each member element—property or column—in conflict can be accessed in the `ObjectChangeConflict.MemberConflicts` collection. Each of the `MemberChangeConflict` objects contains a `CurrentValue`, `OriginalValue`, and `DatabaseValue` property defined as an object type. These properties permit examining or displaying the various possible values in the conflict. Listing 16.14 adds a nested `foreach` statement that walks through all of the member conflicts and displays them to the console.

LISTING 16.14 For Each Entity in Conflict, There Will Be One or More Member Values in Conflict; the Nested foreach Loop Demonstrates How to Access and Display Each of the Possible Values

```
static void worker1_DoWork(object sender, DoWorkEventArgs e)
{
    Northwind northwind = new Northwind();
    var one = northwind.Shippers.Single(r => r.ShipperID == 1);

    // Speedy Express
    one.CompanyName = (string)e.Argument;

    System.Threading.Thread.Sleep(2500);
    try
    {
        northwind.SubmitChanges(ConflictMode.FailOnFirstConflict);
    }
    catch(ChangeConflictException ex)
    {
        Console.WriteLine(ex.Message);
        foreach(ObjectChangeConflict conflict in northwind.ChangeConflicts)
        {
            // entity in conflict
            MetaTable meta = northwind.Mapping.GetTable(conflict.Object.GetType());
            Console.WriteLine("Table: {0}", meta.TableName);
            Console.WriteLine("Object:\n {0}", conflict.Object);

            // member in conflict
            foreach(MemberChangeConflict member in conflict.MemberConflicts)
            {
                var current = member.CurrentValue;
                var original = member.OriginalValue;
                var database = member.DatabaseValue;

                Console.WriteLine("Current:\n{0}", current);
                Console.WriteLine("Original:\n{0}", original);
                Console.WriteLine("Database:\n{0}", database);
            }
        }
    }
    Console.ReadLine();
}
```

Writing Resolved Conflicts to the Database

The remaining aspect of conflict resolution is figuring out what value to keep and resolving the correct value among the elements involved, generally the entity and the table. The `ObjectChangeConflict.Resolve` method performs this task for you.

The `System.Data.Linq.RefreshMode` enumeration supports three choices:

`OverwriteCurrentValues`, `KeepCurrentValues`, and `KeepChanges`. `OverwriteCurrentValues` overwrites an object's state with database values. `KeepCurrentValues` overwrites database values with the mapped object's state, and `KeepChanges` merges database and object values. Listing 16.15 extends Listing 16.14 (and Listing 16.10) and adds an opportunity for a user to indicate whether to overwrite the object values, overwrite the database values, or merge the changes. (With a little ingenuity, a user interface could be added and a conflict resolution screen constructed.)

LISTING 16.15 Use `ObjectChangeConflict.Resolve` to Indicate What to Do with Database Conflicts

```
static void worker1_DoWork(object sender, DoWorkEventArgs e)
{
    Northwind northwind = new Northwind();
    var one = northwind.Shippers.Single(r => r.ShipperID == 1);

    // Speedy Express
    one.CompanyName = (string)e.Argument;

    System.Threading.Thread.Sleep(2500);
    try
    {
        northwind.SubmitChanges(ConflictMode.FailOnFirstConflict);
    }
    catch(ChangeConflictException ex)
    {
        Console.WriteLine(ex.Message);
        foreach(ObjectChangeConflict conflict in northwind.ChangeConflicts)
        {
            // entity in conflict
            MetaTable meta = northwind.Mapping.GetTable(conflict.Object.GetType());
            Console.WriteLine("Table: {0}", meta.TableName);
            Console.WriteLine("Object:\n {0}", conflict.Object);
            // member in conflict
            foreach(MemberChangeConflict member in conflict.MemberConflicts)
            {
                var current = member.CurrentValue;
                var original = member.OriginalValue;
                var database = member.DatabaseValue;
```

LISTING 16.15 Continued

```
Console.WriteLine("Current:\n{0}", current);
Console.WriteLine("Original:\n{0}", original);
Console.WriteLine("Database:\n{0}", database);

Console.WriteLine("1=Use database values, " +
    "2=Use object values, 3=Merge values");

string input = Console.ReadLine();
int value = Convert.ToInt32(input);
switch(value)
{
    case 1:
        conflict.Resolve(RefreshMode.KeepCurrentValues);
        break;
    case 2:
        conflict.Resolve(RefreshMode.OverwriteCurrentValues);
        break;
    case 3:
        conflict.Resolve(RefreshMode.KeepChanges);
        break;
    default:
        throw;
}
}

northwind.SubmitChanges(ConflictMode.FailOnFirstConflict);
}
```

Notice that in the solution the `SubmitChanges` method is called a second time after all of the conflicts are resolved.

N-Tier Applications and LINQ to SQL

On the balance, LINQ to SQL is weighted toward client-side use. You can use LINQ to SQL in n-tier applications, but there are a couple of challenges to overcome. You have to deal with serialization and concurrency. Mapped entities can be serialized, but then they are detached from their `DataContext`. When entities are detached, change tracking is lost. Without change tracking, optimistic concurrency is not supported. To overcome this challenge, you need to keep track of the original object and the new fields (or a new object) and pass both back to the service. `DataContext.Attach` reassociates the returned information with a `DataContext` and applies the changes.

Use ADO.NET to Fill Objects for Maximum Flexibility

I tend to write applications using ADO.NET to read and write data and then the data is placed in custom classes rather than using DataSets or directly mapped entities.

Building applications like this easily permits using LINQ anywhere in the architecture because now we are just talking about classes and collections. Leveraging tools like CodeRush, snippets, and patterns, writing custom classes and using ADO.NET just to move data to and from the database permits maximal flexibility and does not take significantly longer than using DataSets and DataTables.

Because we are talking about an n-tier application, you will need a middle tier. In the spirit of cool, new technology, the listing uses WCF—Windows Communication Foundation—introduced in .NET 3.0. If you need a primer on WCF, check out Windows Communication Foundation Unleashed by Craig McMurtry, et al from Sams. Basically, WCF homogenizes services development through a common set of attributes and the WCF Service Library template from the New Project dialog box will do enough work for our purposes.

The elements to our solution are as follows:

- ▶ An n-tier solution using WCF and LINQ to SQL
- ▶ A service contract method—the terminology used in WCF land—that returns some data from a LINQ to SQL mapped entity
- ▶ A second method that accepts the original entity and a new one with the changes
- ▶ A basic client that gets a customer from the service, clones the customer, changes one copy, and sends both the original and new value back to the WCF service

Listing 16.16 contains the interface that represents the service contract. The interface is literally adorned with the `ServiceContractAttribute`. Each method that represents part of the contract is adorned with the `OperationContractAttribute`. So that the `Customer` can be serialized and used by the client application, in addition to the `TableAttribute` and `ColumnAttribute` necessary to map the `Customer` entity to the `Customers` table, you need to adorn the `Customer` with the `DataContractAttribute` and each property with the `DataMemberAttribute`. It is these four additional attributes—`ServiceContractAttribute`, `OperationContractAttribute`, `DataContractAttribute`, and `DataMemberAttribute`—that comprise the elements of the service contract. Listing 16.16 contains all of this code.

LISTING 16.16 The Service Contract and the LINQ to SQL Code with the New Attributes for Serializing Customer Objects

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Runtime.Serialization;
using System.ServiceModel;
```

LISTING 16.16 Continued

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

namespace WcfService
{
    // NOTE: If you change the interface name "IService1" here,
    // you must also update the reference to "IService1" in App.config.
    [ServiceContract]
    public interface IService1
    {
        [OperationContract]
        List<Customer> GetCustomers();
        [OperationContract]
        void UpdateCustomer(Customer original, Customer modified);
    }

    public class Northwind : DataContext
    {
        private static readonly string connectionString =
            "Data Source=BUTLER;Initial Catalog=Northwind;Integrated Security=True";

        public Northwind(): base(connectionString){}
    }

    [DataContract]
    [Table(Name="Customers")]
    public class Customer
    {
        [DataMember]
        [Column(IsPrimaryKey=true)]
        public string CustomerID{ get; set; }

        [DataMember]
        [Column]
        public string CompanyName{ get; set; }

        [DataMember]
        [Column]
        public string ContactName{ get; set; }

        [DataMember]
        [Column]
        public string ContactTitle{ get; set; }
    }
}
```

LISTING 16.16 Continued

```
[DataMember]
public string Address{ get; set; }

[DataMember]
[Column]
public string City{ get; set; }

[DataMember]
[Column]
public string Region{ get; set; }

[DataMember]
[Column]
public string PostalCode{ get; set; }

[DataMember]
[Column]
public string Country{ get; set; }

[DataMember]
[Column]
public string Phone{ get; set; }

[DataMember]
[Column]
public string Fax{ get; set; }

}
```

Listing 16.17 implements the service contract described by the interface in Listing 16.16. Notice that the `UpdateCustomer` method accepts two `Customer` objects. One needs to be the original, unchanged customer and the second needs to contain the changes. The changes are reconciled in the service layer by calling the `Attach` method with both customer objects.

LISTING 16.17 The Service1 Class Implements the Service Contract and Contains the Code—Using the Attach Method—to Update the Actual Customer Object

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Runtime.Serialization;
using System.ServiceModel;
using System.Text;
```

LISTING 16.17 Continued

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

namespace WcfService
{
    public class Service1 : IService1
    {
        public List<Customer> GetCustomers()
        {
            Northwind northwind = new Northwind();
            Table<Customer> customers = northwind.GetTable<Customer>();
            return customers.ToList();
        }

        public void UpdateCustomer(Customer original, Customer modified)
        {
            Northwind northwind = new Northwind();
            Table<Customer> customers = northwind.GetTable<Customer>();
            customers.Attach(modified, original);
            northwind.SubmitChanges();
        }
    }
}
```

Finally, define a client that contains a reference to the service. Call `GetCustomers` from the service. Using the deep copy `Clone` method make a copy of the `Customer` object, modify the original, and call the service to update the customer. The client code that uses the service is shown in Listing 16.18.

LISTING 16.18 A Simple Client That Demonstrates a Way to Keep a Copy of a Mapped Entity to Use when Updating the Changes Behind the Service with LINQ to SQL

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data.Linq;
using System.Data.Linq.Mapping;
using LinqToSQLDistributed.ServiceReference1;
using System.Reflection;

namespace LinqToSQLDistributed
{
```

LISTING 16.18 Continued

```
class Program
{
    static void Main(string[] args)
    {
        IService1 serv = new Service1Client();
        Customer[] customers = serv.GetCustomers();

        var alfki = (from customer in customers
                     where customer.CustomerID == "ALFKI"
                     select customer).Single();

        //030-0076545
        Customer original = Clone(alfki);
        alfki.Fax = "030-0076541";
        serv.UpdateCustomer(original, alfki);

    }

    public static T Clone<T>(T toClone) where T : new()
    {
        T target = new T();

        PropertyInfo[] properties = toClone.GetType().GetProperties();
        foreach(PropertyInfo property in properties)
        {
            PropertyInfo targetProperty = target.GetType()
                .GetProperty(property.Name);

            targetProperty.SetValue(target,
                property.GetValue(toClone, null), null);
        }

        return target;
    }
}
```

Summary

You can assemble software in a lot of different ways. The most important key to success is that you work on the pieces of software that the customer wants. Having a lot of ways to build those pieces is not about one right way—it's about the way that makes sense to you and your project and sometimes the people (and sometimes preexisting code) that you have to work with.

If you are comfortable and have been successful at building applications using DataSets and DataTables, then LINQ to DataSets lets you keep doing that. If you like the ease of assembling entities with SqlMetal or the LINQ to Classes designer, then use LINQ to SQL. LINQ to DataSets supports traditional client/server programming very well. LINQ to SQL is a step closer to using custom classes, and of course, LINQ can be used with completely custom classes.

In this chapter, you learned about adding, updating, and deleting data with LINQ for data. You learned about using transactions, stored procedures, and SQL functions. You also learned about the LINQ to SQL designer and SqlMetal, and you learned about optimistic concurrency and using LINQ in n-tier applications. All of the elements you learned about in the prior chapters can be used in combination with the material in this chapter.

CHAPTER 17

Introducing ADO.NET 3.0 and the Entity Framework

"There are only two ways to live your life. One is as though nothing is a miracle. The other is as though everything is a miracle."

—Albert Einstein

Everything in this book is based on released technology except this chapter and the final chapter based on LINQ to XSD. LINQ to XSD is closer to being released and the Entity Framework is supposed to ship with Microsoft Visual Studio SP1, sometime in 2008.

The sheer volume, the sheer magnitude of the output from 100,000+ very bright people (at Microsoft) is staggering. The ADO.NET Entity Framework encompasses LINQ, Generics, Entity Data Modeling, something new called "Entity SQL," new classes, and a whole new way of interacting with data providers. The ADO.NET Entity Framework really needs its own book. (If the gods are willing, perhaps I will get to provide you with one. In all likelihood, there will be whole books on just Entity SQL.)

The ADO.NET Entity Framework is designed to lower the speed bump that is current database programming—getting data from flat, relational tables into *bumpy* objects. Of course, to be able to use the framework, you have the new speed bump of learning all of the related capabilities.

When I asked Brian Dawson, Program Manager for the ADO.NET Entity Framework *at Microsoft*, Brian assured me that ADO.NET 3.0 is not an all-or-nothing proposition. Brian's response was that programmers will still be able to write database interactions as they do now with ADO.NET 2.0 or take an alternate path and a different approach and

IN THIS CHAPTER

- ▶ Understanding the General Nature of the Problem and the Solution
- ▶ Finding Additional Resources
- ▶ Building a Sample Application Using Vanilla ADO.NET Programming
- ▶ Programming with the Entity Framework
- ▶ Doing It All with LINQ

use the Entity Framework. (I suspect that they [Microsoft] think that once you get a taste of the new ADO.NET, the old way of doing things will seem less appealing.)

What you will accomplish here is a look at one way of using ADO.NET 2.0 style programming in contrast with the new way, using the ADO.NET Entity Framework. You will also see a brief example of Entity SQL and get a refresher on creating databases with good old-fashioned Structured Query Language (SQL) scripting.

Before you begin, it is crucial that you understand that although there is a lot of information and new technology here, a tremendous amount of the plumbing is handled for you by wizards and Visual Studio designers. You can get an Entity Framework example up and running by selecting one file template that runs a wizard and by writing a simple LINQ query. (Don't let that simplicity get lost in the technology-noise.)

Understanding the General Nature of the Problem and the Solution

The general “database” problem is that based on the current relational technology everyone seems to be encouraged to write their own data access layer. There is no standard for getting data into objects, so everyone writes a bunch of plumbing code to do just that.

When almost everyone needs to do the same thing, it begs the question, “What is the general solution that everyone can use?” Ten years ago, a few companies threw object-oriented databases against the wall. They didn’t stick. So, in the ensuing years, in almost every project—at least of the dozens I have seen—a data access layer is reinvented from scratch.

The challenge is that a huge amount of data is stored in relational databases already. This state of affairs suggests that a solution needs to work existing relational technology, but the solution also needs to bridge the gap between tables, columns, and rows and complex object shapes.

Relational database technology has been around a while and so has object-oriented programming, yet this problem has persisted. I suspect Microsoft believes they are on track for a solution, but the final arbiters are you and I, which depends on the suitability of the solution in the real world.

Understanding Problems with the Relational Database Model as It Pertains to C# Programmers

In the practical world, there is a physical database model that satisfies the needs of the database engine and the file system. There is a logical model that satisfies the needs of relational technology, and then there is a conceptual model that represents the relationships that need to be expressed for the problem space. Programmers generally want to get the data into its conceptual form—customers and orders, stocks and historical prices—as painlessly as possible.

With ADO.NET 2.0 and earlier technologies, programmers had to understand a little bit about the physical model: how database files were stored and how to attach them to some

context. Programmers really had to know the logical model to write queries to get the logical model into a conceptual form. Going from logical to conceptual usually entails writing complex queries that use all manner of joins, and for C# programmers, you had to understand ADO.NET and ADO+ before that.

In this (the present) style, there is no insulation between the logical model and the conceptual model. If anyone needs to change the schema, then everything from the joins up needs to change. That is, the stored procedures change, the ADO.NET plumbing changes, and, often, the classes in C# have to change. Consequently, unless you have a dedicated Database Administrator (DBA) who has available time, C# programmers also have to master SQL—and left joins, right joins, cross joins, inner joins, outer joins, and self joins.

In reality, mastering C# and the .NET Framework is a large enough task for anyone. Becoming a SQL expert, too, is likely too much to ask. Being an expert in C# and SQL at a minimum is not going to promote optimal productivity.

Understanding How the Entity Framework Is Designed to Help

A solution in the pipeline is the ADO.NET Entity Framework. The Entity Framework represents a capability that facilitates quickly and consistently assembling a conceptual data model on top of all of the SQL plumbing. The idea is that the DBA focuses on the physical and logical model and joins and SQL, and the C# programmer simply focuses on programming against the conceptual model.

NOTE

Because the relationships and the incumbent joins are buried beneath the conceptual data model, indexes, keys, and relationships don't play as big a role for the C# developer using the Entity Framework.

17

In simplistic terms, the ADO.NET Entity Framework lets the C# programming focus on shaped objects like cars and parts, customers and orders, or stocks and historical prices without having to track indexes, keys, and joins clauses.

Grokkking the Nature of the Solution

The Entity Framework uses a wizard to code generate wrapper classes that add a level of abstraction between the logical data model, creating a conceptual data model for you. (It's worth noting that the logical model guides the conceptual model and the developer can influence the way the conceptual model is captured in code.)

The wrapper classes are shaped by the underlying logical model. For example, in Northwind, Customers have Orders and Orders have Order Details. The ADO.NET Entity Framework plumbing uses this information and all of the elements of relational programming, like keys and indexes, to figure out how to generate the conceptual model. In the

final result, Customers will still have Orders, but you won't have to think of these in terms of primary and foreign keys and joins. Instead, you will simply think "Customers have Orders." The mechanics of the relational technology is elevated to the object-oriented technology for you.

Conceptual Entity Data Models (EDMs) are stored in eXtensible Markup Language (XML) as a file with an .edmx extension. You have the option of interacting with the EDM using Entity SQL (aka eSQL), a new SQL-like language invented for the Entity Framework. You also have the option of interacting with the EDM using LINQ to Entities. Entity SQL and LINQ to Entities are two different things. (You can read more about Entity SQL in the section "Querying the Entity Data Model with Entity SQL.") Entity SQL is like SQL without joins. You use a notation like `Customers.Orders` instead of a join on Customers and Orders, and you are querying against the EDM instead of the database directly. There is a translation that happens from Entity SQL to regular SQL. LINQ to Entities is LINQ against the entities defined by the EDM. In the hierarchy of technologies working out to in, LINQ to Entities is the high-level technology, Entity SQL is an alternate technology branch for the Entity Framework, and beneath each of these is SQL. You have the option of interacting with your data store at any of these levels.

The basic idea is that in any project in Visual Studio 2008, you can add an ADO.NET Entity Data Model item from the Add New Item dialog box. A wizard lets you select the tables, views, and procedures to include and a conceptual wrapper for those selected items is created by Visual Studio and integrated add-ins. The conceptual model is based on the relationships in the selected database elements. Finally, you can use a Visual Studio designer and influence and add more items to the EDM.

When the wizard has finished, you will program against the EDM with LINQ to Entities in the general case and Entity SQL, perhaps, in specialized cases.

In addition, tools that integrate in Visual Studio can be run from a command line. You could also handcraft the EDM XML and wrapper classes, but all of your productivity gains would be lost in the effort. It is better to use the integrated tools when possible.

Last, but not least, it is worth reiterating that you always have the option of programming database interactions the old way. The problem with the way we, as a group, have programmed ADO.NET is that everyone—each team—basically has invented their own data access layer. This roll-your-own approach is time consuming, expensive, and rife with unmitigated challenges. But you do get to choose.

Finding Additional Resources

Sometimes a slide deck—PowerPoint slides—includes the "Resources" slide at the end of the slide deck. Even though the examples for this chapter follow, a resources section is included here because you need to download the separate Entity Framework and tools, and you might want to download some additional samples. A couple of prevailing blogs and a Wikipedia reference are included here to help you fill in any gaps that might have been missed.

Wikipedia

www.wikipedia.org is a great resource. A free, multilingual, open content encyclopedia is based on the concept that interested parties and the reading audience will ensure a balanced editorial process. You might think that Wikipedia—because it allows anyone regardless of their credentials or biases to add content—is subject to misinformation; however, it can also be a fount of great information. At a minimum, it's a good starting point for finding out what something is. Think of Wikipedia as, at the least, pretty good *Cliff Notes*.

You can get a pretty good overview of the Entity Framework on Wikipedia at http://en.wikipedia.org/wiki/ADO.NET_Entity_Framework#cite_note-0. This entry includes several additional, related links.

Entity SQL Blog

Microsoft is a big place with a lot of people. By all accounts, it looks like Entity SQL was implemented independently of LINQ to Entities. Whether LINQ wasn't ready or Entity SQL was someone's pet project is unclear. And, it's probably not that important for our purposes. You have two options for the Entity Framework: LINQ to Entities or Entity SQL.

A general concept behind LINQ is that you can use one query language for objects, SQL, XML, and Entities. If you choose to use Entity SQL, you are back to having to learn a new language—in addition to C#, SQL, and LINQ. Entity SQL is something different altogether. However, if you want to know more about Entity SQL, check out the Entity SQL blog at <http://blogs.msdn.com/adonet/archive/2007/05/30/entitysql.aspx>.

Downloading and Installing the Entity Framework

To try the Entity Framework, you need to do three things. You need to download and install the ADO.NET Entity Framework Beta 3. You need to install a Visual Studio patch, and you need to download and install the ADO.NET Entity Framework Tools CTP 2 (refer to Figure 17.1).

CAUTION

This software is still beta software. To be safe, the best strategy is to not install on a production workstation. Other possible options include using a virtual machine or completely backing up your workstation before installing beta software, especially if you are installing beta ware on a production box.

You can, of course, Google for the ADO.NET Entity Framework, or you can go right to <http://www.codeplex.com/adonetsamples/>, which is a Microsoft site. Microsoft's downloads section also has links. Download and install the ADO.NET Entity Framework Beta 3 from <http://www.microsoft.com/downloads/details.aspx?FamilyId=15DB9989-1621-444D-9B18-D1A04A21B519&displaylang=en>. Then, you need to download and install a patch from <http://go.microsoft.com/fwlink/?LinkId=104985>. (Don't worry: If you forget this step, the Entity Framework Tools install will remind you to install the patch.) Finally, you

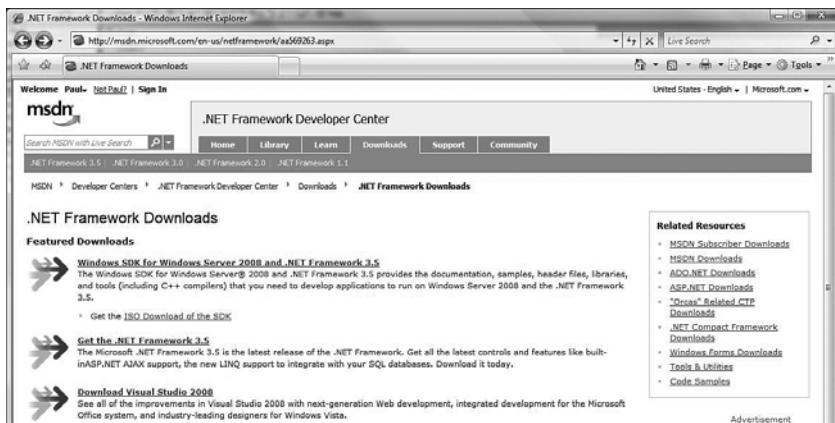


FIGURE 17.1 A download page for the ADO.NET Entity Framework and framework tools.

can download and install the ADO.NET Entity Framework Tools CTP 2 (for Beta 3) here: <http://www.microsoft.com/downloads/details.aspx?FamilyId=D8AE4404-8E05-41FC-94C8-C73D9E238F82&displaylang=en>.

These downloads are just a few megabytes, so the download and installation process goes very quickly.

Downloading Samples

When you download the Entity Framework Beta 3, a shortcut is placed in the installation folder. This shortcut takes you to the download page for some sample applications. You can also go right to <http://www.codeplex.com/adonetsamples/> to get the samples from codeplex.

Go Live Estimate

Quoting from an MSDN blog—<http://blogs.msdn.com/adonet/archive/2008/04/09/entity-framework-ado-net-data-services-to-ship-with-vs-2008-sp1-net-3-5-sp1.aspx>—Elisa Flisko of Microsoft says that the ADO.NET Entity Framework will ship with Visual Studio 2008 SP1. (On May 13, 2008, a notice was posted for the Visual Studio SP1 beta. So, by the time you have this book in your hands, you should be able to get the SP1 beta, if not the RTM for SP1).

All of these resources should get you started. And, as I write this I have begun discussions for a *Teach Yourself ADO.NET Entity Framework Programming* book, as I suspect a few other authors have. Look for books on the Entity Framework probably in the last half of 2008. (SP1 beta was available as of June 2008.)

Building a Sample Application Using Vanilla ADO.NET Programming

A data access layer based on ADO.NET 2.0 and a good object model is a modality that works great. I have it on good authority that ADO.NET is not a lock-in or lock-out strategy. You will be able to continue to use ADO.NET 2.0 going forward or you will be able to use ADO.NET 3.0 and the Entity Framework going forward. Using one doesn't exclude the other.

Generally, an upgrade involves an integration of new capabilities with the old. For example, ADO.NET 2.0 introduced the TransactionScope but many of the other elements such as command, reader, adapter, and DataSet remained. With ADO.NET 3.0, the transition, if you choose it, encompasses a completely different way of programming against the database. How long parallel support for database programming styles will remain is anybody's guess, but the answer I got was for the "foreseeable future."

In this part of the chapter, you get a chance to directly compare vanilla ADO.NET 2.0 code against ADO.NET 3.0 (Entity Framework) code, based on beta 3.

Sample Applications and Production Applications Vary Stylistically

In the example, the quote is requested from Yahoo! and the console application invokes an update to the database immediately. This is good enough for sample applications. In a production application, I would include entity classes for Customer and Quote with Customer classes containing a collection of quotes. I would probably also use a factory method that could construct the Customer and Quote parts from the returned Yahoo! string, encapsulating all of the string chunking. Finally, a class for data access would provide the UpdateHistory capability. For comparing the differences in the ADO.NET versions, it is sufficient to do without these additional elements.

17

Defining a Database for Storing Stock Quotes

To demonstrate, you need a database. You could use the Northwind database or the newer AdventureWorks sample database that shipped with SQL Server 2005. Some of the samples from codeplex.com use the AdventureWorks database, so a new one was created here for you.

For the remaining samples in this chapter, create a new database using Visual Studio's Server Explorer or SQL Server Management Studio to create a database named StockHistory. To create the tables in Listings 17.1 and 17.2 in Visual Studio, follow these steps:

1. With Visual Studio running, open Server Explorer.
2. Right-click Data Connections and select Create New SQL Server Database.
3. Enter **.\\SQLEXPRESS** for the server name, **StockHistory** for the database name, and then click OK.
4. Expand the new database node, and select the Tables node (in Server Explorer).

5. Click Add Table.
6. In the Table designer, add the column names and data types for Listing 17.1.
7. Right-click on the CompanyID definition and select Set Primary Key. (For the PriceHistory table, make QuoteID the primary key.)
8. Close the designer, naming the first table **Company**.
9. Repeat the steps for Listing 17.2, and name the second table **PriceHistory**.

LISTING 17.1 The Data to Enter in the Table Designer in Visual Studio for the Company Table

CompanyID	int	Unchecked
CompanyName	nvarchar(30)	Unchecked
CompanySymbol	nvarchar(6)	Unchecked

LISTING 17.2 The Data to Enter in the Table Designer in Visual Studio for the PriceHistory Table

QuoteID	int	Unchecked
CompanyID	int	Unchecked
WhenRequested	datetime	Unchecked
OpeningPrice	decimal(18, 2)	Checked
LastTradePrice	decimal(18, 2)	Checked
LastTradeTime	smalldatetime	Checked
CurrentPrice	decimal(18, 2)	Checked
TodaysHigh	decimal(18, 2)	Checked

If you are still unsure how to define a table in Visual Studio, the complete script is provided in the section “Reference: The Complete Sample Database Script.” You can type the script in Listing 17.4 into a new query in the SQL Server Management Studio and create everything all at once.

Adding a Stored Procedure for Inserting Quotes

Some developers use ad hoc queries in their C# code. Some use stored procedures in SQL. Using stored procedures supports the division of labor, and the SQL Server engine validates the SQL, so the code in Listing 17.3 employs a stored procedure.

Listing 17.3 contains a stored procedure `InsertQuote` that checks to see if the `Company` record exists. If a `Company` record already exists, that `CompanyID` is returned and used; if not, a `Company` record is created. Finally, `InsertQuote` inserts the new stock quote information into the `PriceHistory` table. (In practice, factor out the `INSERT INTO PriceHistory` statement used twice into a separate, stored procedure and use `EXEC` to invoke it at the various locations in Listing 17.3.)

LISTING 17.3 Using Stored Procedures Creates a Nice Division-of-Labor Point and the SQL Engine Verifies the SQL Code

```
ALTER PROCEDURE dbo.InsertQuote
(
    @CompanyName      nvarchar(30),
    @CompanySymbol    nvarchar(6),
    @WhenRequested    datetime,
    @OpeningPrice     decimal(18, 2),
    @LastTradePrice   decimal(18, 2),
    @LastTradeTime    smalldatetime,
    @CurrentPrice     decimal(18, 2),
    @TodaysHigh       decimal(18, 2)
)
AS
BEGIN
    DECLARE @ID int
    SET @ID = -1

    SELECT @ID = CompanyID FROM Company WHERE
        CompanySymbol = @CompanySymbol
    PRINT @ID
    PRINT @OpeningPrice

    IF(@ID = -1)
    BEGIN
        BEGIN TRANSACTION
        BEGIN TRY
            DECLARE @CompanyID INT
            INSERT INTO Company (CompanyName, CompanySymbol)
                VALUES (@CompanyName, @CompanySymbol)
            SET @CompanyID = @@IDENTITY
            INSERT INTO PriceHistory
                (CompanyID, WhenRequested, OpeningPrice,
                 LastTradePrice, LastTradeTime, CurrentPrice, TodaysHigh)
            VALUES
            (
                @CompanyID, @WhenRequested, @OpeningPrice,
                @LastTradePrice, @LastTradeTime, @CurrentPrice,
                @TodaysHigh
            )
        COMMIT TRANSACTION
    END TRY
    BEGIN CATCH
        RAISERROR('Failed to insert quote %s', 16, 1, @CompanyName)
    END CATCH
END
```

LISTING 17.3 Continued

```
ROLLBACK TRANSACTION
END CATCH
END
ELSE
BEGIN
    SET @CompanyID = @ID
    PRINT 'ID > -1'

    INSERT INTO PriceHistory
    (
        CompanyID, WhenRequested, OpeningPrice, LastTradePrice,
        LastTradeTime, CurrentPrice, TodaysHigh
    )
    VALUES
    (
        @CompanyID, @WhenRequested, @OpeningPrice,
        @LastTradePrice, @LastTradeTime, @CurrentPrice,
        @TodaysHigh
    )
END
END
```

NOTE

Insert PRINT statements to aid in debugging stored procedures if needed. In the procedure in Listing 17.3, I actually wrote the WHERE predicate in the first select as CompanySymbol = CompanySymbol without the @ for the right-side parameter because my keyboard has some dead keys from so much typing. Equating the field to itself is syntactically valid but logically useless as it will always return a CompanyID and the first half of the stored procedure will run only on the first insert. It's real easy to miss that <indexterm startref="iddle3661" class="endofrange" significance="normal"/>:]

PRINT statements are an old-school way of debugging. You can also debug stored procedures in Visual Studio 2008 with breakpoints, and you track things closely with the SQL Server profiler.

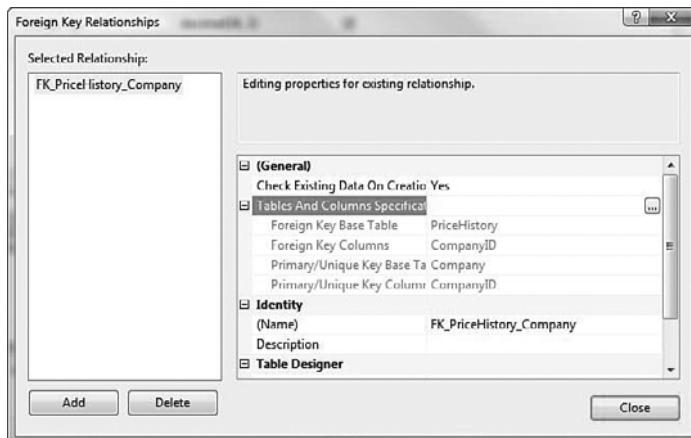
Remember, there is no prohibition against using SQL in your C# code. Consider choosing stored procedures (sprocs, pronounced “*sprox*”) because it is a nice division-of-labor point and because the SQL Server engine will ensure that your SQL is, at least, syntactically correct. Finally, although the performance differences between dynamic SQL and sprocs are diminishing, you probably get at least some minor performance benefits with sprocs versus dynamic SQL.

Adding a Foreign Key

The primary keys are `Company.CompanyID` and `PriceHistory.QuoteID`. The implied relationship is that every company will have a relationship to historical prices, that is, a one-to-many relationship from `Company` to `PriceHistory`. This relationship is implicit in the appearance of the `PriceHistory.CompanyID` column.

To express the relationship, add a foreign key to the database (see Figures 17.2 and 17.3). The steps for defining `CompanyID` as an explicit foreign key are as follows:

1. In Visual Studio, open Server Explorer.



2. Right-click on `PriceHistory` and select Open Table Definition.
3. Right-click on `PriceHistory`'s `CompanyID` and select Relationships.
4. In the selected relationships input, add a foreign key. (Click the Add button.)
5. Click on the Tables and Columns Specification (ellipsis button).
6. For the Primary Key table, select `Company` and pick the `CompanyID`.
7. For the Foreign key table, select `PriceHistory` and pick the `CompanyID`.
8. Click OK.
9. Click Close.

Listing 17.4 provides the SQL script for specifying the foreign key constraint.

LISTING 17.4 SQL Script for Specifying the Foreign Key Constraint

```
ALTER TABLE dbo.PriceHistory ADD CONSTRAINT
FK_PriceHistory_Company FOREIGN KEY
(
    CompanyID
) REFERENCES dbo.Company
(
    CompanyID
) ON UPDATE NO ACTION
ON DELETE NO ACTION
```

Reference: The Complete Sample Database Script

This section is for reference only. Listing 17.5 contains the complete script for generating elements of the StockHistory database. A technique that will help you deploy and manage databases is to create the script for your database. This is a good learning aid, as well as a means of deploying a database dynamically. Scripting also makes it easier to add the database elements, such as stored procedures, to version control supporting tracking a change history.

LISTING 17.5 The Complete Script for the StockHistory Database, Including the Foreign Key Constraint

```
***** Object: Table [dbo].[Company]      Script Date: 05/14/2008 13:48:19 *****/
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
IF NOT EXISTS (SELECT * FROM sys.objects WHERE object_id = OBJECT_ID(N'[dbo].[Company]') AND type in (N'U'))
BEGIN
CREATE TABLE [dbo].[Company](
[CompanyID] [int] IDENTITY(1,1) NOT NULL,
```

LISTING 17.5 Continued

```
[CompanyName] [nvarchar](30) NOT NULL,  
[CompanySymbol] [nvarchar](6) NOT NULL,  
CONSTRAINT [PK_Company_1] PRIMARY KEY CLUSTERED  
(  
    [CompanyID] ASC  
)WITH (IGNORE_DUP_KEY = OFF) ON [PRIMARY]  
) ON [PRIMARY]  
END  
GO  
***** Object: Table [dbo].[PriceHistory]      Script Date: 05/14/2008 13:48:19  
*****/  
SET ANSI_NULLS ON  
GO  
SET QUOTED_IDENTIFIER ON  
GO  
IF NOT EXISTS (SELECT * FROM sys.objects WHERE object_id =  
OBJECT_ID(N'[dbo].[PriceHistory]') AND type in (N'U'))  
BEGIN  
CREATE TABLE [dbo].[PriceHistory](  
    [QuoteID] [int] IDENTITY(1,1) NOT NULL,  
    [CompanyID] [int] NOT NULL,  
    [WhenRequested] [datetime] NOT NULL,  
    [OpeningPrice] [decimal](18, 2) NULL,  
    [LastTradePrice] [decimal](18, 2) NULL,  
    [LastTradeTime] [smalldatetime] NULL,  
    [CurrentPrice] [decimal](18, 2) NULL,  
    [TodaysHigh] [decimal](18, 2) NULL,  
CONSTRAINT [PK_PriceHistory] PRIMARY KEY CLUSTERED  
(  
    [QuoteID] ASC  
)WITH (IGNORE_DUP_KEY = OFF) ON [PRIMARY]  
) ON [PRIMARY]  
END  
GO  
***** Object: StoredProcedure [dbo].[InsertQuote]      Script Date: 05/14/2008  
13:48:19 *****/  
SET ANSI_NULLS ON  
GO  
SET QUOTED_IDENTIFIER ON  
GO  
IF NOT EXISTS (SELECT * FROM sys.objects WHERE object_id =  
OBJECT_ID(N'[dbo].[InsertQuote]') AND type in (N'P', N'PC'))  
BEGIN  
EXEC dbo.sp_executesql @statement = N'CREATE PROCEDURE [dbo].[InsertQuote]  
(
```

LISTING 17.5 Continued

```
@CompanyName nvarchar(30),
@CompanySymbol nvarchar(6),
@WhenRequested datetime,
@OpeningPrice decimal(18, 2),
@LastTradePrice decimal(18, 2),
@LastTradeTime smalldatetime,
@CurrentPrice decimal(18, 2),
@TodaysHigh      decimal(18, 2)
)
AS
BEGIN
    DECLARE @ID int
    SET @ID = -1

    SELECT @ID = CompanyID FROM Company WHERE
        CompanySymbol = @CompanySymbol

    PRINT @ID
    PRINT @OpeningPrice

    IF(@ID = -1)
    BEGIN
        BEGIN TRANSACTION
        BEGIN TRY

            DECLARE @CompanyID INT
            INSERT INTO Company (CompanyName, CompanySymbol)
            VALUES (@CompanyName, @CompanySymbol)

            SET @CompanyID = @@IDENTITY

            INSERT INTO PriceHistory
            (CompanyID, WhenRequested, OpeningPrice, LastTradePrice,
            LastTradeTime, CurrentPrice, TodaysHigh)
            VALUES
            (
                @CompanyID, @WhenRequested, @OpeningPrice,
                @LastTradePrice, @LastTradeTime, @CurrentPrice,
                @TodaysHigh
            )

            COMMIT TRANSACTION
        END TRY
        BEGIN CATCH
            RAISERROR(''Failed to insert quote %s'', 16, 1, @CompanyName)
            ROLLBACK TRANSACTION
        END CATCH
    END
```

LISTING 17.5 Continued

```
END CATCH
END
ELSE
BEGIN
    SET @CompanyID = @ID

    PRINT ''ID > -1''

    INSERT INTO PriceHistory
    (
        CompanyID, WhenRequested, OpeningPrice, LastTradePrice,
        LastTradeTime, CurrentPrice, TodaysHigh
    )
    VALUES
    (
        @CompanyID, @WhenRequested, @OpeningPrice,
        @LastTradePrice, @LastTradeTime, @CurrentPrice,
        @TodaysHigh
    )
END
END'
END
GO
IF NOT EXISTS (SELECT * FROM sys.foreign_keys WHERE object_id =
OBJECT_ID(N'[dbo].[FK_PriceHistory_Company]') AND parent_object_id =
OBJECT_ID(N'[dbo].[PriceHistory]'))
ALTER TABLE [dbo].[PriceHistory] WITH CHECK ADD CONSTRAINT [FK_PriceHistory_
➥Company] FOREIGN KEY([CompanyID])
REFERENCES [dbo].[Company] ([CompanyID])
```

Writing Code to Obtain the Stock Quotes and Update the Database

For arguments sake, let's say that LINQ is leading edge—if not bleeding edge. In keeping with or lagging edge comparison and contrast, Listing 17.6 contains pretty vanilla code for obtaining stock quotes from Yahoo! A description of the code follows Listing 17.6.

LISTING 17.6 Obtaining Stock Quotes from Yahoo!, Parsing the Returned Data, and Updating the Database via ADO.NET 2.0

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Net;
```

LISTING 17.6 Continued

```
using System.Data.SqlClient;
using System.Data;

namespace StockQuotesWithEntityFramework
{
    class Program
    {
        // rename to use the other Main
        static void _Main(string[] args)
        {
            string[] stocks = { "MSFT", "GOOG", "DELL", "GM", "F",
                "CSCO", "INTC" };

            WebClient client = new WebClient();
            for (int i = 0; i < 25; i++)
            {
                for (int j = 0; j < stocks.Length; j++)
                {
                    string result = client.DownloadString(
                        string.Format("http://download.finance.yahoo.com/d/?s={0}&f=nsolph",
                        stocks[j]));
                    UpdatePriceHistory(result);
                    Console.WriteLine(result);
                    System.Threading.Thread.Sleep(100);
                }
            }
            Console.ReadLine();
        }

        static void UpdatePriceHistory(string result)
        {
            // get rid of goofy characters
            result = result.Replace("<b>", "").Replace("</b>", "");

            // split into separate fields; we still need to split
            // last trade time and price
            string[] data = result.Split(
                new char[]{',', '''', '-'}, StringSplitOptions.RemoveEmptyEntries);

            if(data.Length != 7) return;
            string name = data[0].Trim();
            string symbol = data[1].Trim();
            string open = data[2].Trim();
            string lastTradeTime = data[3].Trim();
```

LISTING 17.6 Continued

```
string lastTradePrice = data[4].Trim();
string price = data[5].Trim();
string high = data[6].Trim();

UpdatePriceHistory(name, symbol, open,
    lastTradeTime, lastTradePrice, price, high);
}

private static void UpdatePriceHistory(string name, string symbol,
    string open, string lastTradeTime,
    string lastTradePrice, string price, string high)
{
    UpdatePriceHistory(name, symbol, ToDecimal(open),
        ToDateTime(lastTradeTime), ToDecimal(lastTradePrice),
        ToDecimal(price), ToDecimal(high));
}

private static void UpdatePriceHistory(string name, string symbol,
    decimal open, DateTime lastTradeTime,
    decimal lastTradePrice, decimal price, decimal high)
{
    string connectionString =
        "Data Source=.\SQLExpress;Initial Catalog=StockHistory;" +
        "Integrated Security=True;Pooling=False";

    using(SqlConnection connection = new SqlConnection(connectionString))
    {
        connection.Open();
        SqlCommand command = new SqlCommand("InsertQuote", connection);
        command.CommandType = CommandType.StoredProcedure;
        command.Parameters.AddWithValue("@CompanyName", name);
        command.Parameters.AddWithValue("@CompanySymbol", symbol);
        command.Parameters.AddWithValue("@WhenRequested", DateTime.Now);
        SqlParameter p1 = command.Parameters.AddWithValue("@OpeningPrice", open);
        p1.DbType = DbType.Decimal;
        p1.Precision = 18;
        p1.Scale = 2;

        SqlParameter p2 = command.Parameters.AddWithValue("@LastTradePrice",
            lastTradePrice);
        p2.DbType = DbType.Decimal;
        p2.Precision = 18;
        p2.Scale = 2;
```

LISTING 17.6 Continued

```
        command.Parameters.AddWithValue("@LastTradeTime", lastTradeTime);
        SqlParameter p3 = command.Parameters.AddWithValue("@CurrentPrice", price);
        p3.DbType = DbType.Decimal;
        p3.Precision = 18;
        p3.Scale = 2;

        SqlParameter p4 = command.Parameters.AddWithValue("@TodaysHigh", high);
        p4.DbType = DbType.Decimal;
        p4.Precision = 18;
        p4.Scale = 2;

        command.ExecuteNonQuery();
    }
}

private static decimal ToDecimal(string val)
{
    return Convert.ToDecimal(val);
}

private static DateTime ToDateTime(string val)
{
    return Convert.ToDateTime(val);
}
}
```

The demo application is a console application with no intermediate objects. The data is read from Yahoo! and sent right to the database. The `Main` function defines the stocks to fetch. The `WebClient` class is a simple class that lets you send a URL and accepts the return value as a string. In `Main`, an outer loop makes 25 requests, and an inner loop makes a request for each stock in the array. In the example, 175 total requests are sent to Yahoo! The formatted URL query requests the named stock by symbol (for example, MSFT for Microsoft) and the second parameter `f` indicates that you would like the name, symbol, opening price, last price, current price, and today's high. (An intuitive guess or experimentation will help you figure out these format characters.) Finally, `UpdatePriceHistory` is called and the results are sent to the console. (The artificial latency of 100 milliseconds was added to stagger the times, but it is unlikely prices will change that fast.)

`UpdatePriceHistory` replaces `` and `` (bold tags) from the string returned from Yahoo! The second statement splits the results by comma, quote, or hyphen. (The last price includes a time and price value.) Next, each string is assigned to a named variable indicating the meaning of the data. Only rudimentary error check is used.

Two overloaded `UpdatePriceHistory` methods are provided. The first accepts the string variables and the second accepts these values converted to the desired type. This is a stylistic way of separating the chunks of code. The final `UpdatePriceHistory` method uses ADO.NET code to connect to the database, populate a `SqlCommand` object with the parameters and desired stored procedure, and the query is run. Two superfluous helper methods perform string conversion functions.

That's all there is to it.

Programming with the Entity Framework

In the demo using ADO.NET 2.0, you got some stock quotes from Yahoo! and updated the database. This section starts with creating an Entity Data Model for the StockHistory database and querying the database through the conceptual data model—the Entity Framework code.

The querying capability demonstrated uses Entity SQL and LINQ to Entities. After you have created the EDM and queried it, you will get a chance to define code that uses a pure LINQ and Entity Framework approach to request the quotes and update the database.

Creating the Entity Data Model

Entity Data Models are composed of source code and XML files. The XML describes the EDM and the source code is the code-generated classes that represent the EDM. You probably could hand code these elements, but you would probably lose most, if not all, of the initial productivity gains by doing so.

To define the EDM, use the wizard.

To define an EDM in any Visual Studio 2008 project, select Project, Add New Item. From the Add New Item dialog box, select the ADO.NET Entity Data Model item. This starts the wizard. For the following steps, begin by naming the `.edmx` file `StockHistoryModel.edmx`.

1. Add the ADO.NET Entity Data Model item from the Add New Item dialog box.
2. Name it something like `StockHistoryModel`; this will start a wizard.
3. On the Choose Model Contents page, select Generate from Database.
4. On the Choose Your Data Connection page, select the StockHistory database.
5. Use the default name of **StockHistoryEntities**. Click Next.
6. Choose Your Data Objects. Select everything.
7. On the Model Namespace page, leave the default (`StockHistoryModel`).
8. Click Finish.

After you click finish the wizard fills out the details of the .edmx file and code generates the source code representing the EDM. After you click Finish, the EDM designer view is displayed, showing the entity elements and expressed relationships.

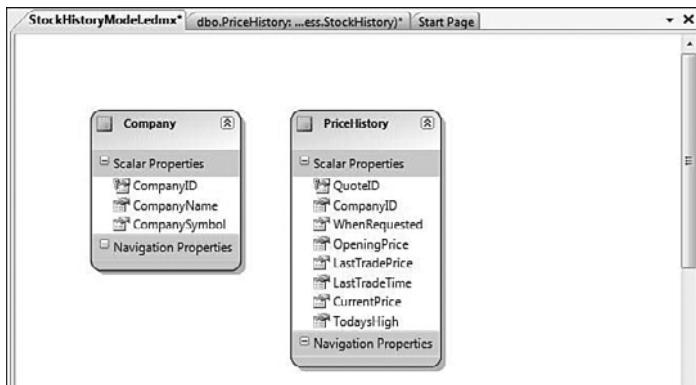


FIGURE 17.4 If you forgot to define the foreign key, the diagram shows the Company and PriceHistory tables without the implied relationship; otherwise, the relationship (visually expressed as a line between the two tables) is shown in the designer.

Adding an Association

If you forgot to define a foreign key, the mapping association is not generated by the wizard. You can define associations after the fact, too. To add an association manually (for the Company and PriceHistory tables), follow these steps:

1. Right-click on the Customer EntityType and select Add, Association.
2. Add the association where one company has many PriceHistory(s) (see Figure 17.5).
3. Click OK. The model designer should show the association now.
4. In the Mapping Details View for the association, click Add a Table or View and select PriceHistory (see Figure 17.6).

Querying the Entity Data Model with Entity SQL

Thus far, all that you have done differently is run the ADO.NET Entity Data Model wizard implicitly by adding an Entity Data Model item from the Add New Item dialog box. That's pretty easy.

After you have the EDM, you can begin querying the data model with Entity SQL or LINQ to Entities right away.

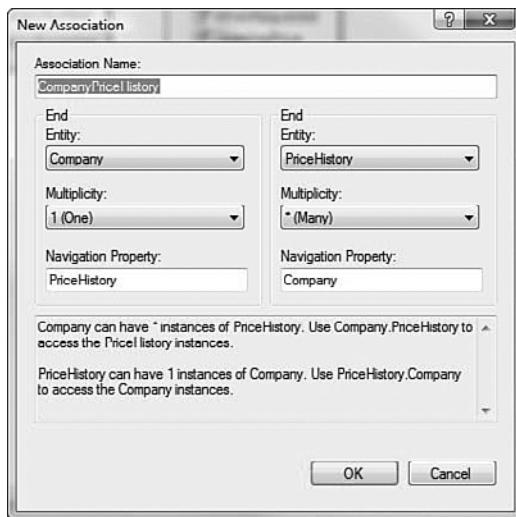


FIGURE 17.5 The New Association dialog box supports adding descriptions after the wizard has run; here, a one-to-many relationship is expressed for Company and PriceHistory.

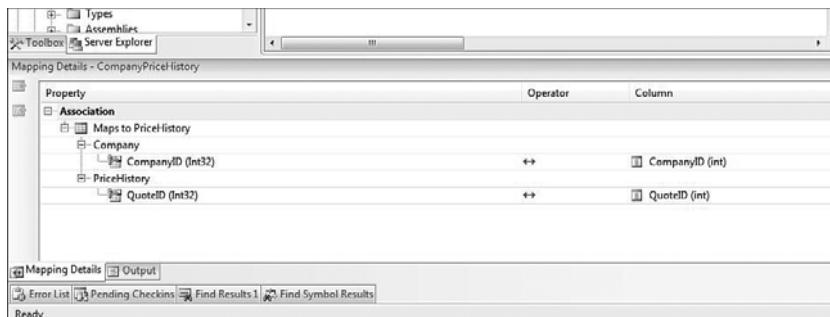


FIGURE 17.6 The CompanyPriceHistory association described, as depicted in Figure 17.5.

Choosing How to Spend Your Time

Entity SQL seems superfluous in the presence of LINQ. Because I don't have an infinite amount of time, my efforts are spent on LINQ to Entities. LINQ supports one programming model for Objects, XML, and SQL, whereas Entity SQL only supports the Entity Framework. Having to make educated guesses on how I spend my time, I suspect LINQ to Entities will have a much longer shelf life than Entity SQL.

Entity SQL is a whole new language. Sometimes, things like Entity SQL are invented in conjunction with technologies like LINQ to Entities and some overlap in capability

results. However, it is worth noting that Entity SQL is a big thing and there isn't enough time or space for a whole tutorial.

In general, Entity SQL is like SQL, but it is based on natural relationships expressed by the Entity Data Model (EDM) instead of joins. Hence, the queries you write refer to the relationships as expressed in the EDM, such as `Company` has a `PriceHistory` and the dot-operator is used instead of a join. There is a lot more to Entity SQL than that. (Check the resources section earlier for links to Entity SQL information.)

The code in Listing 17.5 contains an inline Entity SQL query. One way to use Entity SQL is to create an instance of an `EntityConnection` initializing it with a connection string. (The Entity Data Model Wizard added a connection string in the correct format to an `App.config` file, so use the `ConfigurationManager` to request that.) Next, open the `EntityConnection`, create an `EntityCommand`, and use a `DBDataReader` to access the data. (Look familiar? It's very similar to the pattern for using ADO.NET and regular SQL.)

Notice that the Entity SQL command has a `SELECT`, `FROM`, and `WHERE`. However, also notice that the class name `StockHistoryEntities` is used and the `Company` entity is requested from the `Company` property. The `AS`-clause and the `IN` operator are all pretty similar to SQL though.

The `DBDataReader` is used like a `SQLDataReader` and both actually implement the `IDataReader` interface from `System.Data`. A difference is that a `SqlDataReader` inherits from `DBDataReader`. If you know how to use a `SqlDataReader`, then using a `DbDataReader` isn't that hard; in fact, it's pretty similar (see Listing 17.7).

LISTING 17.7 Using Entity SQL via an `EntityConnection` and `EntityCommand` Follows a Similar Pattern as Does Using SQL and ADO.NET `Sqlclasses` (like `SqlConnection`).

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data.EntityClient;
using System.Data.Common;
using System.Data;
using System.Configuration;
using StockHistoryModel;

namespace StockQuotesWithEntityFramework
{
    class Program2
    {
        static void Main(string[] args)
        {

            string connectionString =

```

LISTING 17.7 Continued

```

ConfigurationManager.ConnectionStrings["StockHistoryEntities"].ConnectionString;

EntityConnection connection = new EntityConnection(connectionString);
connection.Open();
EntityCommand command = new EntityCommand(
    "SELECT company.CompanyID, company.CompanyName " +
    "FROM StockHistoryEntities.Company as company " +
    "WHERE company.CompanySymbol IN {'MSFT', 'F'}", connection);

DbDataReader reader = command.ExecuteReader(CommandBehavior.SequentialAccess);
while (reader.Read())
{
    Console.WriteLine("Company name: {0}", reader.GetValue(1));
}
Console.ReadLine();
}
}
}
}

```

For a complete reference on Entity SQL, refer to <http://msdn.microsoft.com/en-us/library/bb387118.aspx>. This reference contains information about canonical functions, relationship navigation, reference, arithmetic, comparison, logical, and member operators, string concatenation, case expressions type operators, query expressions, comments, namespaces, and an Entity SQL quick reference.

Querying the Entity Data Model with LINQ to Entities

As you will see in Listing 17.8, LINQ to Entities is even easier. All you need to do to query the EDM with LINQ to Entities is create an instance of the `StockHistoryEntities` class and write the LINQ query.

If you check the generated code, you will see that the `StockHistoryEntities` class inherits from the `System.Data.Objects.ObjectContext` class. `ObjectContext` is to LINQ to Entities what `DataContext` is to LINQ to SQL. An `ObjectContext` contains the plumbing like an `EntityConnection` for connecting to the underlying data source. Inside of the custom `ObjectContext`, individual table entities are accessed through properties. These properties are instances of `ObjectQuery<T>`, for example `ObjectQuery<Company>`. An `ObjectQuery<T>` is to LINQ to Entities what a `Table<T>` is to LINQ to SQL. (Are you starting to detect a design pattern here?)

The `ObjectQuery` supports `IEnumerable<T>` and `IQueryable<T>`, which are essential ingredients for supporting LINQ. `ObjectQuery` is defined in the `System.Data.Objects` namespace in the `System.Data.Entity.dll` assembly. Listing 17.8 demonstrates just how easy it is to employ the LINQ skills learned thus far to query the EDM using LINQ.

LISTING 17.8 Using LINQ to Entities to Query the EDM Directly; All of the Underlying SQL Plumbing and Logical Model Elements Are Concealed

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using StockHistoryModel;

namespace StockQuotesWithEntityFramework
{
    class Program1
    {
        // rename to use the other main
        static void Main(string[] args)
        {
            StockHistoryEntities stockHistory = new StockHistoryEntities();

            var quotes = from quote in stockHistory.PriceHistory
                         orderby quote.Company.CompanySymbol
                         select new {Symbol=quote.Company.CompanySymbol,
                                     LastPrice=quote.LastTradePrice, Time=quote.LastTradeTime};

            Array.ForEach(quotes.ToArray(),
                s => Console.WriteLine(s));
            Console.ReadLine();
        }
    }
}
```

What remains to do is to look at how you can use LINQ to Entities to update the data via the conceptual model (the EDM) and LINQ. LINQ to Entities supports updates, too. In the final section of this chapter, updates are demonstrated as well as a rewrite of the original demo, all using LINQ.

Doing It All with LINQ

In this part, the `UpdatePriceHistory` piece from Listing 17.6 was rewritten using almost all LINQ-related technologies. Getting the stock quotes is the same, and basic parsing of the return string is pretty much the same. After that the code diverges. In the example LINQ to XML and *functional construction* is used to build an XML document from the returned data. (You will learn more about LINQ to XML in the final part, Part IV, of this book, so there is just an introduction here.)

Selecting companies uses LINQ to Entities—and there is an alternate chunk that uses the ADO.NET Entity classes like `ObjectQuery`—and the stock quotes are added to the database using the EDM and LINQ to Entities. The code is quite different from Listing 17.6, and the bits are broken into chunks of named methods to help you identify what each chunk is doing. To complete the examples, you can use the same EDM created at the beginning of the “Programming with the Entity Framework” section.

The first half of Listing 17.9 includes the `using` statements and the `Main` function. These didn’t need to change. This part of the code uses a `WebClient` to send a request to Yahoo! and reads the returned string. The changes begin in Listing 17.9’s `UpdatePriceHistory` method.

LISTING 17.9 UpdatePriceHistory, Modified to Use LINQ to XML and LINQ to Entities to Load the Database with Stock Quotes

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Net;
using System.Data.SqlClient;
using System.Data;
using System.Xml.Linq;
using StockHistoryModel;
using System.Data.Objects;

namespace StockQuotesWithEntityFramework
{
    class Program3
    {
        static void Main(string[] args)
        {
            string[] stocks = { "MSFT", "GOOG", "DELL", "GM", "F",
                "CSCO", "INTC" };

            WebClient client = new WebClient();
            for (int i = 0; i < 5; i++)
            {
                for (int j = 0; j < stocks.Length; j++)
                {
                    string result = client.DownloadString(
                        string.Format("http://download.finance.yahoo.com/d/?s={0}&f=nsolph",
                        stocks[j]));
                    UpdatePriceHistory(result);
                    Console.WriteLine(result);
                }
            }
        }
    }
}
```

LISTING 17.9 Continued

```
        System.Threading.Thread.Sleep(100);
    }
}

Console.ReadLine();
}

/// <summary>
/// Use LINQ to XML to get data
/// Use LINQ to Entities to write data to database
/// </summary>
/// <param name="result"></param>
static void UpdatePriceHistory(string result)
{
    XElement xml = CreateXMLElement(result);
    Console.WriteLine(xml.ToString());

    StockHistoryEntities entities = new StockHistoryEntities();

    PriceHistory quote = CreatePriceHistory(xml);

    string name = xml.Element("Name").Value;
    string symbol = xml.Element("Symbol").Value;

#if (!USE_CLASSES)
    var results = from c in entities.Company
                  where c.CompanySymbol == symbol
                  select c;
#else
    ObjectParameter param = new ObjectParameter("symbol", symbol);
    ObjectQuery<Company> results = entities.Company.Where("it.CompanySymbol = @symbol", param);
#endif
    Company company;

    if (results.Count() == 0)
    {
        company = new Company();
        company.CompanyName = name;
        company.CompanySymbol = symbol;
        entities.AddToCompany(company);
    }
    else
    {
        company = results.First();
```

LISTING 17.9 Continued

```
}

company.PriceHistory.Add(quote);
entities.SaveChanges(true);
}

static PriceHistory CreatePriceHistory(XElement xml)
{
    Func< XElement, decimal?> toDecimal =
        val =>
    {
        decimal? temp = (decimal?)val;
        return Math.Round((decimal)val, 2, MidpointRounding.ToEven);
    };

    PriceHistory quote = new PriceHistory();
    quote.OpeningPrice = toDecimal(xml.Element("Open"));
    quote.WhenRequested = DateTime.Now;
    quote.LastTradeTime = Convert.ToDateTime(xml.Element("LastTradeTime").Value);
    quote.LastTradePrice = toDecimal(xml.Element("LastTradePrice"));
    quote.CurrentPrice = toDecimal(xml.Element("Price"));
    quote.TodaysHigh = toDecimal(xml.Element("High"));

    return quote;
}

static XElement CreateXMLElement(string result)
{
    // remove unneeded HTML tags and split into fields
    result = result.Replace("<b>", "").Replace("</b>", "");
    string[] data = result.Split(
        new char[] { ',', '\"', '-' }, StringSplitOptions.RemoveEmptyEntries);

    // use functional construction to create XML document
    return new XElement("Quote",
        new XElement("Name", data[0].Trim()),
        new XElement("Symbol", data[1].Trim()),
        new XElement("Open", data[2].Trim()),
        new XElement("LastTradeTime", data[3].Trim()),
        new XElement("LastTradePrice", data[4].Trim()),
        new XElement("Price", data[5].Trim()));
}
```

LISTING 17.9 Continued

```
new XElement("High", data[6].Trim())
);

}

}
```

The revised code is composed of three new functions: `UpdatePriceHistory`, `CreatePriceHistory`, and `CreateXMLElement`. `UpdatePriceHistory` orchestrates calls to the other two methods, `CreatePriceHistory` and `CreateXMLElement`.

`UpdatePriceHistory` calls `CreateXMLElement` first, returning an `XElement`. `XElement` is introduced as part of LINQ to XML. `CreateXMLElement` accepts a string and uses functional construction to create an XML document in the form of an `XElement` object. Functional construction is the process of creating XML documents via method calls. You will learn a lot more about this in Part IV.

The next thing `UpdatePriceHistory` does is create an instance of the EDM wrapper class, `StockHistoryEntities`, and calls `CreatePriceHistory` to convert the `XElement` object into a `PriceHistory` object. `CreatePriceHistory`, for the most part, copies values out of the XML (`XElement`) object and moves them to a `PriceHistory` object. The generic delegate `Func<T, TResult>` is used to trim up some decimal numbers, but it isn't necessary. (You could simply permit more decimal places in the database in case some of the stock quote values have more than two decimal places.)

Next, the `UpdatePriceHistory` method gets the `Company` values from the XML object (`XElement`). After that, two options are shown. The first option uses LINQ to Entities to see if the company already exists. The second option uses Entity SQL and ADO.NET Entity Framework classes—including `ObjectParameter` and `ObjectQuery`—to see if the company object already exists. You only need to use one of the styles; the LINQ version is a little easier.

If the `Company` doesn't exist, a new one is created and added to the `StockHistoryEntities` collection of companies. (Basically, you are prepping to insert a `Company` row.) Otherwise, if the company exists, that instance is used. In both cases, the `PriceHistory` is added to the `Company`, establishing the underlying relationship and the data is saved. By calling `SaveChanges(true)`, the change flag in the EDM is reset after the save. Without this parameter, you need to call `AcceptAllChanges`.

Listing 17.9 looks about as equally long as `UpdatePriceHistory` in Listing 17.6. However, in Listing 17.9, functions were used to segment the various pieces, and some extra techniques were demonstrated for fun. A big difference between `UpdatePriceHistory` versions is that Listing 17.9 offers a lot more flexibility and power. For example, you have an XML

document (the `XElement`) object, which can be used in a wide variety of scenarios, including presentable data itself. Listing 17.9 also lets you deal with the conceptual model—`Companies` have a `PriceHistory`—and not spend any time on the underlying database plumbing. The result is that you spend your time differently; instead of focusing on stored procedures, joins, logical models, and ADO.NET, your efforts—in Listing 17.9—all focus on the `Company` and `PriceHistory` objects. The result is better focus and better productivity.

Summary

The ADO.NET Entity Framework is part of ADO.NET 3.0. It's still in beta but it is far enough along now that shipping should be imminent—according to Microsoft sources, it will ship with Visual Studio 2008 SP1.

Although you will still have the option of programming in the previous ADO.NET 2.0 style or the ADO.NET 3.0 style, Entity Framework programming will allow you to spend more of your time working on the conceptual business problem.

In previous versions of ADO.NET, C# programmers had to focus on SQL, stored procedures, joins, keys, ADO.NET, and the business problem using several different technologies—like SQL and C#. The ADO.NET Entity Framework will insulate you from nuts-and-bolts database code and let you spend your time on more natural relationships like customers and orders or companies and stock prices. And, you can use one core technology—C#.

The ADO.NET Entity Framework did introduce Entity SQL, but if you use LINQ, you can query SQL, Entities, XML, and objects all with LINQ/C# code. Of course, if you like being a SQL guru too, that's OK, but now everyone doesn't have to be.

Adam Smith wrote in his eighteenth-century book, *The Wealth of Nations*, that improved productivity comes with specialization. This is an insightful economic trend that pervades today's thinking—check out five-term Federal Reserve Chairman Alan Greenspan's book, *The Age of Turbulence: Adventures in a New World*—and specialization will help you and your team be more productive too. Again, technology is fun and so too is knowing SQL, C#, JavaScript, ASP.NET, and a lot of other cool technologies, but the ADO.NET Entity Framework does facilitate some specialization and the result can be enhanced productivity.

If you want productivity, the ADO.NET Entity Framework and working at the conceptual level will help. And, if you just want to do something cool, well ADO.NET 3.0 is way cool, too!

This page intentionally left blank

PART IV

LINQ for XML

IN THIS PART

CHAPTER 18	Extracting Data from XML	415
CHAPTER 19	Comparing LINQ to XML with Other XML Technologies	437
CHAPTER 20	Constructing XML from Non-XML Data	453
CHAPTER 21	Emitting XML with the XmlWriter	463
CHAPTER 22	Combining XML with Other Data Models	469
CHAPTER 23	LINQ to XSD Supports Typed XML Programming	485

This page intentionally left blank

CHAPTER 18

Extracting Data from XML

“Don’t worry. If we screw up they will shoot me first. If I go down stop sucking on my face and run.”

—Jackson Wayfare

Until now, XPath has been the de facto standard for querying Extensible Markup Language (XML). LINQ for XML offers an alternative way to query XML. Each way has its merits and strengths, but LINQ to XML may be easier if you don’t already know XPath.

The key to understanding LINQ to XML is that the basic query structure, grammar, and elements of a LINQ query don’t change just because the source is XML. Like LINQ to SQL, you have to extricate the underlying data items—documents, elements, attributes, and comments—but these all fit in the basic structure of LINQ. In short, everything you have learned thus far is relevant for LINQ to XML. You only need to layer in knowledge about the classes that represent the structure of an XML element and you are up and running.

LINQ to XML supports reading, modifying, transforming, and saving changes to XML documents, as XPath does. This chapter begins with loading and querying documents, handling missing data, annotating nodes, and validating data. The additional capabilities of LINQ to XML are covered in the remaining chapters of this book.

IN THIS CHAPTER

- ▶ Loading XML Documents
- ▶ Querying XML Documents
- ▶ Loading XML from a String
- ▶ Handling Missing Data
- ▶ Using Query Expressions with XML Data
- ▶ Annotating Nodes

Loading XML Documents

When working with LINQ to Objects, the data is in memory. When working with LINQ to SQL, you need to

connect to a data provider—SQL Server—and use a `DataContext` to get the data in memory. For LINQ to XML, you need to load an XML document to get the data in memory. An XML document can be loaded using an instance of `System.Xml.Linq.XDocument` or `System.Xml.Linq XElement` or can be represented as a literal string and then converted to XML using `XElement.Parse`.

Although the following example is a little morbid, assume you have an XML document that tracks a lifetime of family pets called `PetCemetery.xml`. You can use the static method `XDocument.Load("PetCemetery.xml")` to load the entire XML document starting at the root, or you can use `XElement.Load`, which loads the root but lets you skip the root when querying elements.

`XDocument.Load` and `XElement.Load` are overloaded to accept a string filename, a `TextReader`, or an `XmlReader`. One version—`Load(string, LoadOptions)`—loads the XML from a file and `LoadOptions` lets you preserve whitespace, set the base uniform resource identifier (URI), or request line information from the `XmlReader`. `LoadOptions` is defined using the `FlagsAttribute`, which means this enumeration argument is a set; that is, you can assign one or more values together. The literal options for the `LoadOption` enumeration are `None`, `PreserveWhitespace`, `SetBaseUri`, and `SetLineInfo`. The following sections demonstrate `XDocument.Load` and `XElement.Load`.

Querying XML Documents

All of the techniques and underpinnings of LINQ that you have learned in previous chapters apply to this part and to this chapter. The primary difference is that the sequences (or collections, if you prefer) come from XML documents, nodes, attributes, and objects, and these things have their own object representation in the framework.

In this section, you will learn how to use `XDocument`, `XElement`, and manage attributes and write queries against XML documents using these features of the `System.Xml.Linq` namespace.

Using `XDocument`

`System.Xml.Linq.XDocument` inherits from `XContainer`. `XDocument` represents a complete XML document for our purposes. After you have an `XDocument`—you will use the static `Load` method in the example—you can perform LINQ queries against the contents of the document.

Listing 18.1 loads the XML document called `PetCemetery`—an homage to one of my favorite authors, Stephen King, and a history of all of my feline and canine pets over the years. The XML document contains a root node `<pets>` and each immediate descendant is a `<pet>`—one of my pets.

LISTING 18.1 A LINQ to XML Example That Loads an XML Document and Queries the Pets Defined in That Document

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Xml;
using System.Xml.Linq;

namespace XDocumentDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            XDocument xml = XDocument.Load("../..\\PetCemetary.xml");
            var pets = from pet in xml.Elements("pets").Elements("pet")
                       select pet;

            Array.ForEach(pets.ToArray(), p=>Console.WriteLine(p.Element("name").Value));
            Console.ReadLine();
        }
    }
}

```

In the example, the XML document is loaded via the static method `XDocument.Load`. The very next line is a LINQ query. Note that the structure of the query is consistent with previous examples, including the `from` clause, the range and input sequence, and the `select` clause. The sequence is represented in the example by `XDocument.Elements("pets").Elements("pet")`, which, when evaluated, returns a collection of pet nodes. The `Array.ForEach` statement iterates through each pet node and the Lambda Expression requests the `name` element and prints its value. Listing 18.2 contains the complete listing of the sample XML file.

LISTING 18.2 The Sample PetCemetary.xml XML File Contents

```

<?xml version="1.0" encoding="utf-8" ?>
<pets>
    <pet>
        <id>1</id>
        <name>Duke</name>
        <species>Great Dane</species>
        <sex>Male</sex>
        <startYear>1968</startYear>
    </pet>

```

LISTING 18.2 Continued

```
<endYear>1970</endYear>
<causeOfDeath>Suicide</causeOfDeath>
<specialQuality>Big and goofy</specialQuality>
</pet>
<pet>
  <id>2</id>
  <name>Dog</name>
  <species>Some Kind of Cat</species>
  <sex>Female</sex>
  <startYear>1972</startYear>
  <endYear>1974</endYear>
  <causeOfDeath>Car</causeOfDeath>
  <specialQuality>Best mouser</specialQuality>
</pet>
<pet>
  <id>3</id>
  <name>Sam</name>
  <species>Labrador</species>
  <sex>Female</sex>
  <startYear>1973</startYear>
  <endYear>1980</endYear>
  <causeOfDeath>Old Age</causeOfDeath>
  <specialQuality>Great hunting dog</specialQuality>
</pet>
<pet>
  <id>4</id>
  <name>Hogan</name>
  <species>Yellow Lab Mix</species>
  <sex>Male</sex>
  <startYear>1994</startYear>
  <endYear>2004</endYear>
  <causeOfDeath>Seizure</causeOfDeath>
  <specialQuality>A very good dog</specialQuality>
</pet>
<pet>
  <id>5</id>
  <name>Leda</name>
  <species>Chocolate Labrador</species>
  <sex>Female</sex>
  <startYear>2004</startYear>
  <endYear></endYear>
  <causeOfDeath></causeOfDeath>
  <specialQuality>Thumper</specialQuality>
</pet>
```

LISTING 18.2 Continued

```
<pet>
  <id>6</id>
  <name>Po</name>
  <species>Toy Poodle</species>
  <sex>Female</sex>
  <startYear>2003</startYear>
  <endYear>2004</endYear>
  <causeOfDeath>Lethal Injection</causeOfDeath>
  <specialQuality>Mental</specialQuality>
</pet>
<pet>
  <id>7</id>
  <name>Big Mama</name>
  <species>Tabby</species>
  <sex>Female</sex>
  <startYear>1998</startYear>
  <endYear></endYear>
  <causeOfDeath></causeOfDeath>
  <specialQuality>Quarterback</specialQuality>
</pet>
<pet>
  <id>8</id>
  <name>Ruby</name>
  <species>Rotweiler</species>
  <sex>Female</sex>
  <startYear>1997</startYear>
  <endYear></endYear>
  <causeOfDeath></causeOfDeath>
  <specialQuality>Big baby</specialQuality>
</pet>
<pet>
  <id>9</id>
  <name>Nala</name>
  <species>Maine Coon</species>
  <sex>Female</sex>
  <startYear>2007</startYear>
  <endYear></endYear>
  <causeOfDeath></causeOfDeath>
  <specialQuality>La Freaka</specialQuality>
</pet>
</pets>
```

Using XElement

When the document (in Listing 18.1) was loaded with the `XDocument` class, you had to start requesting elements at the root node level, `pets`. If you load the document with `XElement`, as demonstrated in Listing 18.3, you can shorten the chain of element requests—references to `Elements`.

LISTING 18.3 Load the Document With XElement and You Can Request the Desired Elements Directly Without Requesting the Root First

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Xml;
using System.Xml.Linq;

namespace XElementDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            XElement xml = XElement.Load("../\\..\\..\\PetCemetery.xml");
            var pets = from pet in xml.Elements("pet")
                       select pet;

            Array.ForEach(pets.ToArray(), p=>Console.WriteLine(p.Element("name").Value));
            Console.ReadLine();
        }
    }
}
```

Listing 18.3 is almost identical to Listing 18.1. The difference is that Listing 18.3 skipped the `Elements("pets")` piece of the request chain, shortening the LINQ query a bit.

Managing Attributes

The `Elements` property returns an `IEnumerable` of `XElement`. `XElement` objects can have attributes—additional values in the element tag—that contain additional information about the tag. In the original `PetCemetery.xml` file, the `pet` element has a child element `species`. In the hierarchy of classifications, `genus` is next, so you could add a child element `genus` or add a `genus` attribute to the `species` element. The latter would make the `species` element look like the following:

```
<species genus="Dog">Labrador</species>
```

Incorporating information about the genus as an attribute allows you to refine your LINQ queries to incorporate attributed information. For example (as shown in Listing 18.4), you can define a temporary range variable *genus* and add a *where* clause and predicate with the predicate filtering elements that have a *genus* equal to *Feline*.

LISTING 18.4 Querying by Attributes of Elements; in the Example, the Resultset Includes Only Pets in the Genus *Feline*

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Xml;
using System.Xml.Linq;

namespace ManagingAttributesDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            XElement xml = XElement.Load("../\\..\\..\\PetCemetery.xml");
            var pets = from pet in xml.Elements("pet")
                       let genus = pet.Element("species").Attribute("genus")
                       where genus.Value == "Feline"
                       select pet;

            Array.ForEach(pets.ToArray(), p=>Console.WriteLine(p.Element("name").Value));
            Console.ReadLine();
        }
    }
}
```

In the example, the temporary range variable *genus* is defined in the *let* clause. This approach shortens other references to that attribute to the *genus* local range variable. Using *let* is useful for long predicates and those that are used multiple times. The *where* clause's predicate indicates that the LINQ query should only return kitty cats.

ZOOLOGY 101

For you zoologists out there, you will have to forgive me if I have botched the proper zoological classifications. Technically, I think the cat species is *Felis catus* and the genus is *Felis* and the domestic dog's species is *Canis lupus* and the genus is *lupus*. The rest of you get the picture though.

Adding Attributes

You can change attribute values, add new attributes, and remove existing attributes. Attributes are added through the `XElement.Add` method. Listing 18.5 shows a LINQ query that returns pets without the `genus` attribute on the `species` element and adds the `genus` attribute. (In the example, all pets missing the `genus` attribute are assigned the `genus` attribute and the value "Dog".)

LISTING 18.5 Adding New Attributes to an `XElement`

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Xml;
using System.Xml.Linq;

namespace AddingAttributes
{
    class Program
    {
        static void Main(string[] args)
        {
            string filename = "..\\..\\PetCemetery.xml";

            XElement xml = XElement.Load(filename);
            var pets = from pet in xml.Elements("pet")
                       let genus = pet.Element("species").Attribute("genus")
                       where genus == null
                       select pet;

            Array.ForEach(pets.ToArray(), p=>Console.WriteLine(p.Element("name").Value));

            foreach(var pet in pets)
            {
                pet.Element("species").Add(new XAttribute("genus", "Dog"));
            }

            xml.Save(filename);

            Console.ReadLine();
        }
    }
}
```

Removing Attributes

To remove an attribute, you need to invoke the Remove method of the XAttribute object. As the example in Listing 18.6 shows, navigate to the XElement node containing the target attribute. Use the Attribute method, passing in the name of the attribute; this step returns the XAttribute object. Invoke Remove on the returned object. As demonstrated in the listing, these elements can be strung together in a single statement.

LISTING 18.6 Removing an Attribute By Calling the Remove Method of the XAttribute (Attributes Are Added at the XElement Object Level)

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Xml;
using System.Xml.Linq;

namespace RemovingAttributes
{
    class Program
    {
        static void Main(string[] args)
        {
            string filename = "..\\..\\PetCemetery.xml";

            XElement xml = XElement.Load(filename);
            var pets = from pet in xml.Elements("pet")
                       where pet.Element("name").Value == "Ruby"
                       select pet;

            Array.ForEach(pets.ToArray(), p=>Console.WriteLine(p.Element("name").Value));

            foreach(var pet in pets)
            {
                pet.Element("species").Attribute("genus").Remove();
            }

            xml.Save(filename);

            Console.ReadLine();
        }
    }
}
```

Loading XML from a String

Sometimes, Visual Basic (VB) gets left out in the cold, as was the case with anonymous delegates, and sometimes VB gets cool features like My and Literal XML. In VB, you can define literal XML in your VB code. That's cool. In C#, you can define XML in code, but the way you have to do it is to define the XML as a string and then invoke the `XElement.Parse` method.

Listing 18.7 demonstrates how you can define XML in your code as a string, call `XElement.Parse` to convert that string to queryable XML, and then use LINQ to query the XML. Listing 18.7 queries all of the `pet` elements—there is only one in the sample—and then uses `Array.ForEach` to display the name of each pet.

LISTING 18.7 Converting a String Containing XML into Queryable XML with the `XElement.Parse` Method

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Xml.Linq;

namespace XElementParseDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            string xml = "<pets>" +
                "  <pet>" +
                "    <id>2</id>" +
                "    <name>Dog</name>" +
                "    <species>Some Kind of Cat</species>" +
                "    <sex>Female</sex>" +
                "    <startYear>1972</startYear>" +
                "    <endYear>1974</endYear>" +
                "    <causeOfDeath>Car</causeOfDeath>" +
                "    <specialQuality>Best mouser</specialQuality>" +
                "  </pet>" +
                "</pets>";

            XElement elem = XElement.Parse(xml);
            var pets = from pet in elem.Elements("pet")
                      select pet;

            Array.ForEach(pets.ToArray(), p => Console.WriteLine(

```

LISTING 18.7 Continued

```
    p.Element("name").Value));
    Console.ReadLine();
}
}
```

Handling Missing Data

Some data might not exist in your XML. For example, it's valid to have an element with no data, and as you read in the previous section, you can have undefined elements such as attributes. To handle these scenarios, you can use a combination of nullable types, checks for empty element, or null strings.

Listing 18.8 is based on the fact that I have a whole house full of pets that aren't interned in the pet cemetery. The net effect is that the `endYear` elements for cuddly little critters on this side of the hereafter contain no data.

LISTING 18.8 Using Nullable Types and Inline Conditional Checks to Handle Missing Data

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Xml.Linq;

namespace HandlingMissingDataDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            XElement elem = XElement.Load(@"..\..\PetCemetery.xml");
            var pets = from pet in elem.Elements("pet")
                       let endYear = pet.Element("endYear")
                       select new
                       {
                           Name = pet.Element("name").Value,
                           StartYear = (int)pet.Element("startYear"),
                           EndYear = (int?)(endYear.IsEmpty || endYear.Value.Length == 0
                               ? null : (int?)Convert.ToInt32(endYear.Value))
                       };
            Array.ForEach(pets.ToArray(), p =>
            {
                Console.WriteLine("Name: {0}", p.Name);
            });
        }
    }
}
```

LISTING 18.8 Continued

```

        Console.WriteLine("Entered family: {0}", p.StartYear);
        Console.WriteLine("Left family: {0}", p.EndYear);
    });
    Console.ReadLine();
}
}
}
}

```

In the example, the `endYear` element is assigned to a range element of the same name in a `let` statement. In the project, the `EndYear` property is initialized based on an `IsEmpty` check and a `Value.Length == 0` check. If either of these conditions is true, `null` is assigned to the nullable integer; otherwise, the year converted to an integer is assigned to the `EndYear` property of the projection.

It is worth noting the values of attributes and elements always originate from the XML as strings. You can project specific types by converting the string values to the desired type, assuming the types are convertible; for example, “1966” can be converted to an integer.

Using Query Expressions with XML Data

The query techniques you have learned in earlier chapters apply to LINQ to XML, too. To be thorough and to demonstrate subtle differences in how the XML elements are incorporated in queries, the examples in this section demonstrate some common kinds of queries using Yahoo!’s stock quote feature and XML derived from queries to Yahoo!

To request a quote from Yahoo!, enter a uniform resource locator (URL) similar to the following in your browser:

<http://download.finance.yahoo.com/d/?s=goog&f=ncbh>

Listing 18.9 shows the XML file created manually by taking the return values from a couple of quotes and using them to structure an XML file. (The layout was arbitrarily chosen to facilitate demonstrating aspects of LINQ to XML, which includes the fictitious namespace.)

LISTING 18.9 An Arbitrary XML File Created from Actual Stock Quotes from Yahoo!

```

<?xml version="1.0" encoding="utf-8" ?>
<sq:Stocks xmlns:sq="http://www.stock_quotes.com">
    <sq:Stock>
        <sq:Symbol>MSFT</sq:Symbol>
        <sq:Price Change="0.6" Low="42.1" High="51.0">56.0</sq:Price>
    </sq:Stock>
    <sq:Stock>
        <sq:Symbol>MVK</sq:Symbol>
        <sq:Price Change="-3.2" Low="22.8" High="32.4">25.5</sq:Price>
    </sq:Stock>

```

LISTING 18.9 Continued

```
<sq:Stock>
  <sq:Symbol>GOOG</sq:Symbol>
  <sq:Price Change="8.0" Low="24.4" High="34.5">32.0</sq:Price>
</sq:Stock>
<sq:Stock>
  <sq:Symbol>VFINX</sq:Symbol>
  <sq:Price Change="8.0" Low="24.4" High="34.5">32.0</sq:Price>
</sq:Stock>
<sq:Stock>
  <sq:Symbol>HDPMX</sq:Symbol>
  <sq:Price Change="8.0" Low="24.4" High="34.5">32.0</sq:Price>
</sq:Stock>
</sq:Stocks>
```

Using Namespaces

The first thing you will note about Listing 18.9 is that a namespace was added (the `xmlns` attribute). Listing 18.10 demonstrates how you can define an `XNamespace` object and then use that variable as a prefix to all of the `XElement` requests for XML files that use a namespace.

LISTING 18.10 Uses an `XNamespace` to Manage XML Files—from Listing 18.9 Here—Containing Namespaces

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Xml.Linq;

namespace UsingNamespacesWithLinqToXML
{
    class Program
    {
        static void Main(string[] args)
        {
            const string filename = "...\\..\\Stocks.xml";
            XElement xml = XElement.Load(filename);
            XNamespace sq = "http://www.stock_quotes.com";

            var stocks =
                from stock in xml.Elements(sq + "Stock")
                select new {Name=stock.Element(sq + "Symbol").Value};
```

LISTING 18.10 Continued

```

        Array.ForEach(stocks.ToArray(),
            o=>Console.WriteLine(o.Name));
        Console.ReadLine();
    }
}
}

```

Nesting Queries

Nested queries are supported by LINQ and, thus, are supported by LINQ to XML. Nested queries make sense for LINQ to XML because of the hierarchical nature of XML files that have nested nodes. Listing 18.11 produces the same results as Listing 18.12, but uses nesting. The code finds all of the XElement Stock nodes and then the Symbol nodes within the Stock nodes.

Although Listing 18.10 is easier and produces the same results as Listing 18.11, Listing 18.11 does show how to nest queries. Nested queries are especially useful when child nodes contain duplicate elements, for example, when a customer has multiple addresses.

LISTING 18.11 Demonstrating How to Nest LINQ to XML Queries Syntactically

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Xml.Linq;

namespace NestedLinqQueries
{
    class Program
    {
        static void Main(string[] args)
        {
            const string filename = "..\\..\\Stocks.xml";
            XElement xml = XElement.Load(filename);
            XNamespace sq = "http://www.stock_quotes.com";

            var stocks =
                from stock in xml.Elements(sq + "Stock")
                where(
                    from symbol in stock.Elements(sq + "Symbol")
                    select symbol).Any()
                select new {Name=stock.Element(sq + "Symbol").Value};

            Array.ForEach(stocks.ToArray(),
                o=>Console.WriteLine(o.Name));
        }
    }
}

```

LISTING 18.11 Continued

```
        Console.ReadLine();
    }
}
}
```

Filtering with Where Clauses

Where clauses, also called filters, are used to refine the query to selectively return a more specific subset of results. In the example in Listing 18.12, a nested query is used in the outer where clause and an inner where clause is used to check the Change attribute of the Price element. In the example, stocks with a negative price change are returned.

LISTING 18.12 A Nested Query That Returns Stocks with a Price Change—Using the Change Attribute of the Price Node—Less Than 0; That Is, Stocks That Are Going Down in Price

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Xml.Linq;

namespace FilteringWithWhereClauses
{
    class Program
    {
        static void Main(string[] args)
        {
            const string filename = "..\\..\\Stocks.xml";
            XElement xml = XElement.Load(filename);
            XNamespace sq = "http://www.stock_quotes.com";

            var stocksThatLostGround =
                from stock in xml.Elements(sq + "Stock")
                where (
                    from price in stock.Elements(sq + "Price")
                    where (decimal)price.Attribute("Change") < 0
                    select price).Any()
                select stock;

            Array.ForEach(stocksThatLostGround.ToArray(),
                o=>Console.WriteLine(o.Element(sq + "Symbol").Value));
            Console.ReadLine();
        }
    }
}
```

Finding Elements Based on Context

XML documents have a context. For example, the list of stock quotes has a root element `Stocks`. Each `Stock` element is contained within the root context. Based on the way the stock elements are defined, each stock element has a symbol and price information. The symbol and price are subordinate to the stock element, which defines their context.

LINQ to XML supports navigating to elements based on their context. This is accomplished by invoking members such as `XElement.ElementsAfterSelf`, `XElement.ElementsBeforeSelf`, `XElement.NextNode`, and `XElement.HasElements`. In the example, an XML document defined as a string is parsed into an `XElement`, the `XNamespace` is defined, and a query looks for a price change less than 1.

The range variable is defined as all of the `Symbol` elements. The `let` statement is used to create a temporary range of all of the price elements by using the context method `XElement.ElementsAfterSelf` and the `FirstOrDefault` method. The result is that the first `Price` element is returned and assigned to the `price` range. The `where` clause filters the result set by `Price`'s `Change` attribute (see Listing 18.13).

LISTING 18.13 Defining a Local Range Variable `price` Using the Context Method `ElementsAfterSelf`

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Xml.Linq;

namespace FindingElementsBasedOnContext
{
    class Program
    {
        static void Main(string[] args)
        {
            XElement elem = XElement.Parse(
                "<?xml version=\"1.0\" encoding=\"utf-8\" ?>" +
                "<sq:Stocks xmlns:sq=\"http://www.stock_quotes.com\">" +
                "  <sq:Stock>" +
                "    <sq:Symbol>MSFT</sq:Symbol>" +
                "    <sq:Price Change=\"0.6\" Low=\"42.1\" High=\"51.0\">56.0
                   </sq:Price>" +
                "  </sq:Stock>" +
                "  <sq:Stock>" +
                "    <sq:Symbol>MVK</sq:Symbol>" +
                "    <sq:Price Change=\"-3.2\" Low=\"22.8\" High=\"32.4\">25.5
                   </sq:Price>" +
                "  </sq:Stock>" +
```

LISTING 18.13 Continued

```

    "  <sq:Stock>" +
    "    <sq:Symbol>GOOG</sq:Symbol>" +
    "    <sq:Price Change=\"8.0\" Low=\"24.4\" High=\"34.5\">32.0
      =></sq:Price>" +
    "  </sq:Stock>" +
    "  <sq:Stock>" +
    "    <sq:Symbol>VFINX</sq:Symbol>" +
    "    <sq:Price Change=\"8.0\" Low=\"24.4\" High=\"34.5\">32.0
      =></sq:Price>" +
    "  </sq:Stock>" +
    "  <sq:Stock>" +
    "    <sq:Symbol>HDPMX</sq:Symbol>" +
    "    <sq:Price Change=\"8.0\" Low=\"24.4\" High=\"34.5\">32.0
      =></sq:Price>" +
    "  </sq:Stock>" +
  "</sq:Stocks>");

XNamespace sq = "http://www.stock_quotes.com";

var contextStock =
  from symbol in elem.Elements(sq + "Stock").Elements(sq + "Symbol")
  let price = symbol.ElementsAfterSelf().FirstOrDefault()
  where (decimal)(price.Attribute("Change")) < 1M
  select symbol;

Array.ForEach(contextStock.ToArray(), o=>Console.WriteLine(o.Value));
Console.ReadLine();
}
}
}

```

Sorting XML Queries

Sorting is straightforward. Add an `orderby` clause based on the desired criteria and LINQ does the rest. If you modify the query in Listing 18.13 as shown in Listing 18.14, the resultset will be sorted by the `Price's Change` attribute.

LISTING 18.14 Modifying the Query in Listing 18.13 with an `orderby` Clause Permits Sorting the Query Results as You Would Expect

```

var contextStock =
  from symbol in elem.Elements(sq + "Stock").Elements(sq + "Symbol")
  let price = symbol.ElementsAfterSelf().FirstOrDefault()

```

LISTING 18.14 Continued

```

where (decimal)(price.Attribute("Change")) < 1M
orderby (decimal)price.Attribute("Change")
select symbol;

```

Calculating Intermediate Values with Let

The `let` LINQ keyword is an excellent aid in creating temporary range variables. For example, if you want to indicate the difference between the day's low and high values, you could introduce a new value *spread* that is the difference between the `Price`'s `High` and `Low` attribute. This value could then be assigned to an anonymous type using the projection syntax.

In Listing 18.15, the XML string is converted to an `XElement` and the query introduces a calculated value—the spread between the low and high day price—orders the results, and defines a new type that includes the spread using projection syntax in the `select` clause.

LISTING 18.15 Calculating Values Using the `let` Clause

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Xml.Linq;

namespace IntermediateValuesWithLet
{
    class Program
    {
        static void Main(string[] args)
        {
            XElement elem = XElement.Parse(
                "<?xml version=\\"1.0\\" encoding=\\"utf-8\\" ?>" +
                "<sq:Stocks xmlns:sq=\\"http://www.stock_quotes.com\\>" +
                "  <sq:Stock>" +
                "    <sq:Symbol>MSFT</sq:Symbol>" +
                "    <sq:Price Change=\\"0.6\\" Low=\\"42.1\\" High=\\"51.0\\">56.0
                   </sq:Price>" +
                "  </sq:Stock>" +
                "  <sq:Stock>" +
                "    <sq:Symbol>MVK</sq:Symbol>" +
                "    <sq:Price Change=\\"-3.2\\" Low=\\"22.8\\" High=\\"32.4\\">25.5
                   </sq:Price>" +
                "  </sq:Stock>" +
                "  <sq:Stock>" +
                "    <sq:Symbol>GOOG</sq:Symbol>" +

```

LISTING 18.15 Continued

```
"      <sq:Price Change=\"8.0\" Low=\"24.4\" High=\"34.5\">32.0
    ↳</sq:Price>" +
"  </sq:Stock>" +
"  <sq:Stock>" +
"    <sq:Symbol>VFINX</sq:Symbol>" +
"    <sq:Price Change=\"8.0\" Low=\"24.4\" High=\"34.5\">32.0
    ↳</sq:Price>" +
"  </sq:Stock>" +
"  <sq:Stock>" +
"    <sq:Symbol>HDPMX</sq:Symbol>" +
"    <sq:Price Change=\"8.0\" Low=\"24.4\" High=\"34.5\">32.0
    ↳</sq:Price>" +
"  </sq:Stock>" +
"  </sq:Stocks>");

XNamespace sq = "http://www.stock_quotes.com";

var stockSpreads =
  from stock in elem.Elements(sq + "Stock")
  let spread = (decimal)stock.Element(sq + "Price").Attribute("High") -
    (decimal)stock.Element(sq + "Price").Attribute("Low")
  orderby spread descending
  select new {Symbol=stock.Element(sq + "Symbol").Value, Spread=spread};

Array.ForEach(stockSpreads.ToArray(), o=>Console.WriteLine(o));
Console.ReadLine();
}
}
```

Annotating Nodes

Node annotations are used to add useful information for coding purposes. For example, you could annotate stock nodes with the query used to return the stock price if you wanted to update the node at some future point. Annotations are not serialized to the XML file, but you can add annotations, remove them from the `XElement`, and access them for use in code by using the `Annotation` method of the `XElement` object.

Annotations are supported for `XElement`, `XAttribute`, `XCData`—`CDATA` nodes—and `XDocument` objects. Basically, `AddAnnotation` is defined in `XNode` and `XContainer` and the elements of XML documents are derived from one or the other of these abstract classes.

LISTING 18.16 Demonstrating How to Annotate an XElement with a Query Used to Request the Stock Price from Yahoo!—Even Though This Information Is Useful in Code, It Won't Be Serialized with the XML

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Xml.Linq;

namespace AddAnnotationToXML
{
    class Program
    {
        static void Main(string[] args)
        {
            const string filename = "...\\..\\Stocks.xml";
            XElement elem = XElement.Load(filename);
            XNamespace sq = "http://www.stock_quotes.com";

            var stocksToAnnotate =
                from stock in elem.Elements(sq + "Stock")
                select stock;

            const string yahooQuery =
                "http://download.finance.yahoo.com/d/?" +
                "s={0}&f=ncbh";
            foreach(var stock in stocksToAnnotate)
            {
                stock.AddAnnotation(string.Format(yahooQuery,
                    stock.Element(sq + "Symbol").Value));
                Console.WriteLine(stock.Annotation(typeof(Object)));
            }
            Console.ReadLine();
        }
    }
}
```

In Listing 18.16, the annotation is added to the `Stock` element and immediately read back—for demonstration purposes—illustrating how the annotation is retrieved. Note that the argument to `Annotation` is a `Type` argument. This implies that an annotation can be any object, and, consequently, it is possible to associate behaviors in the form of objects as annotations to XML elements.

TIP

Sometimes, XML is poorly formed. Poorly formed XML causes an exception. To manage potential exceptions, wrap your code in a `try...catch` block using the `System.Xml.XmlException` to trap problems that might arise in the XML source.

Summary

XML looks hard to read at first glance. However, at its core, XML is matching tag pairs that are nested and can have attributes. The very general nature and simplicity of XML make it powerful; CData, XML Transforms, and XPath make XML more powerful but also introduce usage complexities related to learning additional technologies, such as XPath or XSLT. You will learn about these subjects in Chapter 19, “Comparing LINQ to XML with Other XML Technologies.”

LINQ to XML makes manipulating XML uniform. Until now, programmers had to know C# to manipulate objects, SQL to manipulate databases, and XPath and XSLT to manipulate XML. LINQ offers a uniform way to handle all of these different kinds of things. A uniform way of handling things permits you to focus on the problem rather than wrangling with the technology.

This page intentionally left blank

CHAPTER 19

Comparing LINQ to XML with Other XML Technologies

“Behind every successful man stands a surprised mother-in-law.”

—Voltaire

There are too many technologies to master all of them. For instance, you might know C# very well and, by association, VB .NET. But what about Ruby, Silverlight, F#, XML, JavaScript, XPath (XML Path Language), and XSLT (Extensible Stylesheet Language Transformations), VBA, SQL, HTML, DHTML, and the list goes on? In my case, I work diligently to know some core, mainstream object-oriented languages like C#, C++, and VB .NET and SQL extremely well. Other things like JavaScript, HTML, XML I know well, and still others like ActivePerl, Ruby, Java, XPath, and XSLT I know less well. It simply comes down to a matter of choice, time, and necessity.

XPath (XML Path Language) is an expression language for selecting nodes in an XML (eXtensible Markup Language) document. XSLT (Extensible Stylesheet Language Transformations) is a declarative language for transforming an XML document from one shape to another. For example, XSLT can be used to transform an XML document into an HTML document. The challenge is that XPath and XSLT are completely separate languages from something like C#. Each has its own grammar, rules, uses, and learning curve. Each (XPath and XSLT) has its own value and strengths, and offers capabilities that might be hard to replicate well any other way. However, LINQ to XML offers some of the same capabilities.

Both XPath and XSLT have entire books written about them, and that information cannot be repeated here. That said, in this chapter, you learn a little about XPath and

IN THIS CHAPTER

- ▶ Comparing LINQ to XML with XPath
- ▶ Comparing LINQ to XML Transformations with XSLT
- ▶ Transforming XML Data Using Functional Construction

XSLT and, more important to this text, you learn how LINQ to XML can provide some of the same capabilities. The benefit to you is that if you know C#, for example, but not XSLT, you will be able to accomplish some of the same goals—as provided by XPath and XSLT—without being required to learn those other technologies. If you have the time and inclination, you are encouraged to become familiar with XPath and XSLT, so you at least know about the strengths and weaknesses of them and can make judicious decisions about when one or the other might be a more suitable choice.

Comparing LINQ to XML with XPath

A few years ago, I wrote a Blackjack game. The game uses statistics from a book on expert play and coaches the player in the statistically best play based on the player's and dealer's hands. (The game and source is available from my website at <http://www.softconcepts.com>.) The game was good enough that a programmer from Harrah's in Biloxi asked to use the source, and my understanding is that it is a pillow favor provided at the casino. In this chapter, game statistics from a round of play were saved as an XML file and that file is used for the demos. (You can download the game and save your own statistics or use the XML provided here in Listing 19.1.)

LISTING 19.1 Statistics from a Round of Play in the Blackjack Game Saved to an XML File

```
<?xml version="1.0" encoding="utf-8"?>
<Blackjack>
    <Player Name="Player 1">
        <Statistics>
            <AverageAmountLost>-28.125</AverageAmountLost>
            <AverageAmountWon>30.6818181818183</AverageAmountWon>
            <Blackjacks>1</Blackjacks>
            <Losses>8</Losses>44
            <NetAverageWinLoss>5.9210526315789478</NetAverageWinLoss>
            <NetWinLoss>112.5</NetWinLoss>
            <PercentageOfBlackJacks>0.04166666666666664</PercentageOfBlackJacks>
            <PercentageOfLosses>33.3333333333329</PercentageOfLosses>
            <PercentageOfPushes>16.66666666666664</PercentageOfPushes>
            <PercentageOfWins>45.83333333333329</PercentageOfWins>
            <Pushes>4</Pushes>
            <Surrenders>1</Surrenders>
            <TotalAmountLost>-225</TotalAmountLost>
            <TotalAmountWon>337.5</TotalAmountWon>
            <Wins>11</Wins>
        </Statistics>
    </Player>
</Blackjack>
```

Screenshots from the game are shown in Chapter 10, “Mastering Select and SelectMany,” and the card images were taken from the cards.dll library that was used to create such Windows games as Solitaire. Examples of how to use the cards.dll are all over the web, including in some of my articles, such as “Programming for Fun and Profit—Using the Card.dll” at <http://www.developer.com/net/vb/article.php/3303671>.

The basic flow of the subsections that follow is that you are shown some code that uses LINQ to XML to query nodes followed by an equivalent XPath query that accomplishes the same goal. (You don’t need both; in practice, use one or the other.)

Using Namespaces

XML documents support namespaces. For example, if you add the following namespace to the XML file in Listing 19.1 (right after the `<xml>` tag—both are shown), you need to include the namespace in both your LINQ to XML and your XPath queries.

```
<?xml version="1.0" encoding="utf-8"?>
<jack:Blackjack xmlns:jack="http://www.blackjack.com">
```

The XML in Listing 19.2 shows the proper placement of the namespace `jack` added to the XML from Listing 19.1. The code in Listing 19.3 incorporates the namespace in the LINQ to XML to obtain the net amount won (or lost) from the XML file in Listing 19.1. The second half of the listing uses the `XPathSelectElement` method and an XPath query to obtain the same value.

LISTING 19.2 The XML from Listing 19.1 with the Namespace `jack` Added

```
<?xml version="1.0" encoding="utf-8"?>
<jack:Blackjack xmlns:jack="http://www.blackjack.com">
  <jack:Player Name="Player 1">
    <jack:Statistics>
      <jack:AverageAmountLost>-28.125</jack:AverageAmountLost>
      <jack:AverageAmountWon>30.6818181818183</jack:AverageAmountWon>
      <jack:Blackjacks>1</jack:Blackjacks>
      <jack:Losses>8</jack:Losses>44
      <jack:NetAverageWinLoss>5.9210526315789478</jack:NetAverageWinLoss>
      <jack:NetWinLoss>112.5</jack:NetWinLoss>
      <jack:PercentageOfBlackJacks>0.04166666666666664</jack:
        PercentageOfBlackJacks>
      <jack:PercentageOfLosses>33.3333333333329</jack:PercentageOfLosses>
      <jack:PercentageOfPushes>16.66666666666664</jack:PercentageOfPushes>
      <jack:PercentageOfWins>45.8333333333329</jack:PercentageOfWins>
      <jack:Pushes>4</jack:Pushes>
      <jack:Surrenders>1</jack:Surrenders>
      <jack:TotalAmountLost>-225</jack:TotalAmountLost>
      <jack:TotalAmountWon>337.5</jack:TotalAmountWon>
      <jack:Wins>11</jack:Wins>
    </jack:Statistics>
```

LISTING 19.2 Continued

```
</jack:Player>
</jack:Blackjack>
```

LISTING 19.3 The Main Function Uses LINQ to XML and a Namespace to Obtain a Value, and an Equivalent XPath Query to Obtain the Same Value

```
using System.Xml;
using System.Xml.Linq;
using System.Xml.XPath;

private static void UseNamespace()
{
    const string filename = "...\\..\\CurrentStatsWithNamespace.xml";
    XDocument doc = XDocument.Load(filename);
    XNamespace jack = "http://www.blackjack.com";

    XElement winLoss1 = doc.Element(jack + "Blackjack")
        .Element(jack + "Player").Element(
            jack + "Statistics").Element(jack + "NetWinLoss");

    Console.WriteLine(winLoss1);
    Console.ReadLine();

    XmlReader reader = XmlReader.Create(filename);
    XElement root = XElement.Load(reader);
    XmlNameTable table = reader.NameTable;
    XmlNamespaceManager manager = new XmlNamespaceManager(table);
    manager.AddNamespace("jack", "http://www.blackjack.com");

    XElement winLoss2 =
        doc.XPathSelectElement(
            "./jack:Blackjack/jack:Player/jack:Statistics/jack:NetWinLoss",
            manager);
    Console.WriteLine(winLoss2);
    Console.ReadLine();
}
```

In the example, an `XmlReader` was created from the XML file. The root `XElement` was obtained from the reader, followed by the `Nametable`. The `Nametable` is an instance of the `System.Xml.Nametable` class, and it contains the atomized names of the elements and attributes of the XML document. If a name appears multiple times in an XML document, it is stored only once in a `Nametable`, as a Common Language Runtime (CLR) object. Such

storage permits object comparisons on these elements and attributes rather than a much more expensive string comparison. (This is managed for you.)

Next, the table is used to create an `XmlNamespaceManager` and the desired XML namespace string is added to the manager. Finally, the `XmlNamespaceManager` is passed as an argument to the `XPathSelectElement` method. The XPath query is `"./jack:Blackjack/jack:Player/jack:Statistics/jack:NetWinLoss"`. The subpath `"jack:"` demonstrates how to incorporate the namespace in the XPath query.

Our examples use the XPath support provided by LINQ to XML in the `System.Xml.Linq` namespace. XPath support is provided in `System.Xml.XPath` too, and you would use different classes and behaviors if you were to use that approach. As an exercise, if you are interested, you can experiment by implementing the equivalent behaviors using the capabilities of the XPath namespace.

Finding Children

Another thing you might want to do is find the value of child elements. To trim up the code for this example, you can use the XML in Listing 19.1 without the namespace. The LINQ to XML uses imperative code and the `XElement` object chained together to request children, and the XPath query uses a value that looks a lot like a file path statement (see Listing 19.4).

LISTING 19.4 Selecting a Child Element with LINQ to XML and Then an XPath Query

```
private static void FindChild()
{
    const string filename = "...\\..\\CurrentStats.xml";
    XElement xml = XElement.Load(filename);
    XElement child1 = xml.Element("Player")
        .Element("Statistics").Element("AverageAmountLost");
    Console.WriteLine(child1);
    Console.ReadLine();

    // XPath expression using System.Xml.Linq capabilities
    XElement child2 = xml.XPathSelectElement("Player/Statistics/AverageAmount-
Lost");
    Console.WriteLine(child2);
    Console.ReadLine();
}
```

The first half of Listing 19.3 is consistent with code introduced in Chapter 18, “Extracting Data from XML.” The second half uses an XPath query, `Player/Statistics/AverageAmountLost`. Because both parts are using capabilities and classes in the `System.Xml.Linq` namespace, you can easily blend queries and chained `XElement` calls in the same code block.

Finding Siblings

In Chapter 18, you were introduced to methods like `XElement.ElementsAfterSelf` to request sibling elements. The first half of Listing 19.5 requests the next sibling element and the second half uses an XPath query to perform the same task.

LISTING 19.5 LINQ to XML and XPath Supporting Requesting Siblings, Children, and Parents

```
private static void FindSibling()
{
    const string filename = "..\\..\\CurrentStats.xml";
    XElement xml = XElement.Load(filename);
    XElement child1 = xml.Element("Player")
        .Element("Statistics").Element("AverageAmountWon");
    XElement sibling1 = child1.ElementsAfterSelf().First();
    Console.WriteLine(sibling1);
    Console.ReadLine();

    XElement child2 = xml.XPathSelectElement("Player/Statistics/AverageAmountWon");
    XElement sibling2 = child2.XPathSelectElement("following-sibling::*");
    Console.WriteLine(sibling2);
    Console.ReadLine();
}
```

The XPath query (or XQuery) `following-sibling::*` illustrates where I think XPath becomes less intuitive. The path statement `Player/Statistics/AverageAmount` looks like a path; `following-sibling::*` begs for a trip to the help documentation. However, because XPath is a W3C (World Wide Web Consortium) open standard, it is unlikely they will change it for us.

NOTE

XPath and XSLT are W3C open standards determined by a committee (or consortium). LINQ to XML is a proprietary part of the .NET Framework. This difference alone might discourage “open standards” wonks from using LINQ to XML, but something that can be gleaned by intuition gets higher marks than open standards for standards sake with me.

Filtering Elements

In Chapter 18, filtering XML documents with LINQ queries and `where` clauses was demonstrated. Listing 19.6 demonstrates how to query the `Player` element with LINQ to XML and a LINQ query, and the second half of the code shows the equivalent behavior using an XQuery. Again, the LINQ query seems more intuitive than the XQuery `Player[@Name='Player 1']`.

LISTING 19.6 Filtering with LINQ and XQuery

```
private static void FilterOnAttribute()
{
    const string filename = "..\\..\\CurrentStats.xml";
    XElement xml = XElement.Load(filename);

    XElement player1 =
        (from elem in xml.Elements("Player")
         where elem.Attribute("Name").Value == "Player 1"
         select elem).First();

    Console.WriteLine(player1);

    XElement player2 = xml.XPathSelectElement("Player[@Name='Player 1']");
    Console.WriteLine(player2);
    Console.ReadLine();
}
```

In the XQuery `Player[@Name='Player 1']`, `Player` is the node and the bracketed `@Name` part refers to the `Name` attribute and its value. The correct statement looks like an index operation, but perhaps from a C# programmer's point of view `Player.Name = 'Player 1'` would be more intuitive. This lends itself to my argument that if you are comfortable with C#, then using LINQ to XML and method calls might be significantly easier to pick up than XPath queries.

Without an exhaustive comparison of XPath and LINQ to XML, you get the idea. LINQ to XML is going to be `XDocument` and `XElement` method calls and XPath is going to be queries defined by that standard; the XPath query syntax is distinct from C# code. At some level, you will be able to do more with less typing if you use XPath just as you can do some very advanced comparisons with regular expressions with less typing. The decision matrix that helps you decide which technology to use depends on your experience and the experience of the members of your team.

Comparing LINQ to XML Transformations with XSLT

XSL Transformations (XSLT) is distinct from XPath. Where XPath is used to query nodes, XSLT is used to transform documents. Again, you can elect to use LINQ to XML and what is referred to as *functional construction* to transform documents or you can use XSLT to transform documents. Again, too, LINQ to XML employs imperative lines of code and XSLT uses declarative code.

NOTE

The neat thing about declarative code and the way XSLT works is that you can define multiple variations of a transform, and depending on the actual transform applied, you can derive a completely different result.

Listing 19.7 contains a modest XSL stylesheet document that when applied to the XML in Listing 19.1 transforms the XML document into an HTML document. If you examine the code, you will see the XSLT elements and the HTML elements. The data in the XML file is plugged into the HTML body by the transformation statements.

LISTING 19.7 An XSLT Document That Converts the XML Document in Listing 19.1 to an HTML Page

```
<?xml version='1.0'?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
  <xsl:template match="/">
    <html>
      <head>
        <style type="text/css">
          table
          {
            cellspacing: 0;
            cellpadding: 0;
            border-top: "1 solid #000000";
            border-bottom: "1 solid #000000";
            border-right: "1 solid #000000";
            border-left: "1 solid #000000";
          }
          td.Header
          {
            background-color: "#000111";
            color: "#FFFFFF";
            border: "1 solid #000000";
          }
          td.Name
          {
            background-color: "silver";
            border: "1 solid #000000";
          }
          td.Data
          {
            text-align: right;
            width: 100px;
            border-bottom: "1 solid #000000";
          }
        </style>
      </head>
      <body>
        <table>
          <tr>
            <td>Header</td>
            <td>Name</td>
            <td>Data</td>
          </tr>
          <tr>
            <td>Header</td>
            <td>Name</td>
            <td>Data</td>
          </tr>
        </table>
      </body>
    </html>
  </xsl:template>
</xsl:stylesheet>
```

LISTING 19.7 Continued

```
</style>
</head>
<body>
<table cellspacing="0" cellpadding="0">
<xsl:for-each select="Blackjack/Player/Statistics">
<tr>
<td class="Header">Name</td>
<td class="Header">Value</td>
</tr>
<tr>
<td class="Name">
    Average Amount Lost:
</td>
<td class="Data">
<strong>
<xsl:value-of select='format-number(AverageAmountLost,
    "#.000")' />
</strong>
</td>
</tr>
<tr>
<td class="Name">
    Average Amount Won:
</td>
<td class="Data">
<strong>
<xsl:value-of select='format-number(AverageAmountWon, "#.000")' />
</strong>
</td>
</tr>
<tr>
<td class="Name">
    Losses:
</td>
<td class="Data">
<strong>
<xsl:value-of select="Losses" />
</strong>
</td>
</tr>
<tr>
<td class="Name">
    Blackjacks:
</td>
<td class="Data">
```

LISTING 19.7 Continued

```
<strong>
    <xsl:value-of select="Blackjacks" />
</strong>
</td>
</tr>
<tr>
    <td class="Name">
        Net Average Win Loss:
    </td>
    <td class="Data">
        <strong>
            <xsl:value-of select='format-number(NetAverageWinLoss,
                "0.000")' />
        </strong>
    </td>
</tr>
<tr>
    <td class="Name">
        Net Win Loss:
    </td>
    <td class="Data">
        <strong>
            <xsl:value-of select="NetWinLoss" />
        </strong>
    </td>
</tr>
<tr>
    <td class="Name">
        Percentage of Blackjacks:
    </td>
    <td class="Data">
        <strong>
            <xsl:value-of select='format-number(PercentageOfBlackJacks,
                "0.000%")' />
        </strong>
    </td>
</tr>
<tr>
    <td class="Name">
        Percentage of Losses:
    </td>
    <td class="Data">
        <strong>
            <xsl:value-of select='format-number(PercentageOfLosses div 100,
                "0.000%")' />
        </strong>
    </td>
</tr>
```

LISTING 19.7 Continued

```
        </strong>
    </td>
</tr>
<tr>
    <td class="Name">
        Percentage of Pushes:
    </td>
    <td class="Data">
        <strong>
            <xsl:value-of select='format-number(PercentageOfPushes div 100,
                "0.000%")' />
        </strong>
    </td>
</tr>
<tr>
    <td class="Name">
        Percentage of Wins:
    </td>
    <td class="Data">
        <strong>
            <xsl:value-of select='format-number(PercentageOfWins div 100,
                "0.000%")' />
        </strong>
    </td>
</tr>
<tr>
    <td class="Name">
        Pushes:
    </td>
    <td class="Data">
        <strong>
            <xsl:value-of select="Pushes" />
        </strong>
    </td>
</tr>
<tr>
    <td class="Name">
        Pushes:
    </td>
    <td class="Data">
        <strong>
            <xsl:value-of select="Pushes" />
        </strong>
    </td>
</tr>
<tr>
```

LISTING 19.7 Continued

```
<td class="Name">
    Surrenders:
</td>
<td class="Data">
    <strong>
        <xsl:value-of select="Surrenders" />
    </strong>
</td>
</tr>
<tr>
    <td class="Name">
        Total Amount Won or Lost:
    </td>
    <td class="Data">
        <strong>
            <xsl:value-of select="TotalAmountLost" />
        </strong>
    </td>
</tr>
<tr>
    <td class="Name">
        Total Amount Won:
    </td>
    <td class="Data">
        <strong>
            <xsl:value-of select="TotalAmountWon" />
        </strong>
    </td>
</tr>
<tr>
    <td class="Name">
        Wins:
    </td>
    <td class="Data">
        <strong>
            <xsl:value-of select="Wins" />
        </strong>
    </td>
</tr>
</xsl:for-each>
</table>
</body>
</html>
</xsl:template>
</xsl:stylesheet>
```

The parts that do the work are those elements in the `<xsl>` tag. For example,

```
<xsl:value-of select='format-number(AverageAmountWon, "#.000")' />
```

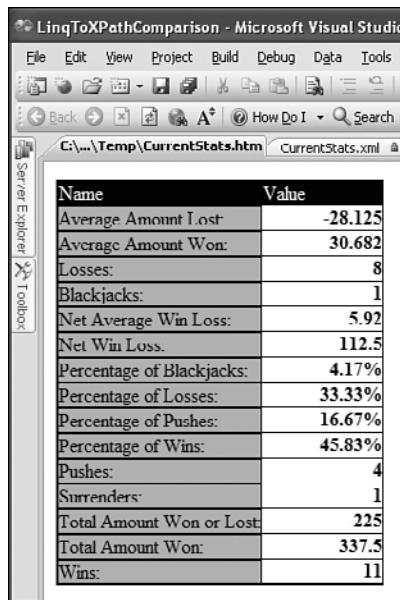
indicates that the value `AverageAmountWon` should be selected from the XML source document. `format-number` is a function call and the format string `"#.000"` indicates a number with three points of precision. The statement

```
<xsl:value-of select='format-number(PercentageOfLosses div 100, "0.000%")' />
```

demonstrates how to perform arithmetic—in this instance, division—and format the value for `PercentageOfLosses` with the % symbol.

To test the transform in Microsoft Visual Studio, complete the following steps:

1. Open the XML document.
2. Press F4 to access the Properties window.
3. Click the Browse button for the Stylesheet property (of the XML document) and select the desired XSLT document.
4. Select the XML menu and the Show XSLT Output menu item. (The output is shown in Figure 19.1.)



The screenshot shows the Microsoft Visual Studio interface with the title bar "LinqToXPathComparison - Microsoft Visual Studio". The menu bar includes File, Edit, View, Project, Build, Debug, Data, Tools. Below the menu is a toolbar with various icons. The main area shows a browser-like view with tabs for "C:\...\Temp\CurrentStats.htm" and "CurrentStats.xml". The "CurrentStats.htm" tab is active, displaying a table of current statistics:

Name	Value
Average Amount Lost	-28.125
Avgcage Amount Won:	30.682
Losses:	8
Blackjacks:	1
Net Average Win Loss:	5.92
Net Win Loss.	112.5
Percentage of Blackjacks:	4.17%
Percentage of Losses:	33.33%
Percentage of Pushes:	16.67%
Percentage of Wins:	45.83%
Pushes:	4
Surrenders:	1
Total Amount Won or Lost	225
Total Amount Won:	337.5
Wins:	11

FIGURE 19.1 The output from the XML document in Listing 19.1 and the XSLT document in Listing 19.7.

XSLT takes some work to master, but you can do some amazing things with it. The Integrated Development Environment (IDE) also supports XSLT debugging. To debug an

XSLT document in the IDE, which permits stepping through `<xsl>` statements, open the XML document and select the XML, Debug XSLT menu item. A debugging session for the document in Listing 19.7 is shown in Figure 19.2.



FIGURE 19.2 A debugging session for the XSLT document in Listing 19.7.

LINQ to XML is a possible alternative choice for XSLT that you might want to use in some circumstances. Again, XSLT is a W3C standard and LINQ to XML is a proprietary part of the .NET Framework. The factors that go into which technology you choose include the technology you know and the approach that is sufficient to get the job done. There are benefits to open standards, and that might be part of your decision matrix.

Converting an XML document with LINQ to XML is called *functional construction*. Quite literally, this means using imperative code and function calls to take one thing and construct another.

Transforming XML Data Using Functional Construction

Functional construction is quite literally a way to create an XML tree in a single statement by chaining function calls together where subordinate calls are arguments to the calling method. (This same nesting of method calls is also often used in the `CodeDOM` namespace to create code graphs.)

The XML document in Listing 19.1 was created with functional construction. Quite straightforward really, the XML tree was created by chaining `XElement` (and `XAttribute`) objects together to form the shape of the XML document and calling the `XElement.Save` method.

LISTING 19.8 Using *Functional Construction*, or Chains of `System.Xml.Linq` Objects to Shape the Desired Form of the XML Output

```
private void SerializeGameStatistics(BlackJack game)
{
    try
    {
        Statistics stats = game.Players[0].Statistics;
        //serialize game to XML
        XElement xml =
            new XElement("Blackjack",
                new XElement("Player",
                    new XAttribute("Name", game.Players[0].Name),
                    new XElement("Statistics",
                        new XElement("AverageAmountLost", stats.AverageAmountLost),
                        new XElement("AverageAmountWon", stats.AverageAmountWon),
                        new XElement("Blackjacks", stats.BlackJacks),
                        new XElement("Losses", stats.Losses),
                        new XElement("NetAverageWinLoss", stats.NetAverageWinLoss),
                        new XElement("NewWinLoss", stats.NetWinLoss),
                        new XElement("PercentageOfBlackJacks",
                            stats.PercentageOfBlackJacks),
                        new XElement("PercentageOfLosses", stats.PercentageOfLosses),
                        new XElement("PercentageOfPushes", stats.PercentageOfPushes),
                        new XElement("PercentageOfWins", stats.PercentageOfWins),
                        new XElement("Pushes", stats.Pushes),
                        new XElement("Surrenders", stats.Surrenders),
                        new XElement("TotalAmountLost", stats.TotalAmountLost),
                        new XElement("TotalAmountWon", stats.TotalAmountWon),
                        new XElement("Wins", stats.Wins))));

        xml.Save(
            Path.GetDirectoryName(
                Application.ExecutablePath) + "\\CurrentStats.xml");
    }
    catch{}
}
```

The method in Listing 19.8 was added to the Blackjack sample application. The data is derived from objects in that game, but the orchestration of the `XElement` objects could really be applied to any objects.

Summary

As a primer for XPath and XSLT, this chapter barely scratched the surface of those technologies, but that was not its intent. XPath and XSLT are open standards for querying and transforming XML documents; however, they are completely different technologies than C# programming, which you already know. Now C# with LINQ to SQL is also a technology for querying and transforming XML documents, and if you are reading this book, then you either know C# or are well on your way to learning it.

At no time was it my intent to imply that XPath or XSLT are bad. Rather, the implication is that C#, classes, function calls, and arguments (and now LINQ) are things that you already know how to do. With LINQ to XML, you can now do some of the things provided by XPath and XSLT in a way you already know how to do them.

CHAPTER 20

Constructing XML from Non-XML Data

IN THIS CHAPTER

- ▶ Constructing XML from CSV Files
- ▶ Generating Text Files from XML
- ▶ Using XML and Embedded LINQ Expressions (in VB)

“Thinking is the hardest work there is, which is probably the reason why so few engage in it.”

—Henry Ford

The previous chapter provided some technical comparisons and contrasts between LINQ to XML, XSLT, and XPath. (LINQ is Language Integrated Query, XML is eXtensible Markup Language, and XSLT is eXtensible Stylesheet Language Transformations.) This chapter is more of a “how to get some routine tasks done” rather simply with LINQ to XML and functional construction, and the last section demonstrates literal XML with embedded LINQ. (Although the literal XML is a Visual Basic [VB] feature, you will learn how you can use literal XML and embedded LINQ in C# if you want to.)

Sometimes being surprised surprises me. For instance, it is surprising that there is a staggeringly large amount of data that is moved around using File Transfer Protocol (FTP) and comma-delimited text files (often indicated by .csv file extension where CSV means comma-separated values). On a recent project with somewhere in the neighborhood of 100 third-party providers motor vehicle data was moved around as fixed-length files over FTP. This is important data, too, such as problem drivers, driving records for truck drivers, birth and death records, Social Security numbers, address standardization, and much, much more. What’s surprising about this—fixed-length records, text, and FTP combo-platter—is that it is based on technology that is at least 20 years old. (Clearly, “services” have not ubiquitously replaced the old ways of doing things.)

The problem with all this fixed-length data and FTPing of files is that it's not very robust, it's not real time, and every programmer has to write parsing algorithms to convert this data—whether fixed-length or comma separated values—into something usable, such as objects. Worse, XML can basically represent serialized objects that can be reconstituted with a couple of lines of code (in .NET). This chapter shows you just how easy it is to convert to and from text data to objects using LINQ to XML and functional construction. (Hopefully, as a practical chapter, it will save you a lot of work.)

Constructing XML from CSV Files

A comma separated value file is a text file containing values that are separated by commas; generally, each line of text represents an individual record. You can split each value on the comma and use functional construction to convert the text file to XML. Once in XML, deserialization (or LINQ) can be used to convert the XML neatly into an object or collection of objects.

For the example, use Yahoo!'s stock-quoting capability. The quote string can be defined to return a .csv file. The following query

```
http://quote.yahoo.com/d/quotes.csv?s={0}&f=nlh
```

requests one or more quotes and returns the data in quotes.csv. The parameter s={0} contains the stocks to obtain quotes for, and the parameter f=nlh returns the name, the last price, and the intraday high. Listing 20.1 contains the complete example. The listing is followed by a decomposed explanation of the technical solution.

LISTING 20.1 Obtaining Quotes from Yahoo! in a .csv File and Using Functional Construction to Convert That Data to XML

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.IO;
using System.Xml.Linq;
using System.Net;
using System.Text.RegularExpressions;

namespace XMLFromCommaSeparatedValues
{
    class Program
    {
        static void Main(string[] args)
        {
            string all = GetQuotes("MSFT GOOG DELL");
            string[] quotes =
                all.Replace("<b>", "").Replace("</b>", "")
```

LISTING 20.1 Continued

```
.Replace("\"", "").Split(new char[]{'\n'},
StringSplitOptions.RemoveEmptyEntries);

// Read into an array of strings.
XElement stockQuotes = new XElement("Root",
    from quote in quotes
    let fields = quote
        .Split(new char[]{',', '-'},
            StringSplitOptions.RemoveEmptyEntries)
    select
        new XElement("Company", fields[0].Trim(),
        new XElement("LastPrice", fields[2].Trim(),
            new XAttribute("Time", fields[1].Trim())),
        new XElement("HighToday", fields[3].Trim())));
stockQuotes.Save("../..\\..\\quotes.xml");
Console.WriteLine(stockQuotes);
Console.ReadLine();
}

static string GetQuotes(string stocks)
{
    string url =
@"http://quote.yahoo.com/d/quotes.csv?s={0}&f=nlh";

    HttpWebRequest request =
(HttpWebRequest)HttpWebRequest.Create(string.Format(url, stocks));
HttpWebResponse response = (HttpWebResponse)request.GetResponse();

using(StreamReader reader = new StreamReader(
    response.GetResponseStream(), Encoding.ASCII))
{
    try
    {
        return reader.ReadToEnd();
    }
    finally
    {
        // don't need to close the reader because Dispose does
        response.Close();
    }
}
}

}
```

The Main function as written calls GetQuotes passing the symbols for Microsoft, Google, and Dell to GetQuotes. The result is parsed into a string array, effectively removing the HTML tags, the quotation marks, and splitting the string results by line. The remaining code uses a LINQ query that projects a new type using functional construction. A Root element is added with a nested Company element, the LastPrice, an Attribute on LastPrice, the Time, and the HighToday (see Listing 20.2 for the resulting XML). Finally, the XML is written to quotes.xml.

GetQuotes passes the formatted uniform resource locator (URL) to Yahoo! using an `HttpWebRequest`. The response is obtained from the `HttpWebRequest` as an `HttpWebResponse`, and a `StreamReader` is used to read the entire response. The response is returned as a string—`StreamReader.ReadToEnd`. Listing 20.2 shows the resulting XML after the code runs.

LISTING 20.2 The Formatted XML After the LINQ Query Uses Functional Construction to Build the XML

```
<Root>
  <Company>MICROSOFT CP
    <LastPrice Time="2:12pm">31.38</LastPrice>
    <HighToday>31.45</HighToday>
  </Company>
  <Company>GOOGLE
    <LastPrice Time="2:12pm">547.08</LastPrice>
    <HighToday>559.31</HighToday>
  </Company>
  <Company>DELL INC
    <LastPrice Time="2:12pm">19.06</LastPrice>
    <HighToday>19.20</HighToday>
  </Company>
</Root>
```

It goes without saying that the actual data will change each time you run the query. When you have the XML, the data can be used as is and formatted with XSLT, or LINQ to XML to project a new object, or deserialization to read into a class containing the desired properties.

Generating Text Files from XML

The symmetric reverse operation can be performed to convert an XML file into a text file, for example, if you are sending data to an entity that expects text .csv files. (With slight modifications, you could send fixed-length text files too by changing the formatting string in the example.)

Listing 20.3 uses LINQ to XML to read the elements of an XML document and format them as a .csv file. The decomposition follows the listing.

LISTING 20.3 Using LINQ to XML to Convert XML to a Comma-Separated Data File

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Xml.Linq;

namespace CommaSeparatedFileFromXML
{
    class Program
    {
        static void Main(string[] args)
        {
            XElement blackjackStats = XElement.Load("../..\..\CurrentStats.xml");
            string file =
                (from elem in blackjackStats.Elements("Player")
                 let statistics = elem.Element("Statistics")
                 select string.Format("{0},{1},{2},{3},{4},{5},{6},{7},{8}" +
                     "{9},{10},{11},{12},{13},{14},{15}{16}",
                     (string)elem.Attribute("Name"),
                     (string)statistics.Element("AverageAmountLost"),
                     (string)statistics.Element("AverageAmountWon"),
                     (string)statistics.Element("Blackjacks"),
                     (string)statistics.Element("Losses"),
                     (string)statistics.Element("NetAverageWinLoss"),
                     (string)statistics.Element("NetWinLoss"),
                     (string)statistics.Element("PercentageOfBlackJacks"),
                     (string)statistics.Element("PercentageOfLosses"),
                     (string)statistics.Element("PercentageOfPushes"),
                     (string)statistics.Element("PercentageOfWins"),
                     (string)statistics.Element("Pushes"),
                     (string)statistics.Element("Surrenders"),
                     (string)statistics.Element("TotalAmountLost"),
                     (string)statistics.Element("TotalAmountWon"),
                     (string)statistics.Element("Wins"),
                     Environment.NewLine)).
                    Aggregate(new StringBuilder(),
                    (builder, str) => builder.Append(str),
                    builder => builder.ToString()));

            Console.WriteLine(file);
            Console.ReadLine();
        }
    }
}
```

LISTING 20.3 Continued

```

    }
}
}
```

Using `CurrentStats.xml` from the Blackjack example in Chapter 18, “Extracting Data from XML,” you begin by loading the XML document. Next, the `from` clause defines a range value `elem`, which is each of the `Player` nodes. (There is only one in the file, but this example would work correctly if there were multiple player stats.) After the `from` clause, a temporary range value, `statistics`, is initialized with the `Player`'s `Statistics` node. The projection is the result of a string-formatting call that creates a single, comma-delimited string from each of the `statistics`' child nodes.

The string items are read variously from the attributes and elements subordinate to the `statistics` element and a new line. All of the results of the query are aggregated using the extension method `Aggregate`. The `StringBuilder` is used to accumulate the CSV lines. The first Lambda Expression accumulates each of the strings into the builder, and the second Lambda Expression (the third parameter) converts the final result into a string. The result of the `Aggregate` operation is the CSV values.

In the listing (20.3), the result is written to the console, but you could just as easily save the result to a file. Listing 20.4 contains the output from Listing 20.3.

LISTING 20.4 The Output From the LINQ to XML Statement and the Aggregate Operation

```

Player 1,-
28.125,30.6818181818183,1,8,5.9210526315789478,
➥112.5,0.04166666666666664,33.3333333333332916.666666666666664,
➥-45.83333333333329,4,1,-225,337.5,11,
```

The output is actually written as a single line for each `Player` object in the XML file but is wrapped here because of page-space limitations.

Using XML and Embedded LINQ Expressions (in VB)

The potential exists in .NET for feature envy. For example, C# has anonymous types, but VB doesn't. VB has literal XML and C# doesn't. Although it would be easier to reconcile features because the purposes of these two languages are so similar, technically, it really doesn't matter. If there is a feature in one or the other language, simply use that feature in a class library and reference the class library.

Because literal XML in VB and literal XML with embedded LINQ queries are so cool, so an example was added here. (Lobby Microsoft to add it to C#.)

The example is composed of three assemblies. One assembly contains a LINQ to SQL mapped ORM for the Northwind Customers table. This assembly is shared by the console

application and the VB assembly. The VB assembly contains literal XML and embedded LINQ. The VB assembly converts the LINQ to SQL Customer objects to XML. The third assembly, the console application, contains the DataContext, requests the Customers from the Northwind Traders database, and uses the VB assembly to easily convert the Table<Customer> collection to formatted XML.

Listing 20.5 contains the ORM mapped Customers table. This information was covered in Chapter 15, “Joining Database Tables with LINQ Queries,” so the code is shown without elaboration.

LISTING 20.5 The Customer Class Defined Using LINQ to SQL

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data.Linq;
using System.Data.Linq.Mapping;

namespace CustomerClass
{
    [Table(Name="Customers")]
    public class Customer
    {

        [Column(IsPrimaryKey=true)]
        public string CustomerID{ get; set; }

        [Column]
        public string CompanyName{ get; set; }

        [Column]
        public string ContactName{ get; set; }

        [Column]
        public string ContactTitle{ get; set; }

        [Column]
        public string Address{ get; set; }

        [Column]
        public string City{ get; set; }

        [Column]
        public string Region{ get; set; }
```

LISTING 20.5 Continued

```
[Column]
public string PostalCode{ get; set; }

[Column]
public string Country{ get; set; }

[Column]
public string Phone{ get; set; }

[Column]
public string Fax{ get; set; }
}
}
```

In Listing 20.5, the `Customer` class uses automatic properties because this table is a dumb entity—a record structure really.

Listing 20.6 contains the console application. You have seen how to define a custom `DataContext`—also in Chapter 15—so this code is also presented without elaboration.

LISTING 20.6 A C# Application That Defines the `DataContext` and Orchestrates Reading the Customers Table Using LINQ to SQL and Calling the VB Class Library's `GetXML` Method

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data.Linq;
using System.Xml.Linq;
using System.Data.Linq.Mapping;
using CustomerClass;
using Temp;

namespace LinqToSqlToXmlWithEmbeddedExpression
{
    class Program
    {
        static void Main(string[] args)
        {
            Northwind northwind = new Northwind();
            Table<Customer> customers = northwind.GetTable<Customer>();

            Console.WriteLine(LinqtoSqlToXml.GetXML(customers.ToList<Customer>()));
            Console.ReadLine();
        }
    }
}
```

LISTING 20.6 Continued

```
        }
    }

public class Northwind : DataContext
{
    private static readonly string connectionString =
        "Data Source=BUTLER;Initial Catalog=Northwind;Integrated Security=True";

    public Northwind() : base(connectionString){}
}

}
```

Finally, here is the VB code demonstrating literal XML and embedded LINQ. In the listing (20.7), an anonymous type—Dim `xmlLiteral`—is defined and assigned to literal XML. The type of `xmlLiteral` will be `XElement`. The root node is `<Customers>`. The part that looks like block script is called *embedded expression*. The embedded expressions are denoted by the `<%= %>` pairings.

Notice that the embedded expression is really a LINQ query with `From` and `Select` clauses. The `Select` clause is a projection whose result is XML. For instance, `cust` is the range variable and the attribute and element values are derived from embedded expressions that execute the statement in the script. In Listing 20.7, for example, the `select` clause produces a `<Customer>` tag for each customer, and the `<Customer>` tag has attributes `CustomerID`, `CompanyName`, and `ContactName`. The values for the attributes are defined by executing the embedded expressions, such as `cust.CustomerID`. Finally, the resultant `XElement` (XML) is returned as a string to the caller.

LISTING 20.7 Literal XML (in VB) with Embedded Expressions and LINQ

```
Imports CustomerClass
```

```
Public Class LinqtoSqlToXml
    Public Shared Function GetXML(ByVal customers As List(Of Customer)) As String
        Dim xmlLiteral = <Customers>
            <%= From cust In customers _>
            Select <Customer CustomerID=<%= cust.CustomerID %>
                    CompanyName=<%= cust.CompanyName %>
                    ContactName=<%= cust.ContactName %>>
                    <Address><%= cust.Address %></Address>
                    <City><%= cust.City %></City>
                    <State><%= cust.Region %></State>
                    <ZipCode><%= cust.PostalCode %></ZipCode>
            </Customer> %
        </Customers>
    End Function

```

LISTING 20.7 Continued

```
</Customers>
Return xmlLiteral.ToString()
End Function
End Class
```

The compiler treats each XML literal and embedded expression as a constructor call to the appropriate XML type, passing the literal or expression as an argument to the constructor. The result is an `XElement` with element and attribute child nodes.

TIP

Literal XML can span multiple lines in VB without using that annoying line continuation character _.

For more information on Literal XML in VB, see the help topic “XML Element Literal” and “Embedded Expressions in XML” in Visual Studio’s help documentation.

Summary

A real problem in the enterprise space is how to deal with legacy data in all of its various formats. The public, commercial answer is to use services—services oriented architecture (SOA) being the catchall phrase. At its essence, a service is a façade that simplifies access to things. But, what is underneath the service? The legacy data still has to be put in a form that is consumable in the first place.

XML is very consumable data. And, the easiest way to get data into an XML format is to use LINQ to XML. In this chapter, the examples were intended to help illustrate just how little code is needed if you use LINQ to XML to convert likely kinds of legacy data from comma separated values (or fixed-length records) to highly transmittable text and XML.

CHAPTER 21

Emitting XML with the `XmlWriter`

“Any sufficiently advanced technology is indistinguishable from magic.”

—Arthur C. Clarke

“Open the pod bay doors, Hal” Arthur C. Clarke is the author of, among other things, *2001: A Space Odyssey*. I remember seeing it for the first time in 1980 or 1981. At that time, I had only trifled with a TRS-80 and GW-BASIC just a little bit (in 1978).

In 1978, I recall having fun tweaking the basic tank game, which is a little bit like Space Invaders turned sideways, but not nearly as fun to play as it was to tweak. (For very realistic Space Invaders fun (see Figure 21.1), check out <http://spaceinvaders.de/>. If you don’t know what Space Invaders is, then skip the sigh of nostalgia.)

Clarke basically meant new technology seems like magic. As technologists, it is our job to look behind the curtain. That said, this chapter was debatable for this book because it is not LINQ. This chapter actually demonstrates how the `XmlWriter` works. The `XmlWriter` was introduced in .NET 2.0. However, again as practicing magicians (as opposed to being part of the enthralled audience), it is helpful for you to know how the .NET Framework is gradually layering complexity within itself to make our jobs easier.

This short chapter demonstrates how to use the `XmlWriter`. It is worth clearly noting though that the `XmlWriter` is underneath `XElement` and `XDocument` in `System.Xml.Linq`—so if you are using LINQ to XML, you might not need to use the `XmlWriter` and its descendants directly, but the `XmlWriter` is the wizard behind the curtain.

IN THIS CHAPTER

- ▶ Exploring the `XmlWriter`, Quickly
- ▶ Using `XmlTextWriter` to Write an XML File

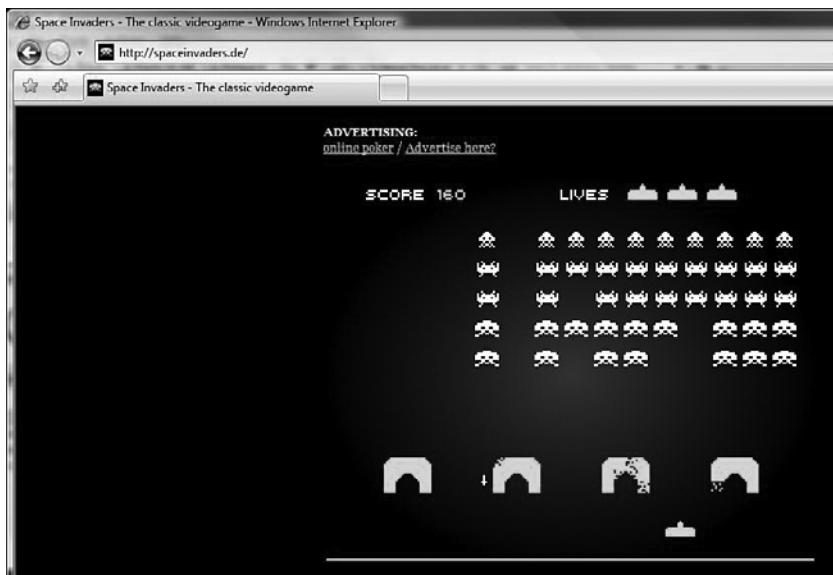


FIGURE 21.1 Tragically, a lot of seventh-grade biology was skipped to play Space Invaders, but it is a trade I am happy to recall having made.

Exploring the `XmlWriter`, Quickly

The `XmlWriter` class is an abstract class. Like an interface, this means you can declare it but you can't instantiate it. More precisely, the left side of an assignment operator can be an `XmlWriter`, but the right side of the assignment operator must be one of its concrete descendants.

The basic behavior of the `XmlWriter` is that it has methods for writing the elements of a well-formed XML document, but this class doesn't enforce a correct XML document.

TIP

An interface is a contract declaration with no behaviors. An abstract class like `XmlWriter` might specify all of the behaviors except one. Just one aspect of a class needs to be abstract for the *class* to be considered abstract, which means all of the other behaviors are well defined.

The `XmlWriter` encodes binary data as base-64 data or hexadecimal values. You can specify whether namespaces are supported, flush and close documents, determine current namespaces, and write valid names and tokens. The `XmlWriter` doesn't check for invalid elements or attributes, characters that don't match the specified encoding, or duplicate characters. That is, using a derivative of the `XmlWriter` permits you to emit a poorly formed XML file.

Among some of the elements an `XmlWriter` will support are CDATA elements, comments, processing instructions, elements, and attributes. These behaviors are named accordingly. For example, to write out a `<![CDATA[. . .]]>` block, invoke the `WriteCData` method. (CDATA is text in an XML document that is not parsed by the XML parser.)

Using `XmlTextWriter` to Write an XML File

Chapter 20, “Constructing XML from Non-XML Data,” introduced functional construction with LINQ to XML. Functional construction is nested calls to classes like `XElement`. Underneath these constructional methods are several more lines of code containing calls to `XmlTextWriter`’s methods.

Listing 21.1 demonstrates how you can take comma-separated values like those returned from the Yahoo! quotes query and write them out long to an XML file using the `XmlWriter`. The code illustrates by counterexample how much extra leverage you get out of using LINQ to XML. (In Listing 21.1, there are many more lines of code to generate an XML file than there would be using LINQ to XML, as demonstrated in Listing 20.1 in Chapter 20.)

LISTING 21.1 Producing an XML File Using the `XmlWriter` and an IO Stream

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.IO;
using System.Xml.Linq;
using System.Net;
using System.Text.RegularExpressions;
using System.Xml;

namespace WriteXmlFileWithXmlWriter
{
    class Program
    {
        static void Main(string[] args)
        {
            string all = GetQuotes("MSFT GOOG DELL");
            string[] quotes =
                all.Replace("<b>", "").Replace("</b>", "")
                    .Replace("\"", "").Split(new char[] { '\n' },
                    StringSplitOptions.RemoveEmptyEntries);

            using (XmlTextWriter writer =
                new XmlTextWriter("quotes.xml", System.Text.Encoding.UTF8))
            {

```

LISTING 21.1 Continued

```
writer.WriteStartDocument(true);
writer.WriteStartElement("Root", "");

foreach (string str in quotes)
{
    string[] fields =
        str.Split(new char[]{',', '-'}, StringSplitOptions.RemoveEmptyEntries);
    writer.WriteStartElement("Company");
    writer.WriteLine(fields[0].Trim());
    writer.WriteStartElement("LastPrice");
    writer.WriteAttributeString("Time", fields[1].Trim());
    writer.WriteLine(fields[2].Trim());
    writer.WriteEndElement();
    writer.WriteLineString("HighToday", fields[3].Trim());
    writer.WriteEndElement();
}

writer.WriteLine();
writer.WriteLine();
writer.Close();
}

static string GetQuotes(string stocks)
{
    string url =
        @"http://quote.yahoo.com/d/quotes.csv?s={0}&f=nlh";

    HttpWebRequest request =
        (HttpWebRequest)HttpWebRequest.Create(string.Format(url, stocks));
    HttpWebResponse response = (HttpWebResponse)request.GetResponse();

    using (StreamReader reader = new StreamReader(
        response.GetResponseStream(), Encoding.ASCII))
    {
        try
        {
            return reader.ReadToEnd();
        }
        finally
        {
            // don't need to close the reader because Dispose does
            response.Close();
        }
    }
}
```

If you look back at Chapter 20, you will see that the LINQ query takes care of a lot of housekeeping, including the plumbing necessary to open and close XML tags.

This gradual layering of functionality, making things a little more seamless and requiring fewer lines of code, is the hallmark of a solid framework. In some technologies, new things happen in a big bang sort of way. The .NET Framework is layering solid features on existing, solid features gradually, permitting you to do more with fewer lines of code and less housekeeping. We might call this “good eats.”

Summary

The `Xm1Writer` was introduced in the .NET Framework version 2.0. Built on top of that capability are the `XElement` and `XDocument` in the `System.Xml.Linq` namespace. `XElement` and `XDocument` layer in some additional plumbing that makes LINQ to XML (an additional layer) work. In a great framework like .NET Framework, you can achieve exceptional productivity by using the technologies like LINQ to XML or you can peel back layers, when necessary, and perform operations at a lower level of abstraction.

This page intentionally left blank

CHAPTER 22

Combining XML with Other Data Models

"I think you end up doing the stuff you were supposed to do at the time you were supposed to do it."

—Robert Downey, Jr.

In a very simple but real sense, software development is about moving data around in a digital world that exists at the atomic level and mostly in our minds. Software development is about moving data around and the rules for how, when, where, and why the data moves. That doesn't sound so hard, but, as it turns out, creating software is very hard.

Creating software is hard because software is purely an invention of minds. Almost everything that really happens—if *happens* is the right word—in software happens in the gray matter of the brain. Possibly the fact that no one is really sure how the mind works—except Stephen Pinker—explains why creating software is hard, because software is purely mind-stuff.

This chapter is about moving data from place to place and how you can do that with Language INtegrated Query (LINQ). Because this part of the book is about Extensible Markup Language (XML), this chapter shows you how to create XML from Structured Query Language (SQL) data and how to update a SQL database from XML data, all accomplished by using LINQ to XML and LINQ to SQL.

Creating XML from SQL Data

Every custom bit of software is purely a new invention in the world. Programmers are really inventors. This is true

IN THIS CHAPTER

- ▶ Creating XML from SQL Data
- ▶ Updating SQL Data from XML

because otherwise you would just grab someone else's invention and use it rather than roll your own. Invention is a messy business.

In one of my favorite movies, the character Ben Gates played by actor Nicholas Cage paraphrases Thomas Edison when referring to inventing the light bulb: *"I didn't fail. I found 2,000 ways how not to make a light bulb."* Unfortunately, in software you seldom get more than one or two tries, but you still have to find the right way. Patterns and practices and frameworks are actually instances where someone believes they have found the right way (not the *only* way but *a* right way).

This section shows you one right way to move data easily out of a SQL database and into an XML form. This is useful because SQL data doesn't travel across wide area networks well but XML, because it's self-describing and text, does travel well. Hence, the scenarios supporting the need to move between an open standard, XML, and a proprietary technology, SQL databases, will occur frequently.

To summarize, in this section, you use LINQ to SQL to get data from a SQL database and LINQ to XML to convert the data from an object-relational mapping (ORM) entity to an XML document. (You can also accomplish a similar thing by using a `DataSet` and the serialization capabilities of a `DataSet`. The difference between this technique and `DataSet` serialization is that LINQ to XML supports functional construction and allows you to reorganize the XML tree. With `DataSet` serialization, you essentially still have the same `DataSet`.)

Defining the Object-Relational Map

Chapter 13, "Querying Relational Data with LINQ," went into a lot of depth about defining object relational maps for LINQ to SQL, so that information is not repeated here. However, before you see the code for the example, let's take a moment to talk about strategy.

Having worked all over the Western Hemisphere, I have witnessed a lot of talk about standards, procedures, processes, and similar such folderol. I am not saying that this dialogue is not important—what I am saying is that on a project, where the rubber is meeting the road, is the wrong place and time to have such discussions. A key to success on an actual project is to put a specialist in each role, someone who already knows how to do the work that needs to be done. For example, when you need requirements, put someone who is very good at getting requirements in that role. Next, it is essential that that person have local authority sufficient to make a judgment call about how that role is played and the team provide the forgiveness to allow course corrections. These three aspects are paramount: highly specialized role, role authority, and acceptance that things happen *in situ* that will need to be corrected.

Specialization and local authority are important because it permits the expert—not the armchair quarterback—the latitude to do what is necessary and sufficient to achieve the end.

How this applies here is that LINQ to SQL supports complicated ORM maps that specify keys, nulls, database type maps, and additional information, but you don't always need to add all of the details to an ORM that you can. In fact, it might be counterproductive to do so because details consume time and, sometimes, because the more detail your ORM has

the more you will have to change entities if your database schema changes. In the example in Listing 22.1, you just don't need a lot of the details.

For our purposes, the code in Listing 22.1 maps the Northwind *Customers* table to an entity. The remaining subsections use that entity and LINQ to XML to convert the entity to an XML document.

NOTE

I suspect that authors (and readers) get tired of the ol' Northwind Traders database. I know I do. Sometimes I'd like to create a clever new database. However, almost everyone has Northwind or can get it and by using an existing database, it doesn't distract from the discussion at hand, in this instance using LINQ to SQL and LINQ to XML to produce an XML document from a SQL table. So, just so you know, using Northwind or Hello, World! examples is not completely institutionalized sloth<g>.

LISTING 22.1 A Very Basic ORM Mapping of the Northwind *Customers* Table, Mapped Minimally as a Matter of Discretion Based on Sufficient Need

```
[Table(Name="Customers")]
public class Customer
{
    [Column()]
    public string CustomerID{ get; set; }
    [Column()]
    public string CompanyName{ get; set; }
    [Column()]
    public string ContactName{ get; set; }
    [Column()]
    public string ContactTitle{ get; set; }
    [Column()]
    public string Address{ get; set; }
    [Column()]
    public string City{ get; set; }
    [Column()]
    public string Region{ get; set; }
    [Column()]
    public string PostalCode{ get; set; }
    [Column()]
    public string Country{ get; set; }
    [Column()]
    public string Phone{ get; set; }
    [Column()]
    public string Fax{ get; set; }
}
```

Listing 22.1 uses a bare-bones mapped entity. Notice automatic properties and empty ColumnAttributes are used. In this case, both of these choices are sufficient for the problem at hand. For example, you can use a no-parameters approach to the ColumnAttribute because the property names match the column names, and you can use automatic properties because there is no business logic. If you need business logic, automatic properties won't work.

Listing 22.2 contains the custom DataContext. In practice, the difference you might want to make is to put the connection string in the App.config file and perhaps encrypt it. At least those are the choices I would make in practice.

LISTING 22.2 A Typed DataContext Containing a Customers Property, Making It Easy to Get the Customer Data

```
public class Northwind : DataContext
{
    private static readonly string connectionString =
        "Data Source=.\SQLExpress;AttachDbFilename=" +
        "\"C:\\Books\\Sams\\LINQ\\Northwind\\northwnd.mdf\" " +
        ";Integrated Security=True;Connect Timeout=30;User Instance=True";

    public Northwind() : base(connectionString){}

    public Table<Customer> Customers
    {
        get{ return this.GetTable<Customer>(); }
    }
}
```

In your code, remember to change your connection string. Because this book was written in part on my laptop (in hotels) and in part on my workstation at home (between games of Warcraft III and Grand Theft Auto IV, near the end) different instances of the Northwind database were used.

TIP

If you have trouble figuring out how to properly define a connection string, you can use one of two techniques. One technique is to select the database in the Server Explorer, press F4, and copy the connection string from the Properties Window. The other technique is to open Windows Explorer, create a text file, change the extension to .ud1, and double-click on the file. This opens the DataLink Properties applet and lets you construct a proper connection string using a graphical user interface (GUI). (The latter technique still works nicely in Windows Vista.)

Constructing the XML Document from the SQL Data

Assuming you put the code in Listings 22.1 and 22.2 in a console application, you are now ready to use LINQ to XML and functional construction in the projection clause—the select clause with new—of the LINQ to query and generate the XML.

You learned about functional construction in Chapter 20, “Constructing XML from Non-XML Data,” so that information is not repeated here. Listing 23.3 orchestrates the two technologies—LINQ to SQL and LINQ to XML—to convert SQL data to XML data. Listings 22.1 and 22.2 support the LINQ to SQL and Listing 22.3 shows a Main function that combines the two technologies. Listing 22.4 shows the output XML document.

LISTING 22.3 A Console Application That Uses LINQ to SQL and LINQ to XML

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data.Linq;
using System.Xml.Linq;
using System.Data.Linq.Mapping;

namespace CreatingXMLFromSQLData
{
    class Program
    {
        static void Main(string[] args)
        {
            Northwind northwind = new Northwind();

            XDocument xml = new XDocument(
                new XComment("First 3 Customers Starting with 'C'"),
                new XElement("Customers",
                    (from cust in northwind.Customers
                     where cust.CompanyName.StartsWith("C"))
                     orderby cust.CompanyName
                     select new XElement("Customer",
                         new XAttribute("CustomerID", cust.CustomerID),
                         cust.CompanyName,
                         new XElement("ContactName",
                             new XAttribute("ContactTitle", cust.ContactTitle),
                             cust.ContactName),
                         new XElement("Address",
                             new XElement("Street", cust.Address),
                             new XElement("City", cust.City),
                             new XElement("Region", cust.Region),
                             new XElement("PostalCode", cust.PostalCode),
                             new XElement("Country", cust.Country),
                             new XElement("Phone", cust.Phone),
                             new XElement("Fax", cust.Fax),
                             new XElement("Email", cust.Email),
                             new XElement("Notes", cust.Notes),
                             new XElement("LastUpdate", cust.LastUpdate)
                         )
                     )
                 )
             );
        }
    }
}
```

LISTING 22.3 Continued

```

        new XElement("PostalCode", cust.PostalCode),
        new XElement("Country", cust.Country)),
        new XElement("Phones",
            new XElement("Phone",
                new XAttribute("Type", "Primary"),
                cust.Phone),
            new XElement("Phone",
                new XAttribute("Type", "Fax"),
                cust.Fax))).Take(3)));
    }

    Console.WriteLine(xml.ToString());
    Console.ReadLine();
}
}
}

```

LISTING 22.4 The Generated XML Document from the Code in Listings 22.1, 22.2, and 22.3

```

<!—First 3 Customers Starting with 'C' —>
<Customers>
    <Customer CustomerID="CACTU">
        Cactus Comidas para llevar
        <ContactName ContactTitle="Sales Agent">Patricio Simpson</ContactName>
        <Address>
            <Street>Cerrito 333</Street>
            <City>Buenos Aires</City>
            <Region></Region>
            <PostalCode>1010</PostalCode>
            <Country>Argentina</Country>
        </Address>
        <Phones>
            <Phone Type="Primary">(1) 135-5555</Phone>
            <Phone Type="Fax">(1) 135-4892</Phone>
        </Phones>
    </Customer>
    <Customer CustomerID="CENTC">Centro comercial Moctezuma
        <ContactName ContactTitle="Marketing Manager">Francisco Chang</ContactName>
        <Address>
            <Street>Sierras de Granada 9993</Street>
            <City>México D.F.</City>
            <Region></Region>
            <PostalCode>05022</PostalCode>
            <Country>Mexico</Country>
        </Address>

```

LISTING 22.4 Continued

```
<Phones>
  <Phone Type="Primary">(5) 555-3392</Phone>
  <Phone Type="Fax">(5) 555-7293</Phone>
</Phones>
</Customer>
<Customer CustomerID="CHOPS">
  Chop-suey Chinese
  <ContactName ContactTitle="Owner">Yang Wang</ContactName>
  <Address>
    <Street>Hauptstr. 29</Street>
    <City>Bern</City>
    <Region></Region>
    <PostalCode>3012</PostalCode>
    <Country>Switzerland</Country>
  </Address>
  <Phones>
    <Phone Type="Primary">0452-076545</Phone>
    <Phone Type="Fax"></Phone>
  </Phones>
</Customer>
</Customers>
```

Notice three things about Listing 22.3. The first is that the projection clause changed the organization of the `Customer` object for the XML document. The second is that an aggregate method—`Take`—was used to limit the size of the resultset, and the third is that the `XComment` node type was introduced.

Using the `XComment` Node Type

LINQ to XML classes like `XComment` inherit from `XNode` or `XContainer`. Similar to classes in the `CodeDOM` namespace, classes like `XDocument`, `XElement`, `XComment`, and `XCData` represent objects in a tree graph that will each emit the corresponding element. `XComment` is used to write properly formatted comments to an XML document.

Displaying the XML Document in a TreeView

There are practical uses for SQL to XML to SQL scenarios. A neat one that I read about on the Internet was to support bulk loading of Active Directory from an XML file using LINQ to XML. If you combine what you learned about implementing an `IQueryProvider` for Active Directory in Chapter 12, “Querying Outlook and Active Directory,” with the LINQ to XML information in this part of the book, it shouldn’t be much of a stretch to implement such a bulk loader.

The scenario in this section simply gets SQL from a database, converts it to a flat XML tree, and uses that tree to create a simple web page with a `TreeView` of the selected customers. Listing 22.5 depends on the same `Customer` entity class and `Northwind`

`DataContext` from Listings 22.1 and 22.2 to get the SQL data. Listing 22.5 is a new method, `GetData`. `GetData` can be used to replace the LINQ code in Listing 22.3. `GetData` yields a flatter construction of the XML document than the LINQ query in the `Main` function in Listing 22.3.

Listing 22.5 shows the implementation of the `GetData` method with a query that defines an XML document with `XElements` for each column in the `Customers` table.

LISTING 22.5 `GetData` Can Be Placed in a Separate Library with Listings 22.1 and 22.2; This Listing Returns an `XDocument` Object Representing the Constructed XML Document

```
// simpler xml function construction
public static XDocument GetData()
{
    Northwind northwind = new Northwind();

    XDocument xml = new XDocument(
        new XComment("First 3 Customers Starting with 'C'"),
        new XElement("Customers",
            (from cust in northwind.Customers
             where cust.CompanyName.StartsWith("C")
             orderby cust.CompanyName
             select new XElement("Customer",
                 new XElement("CustomerID", cust.CustomerID),
                 new XElement("CompanyName", cust.CompanyName),
                 new XElement("ContactTitle", cust.ContactTitle),
                 new XElement("ContactName", cust.ContactName),
                 new XElement("Address", cust.Address),
                 new XElement("City", cust.City),
                 new XElement("Region", cust.Region),
                 new XElement("PostalCode", cust.PostalCode),
                 new XElement("Country", cust.Country),
                 new XElement("Phone", cust.Phone),
                 new XElement("Fax", cust.Fax))).Take(3)));

    return xml;
}
```

To create the web page, simply create an ASP.NET website in Microsoft Visual Studio. Place a `TreeView` on the page and bind the `XDocument` as a string—call `ToString`—to an `XmlDataSource`. Use the `XmlDataSource` as the `DataSource` for the `TreeView`. You will need to map the `TreeNodeBinding`'s `DataMember` and `TextField` properties for each of the nodes that you want to see in the tree. The `foreach` statement shows you how to do that (see Listing 22.6). (Figure 22.1 shows the very basic output from Listing 22.6.)

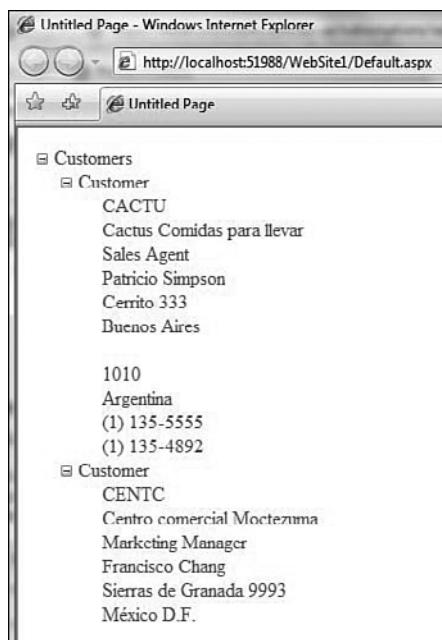


FIGURE 22.1 The very simple TreeView output from the XML document.

LISTING 22.6 Showing the Code-Behind for the Web Page

```
using System;
using System.Configuration;
using System.Data;
using System.Linq;
using System.Web;
using System.Web.Security;
using System.Web.UI;
using System.Web.UI.HtmlControls;
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;
using System.Xml.Linq;
using System.Xml.XPath;

public partial class _Default : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        if (!IsPostBack)
```

LISTING 22.6 Continued

```
{  
    XDocument doc = XmlData.GetData();  
    XmlDataSource1.Data = doc.ToString();  
  
    foreach ( XElement elem in doc.Element("Customers").Elements("Customer") .  
        Elements() )  
    {  
        TreeNodeBinding binding = new TreeNodeBinding();  
        binding.DataMember = elem.Name.ToString();  
        binding.TextField = "#InnerText";  
        TreeView1.DataBindings.Add(binding);  
    }  
  
    TreeView1.DataSource = XmlDataSource1;  
    TreeView1.DataBind();  
}  
}
```

Direct support for binding right to WinForm DataGridViews and TreeViews is iffy. It would be nice if the `XDocument` or `XElement` could be bound to these controls without a lot of code, but a thorough search of the web makes it seem as if this problem hasn't been completely resolved. Some legwork is required, but Chapter 23, "LINQ to XSD Supports Typed XML Programming,"—the last chapter—explores LINQ to XSD and how you can use schemas to help with some of the extra work involved.

NOTE

If you bind XML right to the `DataSource` of a `DataGridView`, the result is the schema of the XML document by default, which includes not only the data, but XML concepts too. This is probably not what you want.

Updating SQL Data from XML

Now let's go in the other direction. Suppose you have some XML data and you want to send it to your database. To do this, you use the LINQ technologies in reverse: Use LINQ to XML to walk the XML nodes and LINQ to SQL to update the database.

Listing 22.7 offers a sample XML file that contains a reference to Okemos' eclectic international tuba restaurant, which is a lot of fun if you are in touch with your inner hipster and the food is good, and one of Okemos' first-class dining spots, Dusty's Cellar.

LISTING 22.7 A Sample XML File Containing Some New Customers for the Northwind Traders Database

```
<?xml version="1.0" encoding="utf-8" ?>
<Customers>
    <Customer>
        <CustomerID>TRAVL</CustomerID>
        <CompanyName>Travelers Club Rest. and Tuba Museum</CompanyName>
        <ContactTitle></ContactTitle>
        <ContactName>William White</ContactName>
        <Address>2138 Hamilton</Address>
        <City>Okemos</City>
        <Region>MI</Region>
        <PostalCode>48864</PostalCode>
        <Country>USA</Country>
        <Phone>517-349-1701</Phone>
        <Fax></Fax>
    </Customer>
    <Customer>
        <CustomerID>DUSTY</CustomerID>
        <CompanyName>Dusty's Cellar</CompanyName>
        <ContactTitle></ContactTitle>
        <ContactName>Matt Rhodes</ContactName>
        <Address>1839 Grand River Avenue</Address>
        <City>Okemos</City>
        <Region>MI</Region>
        <PostalCode>48864</PostalCode>
        <Country>USA</Country>
        <Phone>1 517 349-5150</Phone>
        <Fax>1 517 349-8416</Fax>
    </Customer>
</Customers>
```

In this scenario, both restaurants have earned a place in the Northwind database. To get the customers in, you can use the Customer ORM from Listing 22.1 and the `DataContext` from Listing 22.2. In Listing 22.7, the `XElement` class loads the XML document (Listing 22.6). A `Customer` object is instantiated and a `foreach` loop walks the elements initializing the properties of the `Customer` object. At the end of the loop, LINQ to SQL is used to insert the new object and submit the changes.

NOTE

The only change to the *Customer* class was to add the *IsPrimaryKey* argument to the *CustomerID* property's *ColumnAttribute*. A primary key is needed for modifications to the database. Here is the *CustomerID* property changed to support the insert in Listing 22.8:

```
[Column(IsPrimaryKey=true)]
public string CustomerID { get; set; }
```

LISTING 22.8 Loading the XML Document, Walking the Nodes with a foreach Loop Updating the Entity Object, Customer, and Inserting the Customer Using the DataContext and LINQ to SQL

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

namespace UpdateSQLFromXML
{
    class Program
    {
        static void Main(string[] args)
        {
            XElement xml = XElement.Load("../..\..\DataToUpdate.xml");
            Northwind northwind = new Northwind();

            foreach (XElement elem in xml.Elements("Customer"))
            {
                Customer cust = new Customer();
                cust.CustomerID = elem.Element("CustomerID").Value;
                cust.CompanyName = elem.Element("CompanyName").Value;
                cust.ContactTitle = elem.Element("ContactTitle").Value;
                cust.ContactName = elem.Element("ContactName").Value;
                cust.Address = elem.Element("Address").Value;
                cust.City = elem.Element("City").Value;
                cust.Region = elem.Element("Region").Value;
                cust.PostalCode = elem.Element("PostalCode").Value;
                cust.Country = elem.Element("Country").Value;
                cust.Phone = elem.Element("Phone").Value;
                cust.Fax = elem.Element("Fax").Value;
            }
        }
    }
}
```

LISTING 22.8 Continued

```

    northwind.Customers.InsertOnSubmit(cust);
    northwind.SubmitChanges();

}
}

```

To verify that the data was inserted, you can execute the following SQL statement in Visual Studio (see Listing 22.9). To execute a SQL statement in Visual Studio, follow these steps:

1. Expand the Server Explorer.
2. Expand the Data Connections node, Database node, and Tables node each in turn. (In this example, the Database node will be the Northwind data connection.)
3. Right-click on the Customers table and select New Query.
4. In the Add Table dialog box, add the Customers table and close the dialog box.
5. Enter the SQL text in the SQL pane.
6. Click the Execute button (the exclamation icon) or press Ctrl+R.

You also have the option of using the designer to builder the query, using SQL Server Management tools, or the osql.exe or sqlcmd.exe command-line tools. Figure 22.2 shows the output from the query in Listing 22.9.

CustomerID	CompanyName	ContactName	ContactTitle	Address	City	Region	PostalCode	Country	Phone	Fax
DUSTY	Dusty's Cellar	Matt Rhodes	1839 Grand River	Oklahoma City	OK	48064	USA	1 517 349-5150	1 517 349-8416	
TRAVL	Travelers Club ...	William White	2138 Hamilton	Oklahoma City	OK	48064	USA	517-349-1701		
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

FIGURE 22.2 The results from the SQL query in Listing 22.9, after you have run the insertion code from Listing 22.8.

LISTING 22.9 A SQL Query to Examine the Inserted Data from Listing 22.8

```

SELECT
    CustomerID, CompanyName, ContactName, ContactTitle, Address, City,
    Region, PostalCode, Country, Phone, Fax
FROM
    Customers
WHERE
    (CustomerID = 'DUSTY') OR (CustomerID = 'TRAVL')

```

To use the osql.exe command line, follow these steps:

1. Make sure that Authenticated Users have full access to the drive containing the Northwind database. (Right-click on the folder, select Properties, select the Security tab, select Authenticated Users, and then select Edit. Change control to Full Control; see Figure 22.3.)



FIGURE 22.3 Modified Northwind properties folder permission settings, permitting osql.exe command-line interaction.

2. At the command prompt, type **osql -E -S .\SQLEXPRESS** and press Enter.
3. At the 1> prompt, type **sp_attach_db "northwind", "[path]\Northwind.mdf"** <enter>.
4. Type **go <enter>**.
5. Type **use northwind <enter>**.
6. Type **go <enter>**.
7. Type **SELECT * FROM Customers WHERE CustomerID = 'DUSTY' OR CustomerID = 'TRAVL'** <enter>.
8. Type **go <enter>**.

After the last steps, you should see the output from the database showing you the two inserted records. If you want to generate a script with inputs and outputs, change step 2 to include the **-o [outputfile] -e** (echo command) as follows:

```
osql -E -S .\SQLEXPRESS -o script.txt -e
```

And type **quit** after the last command. Listing 22.10 shows the output from the script file (named **script.txt**).

LISTING 22.10 The Output from the osql Scripting Session

Summary

As the last chapter of this book approaches, you will forgive me if I get a bit sentimental. A book like this takes me (now at 42) several months and a thousand hours or more to research and write. Because I am sentimental, I throw in things that you must know like LINQ to SQL to LINQ to XML and things I think you'd like to know like Universal Data Links (.ud1 file) tips and osql.exe command-line sessions, along with the movie references and analogies. These exist in the text in an attempt to make the journey both informative and enjoyable. That is, I hope you will read the book rather than Google through it.

After 20 years of experience, I occasionally still am subjected to technical interviews, nuts-and-bolts stuff, for development work. Perhaps this happens because some (a few) people think that writers write and don't do. In my case, my time is spent writing, coding, or modeling 40 to 60 hours per week developing software—up until two years ago closer to 80—and as many as 40 more on writing books like this one.

My books are written for two audiences. The first one is for me. There is no way one person can have 100% recall on all of this material at all times. For example, I had to look up the `sp_attach_db` stored procedure, but I remember that the `IDTExtensibility2` interface is for writing Visual Studio wizards. And, the second audience is you. From the many hundreds of encouraging emails and sometimes gifts—\$100 in Denny’s coupons, smoked fish from Norway, and a free dinner in St. Louis—it’s obvious that we are in this together.

Perhaps given a photographic memory, there would be less need to spend weekends, holidays, and family gatherings writing. But, I have to keep learning too, and writing is a constructive way to do it. So, in part, selfishly this—my umpteenth book—is sort of like an extension of or overflow from my brain, like Dumbledore’s pen sieve, and in part I hope you will actually enjoy the journey with me.

So to borrow from my friend Jackson Wayfare, “brevity does not mean inconsequence,” but a whole lot of palaver won’t get us to the end of this book.

CHAPTER 23

LINQ to XSD Supports Typed XML Programming

“Necessity is the mother of invention, it is true, but its father is creativity, and knowledge is the midwife.”

—Jonathan Schattke

When the CodeDOM was invented, it was mentioned in several articles that the CodeDOM would end up being an important part of the .NET Framework that would let developers create new and imaginary things. (If you don't know, the CodeDOM supports code-generating source code and emitting running binaries.) And, the CodeDOM has been quietly playing an important role in the background since .NET's inception and it plays a role here, too.

LINQ to XSD—also referred to as LINQ to XML for Objects—is typed programming support for LINQ to XML. LINQ to XML is based on the `XDocument` and `XElement` classes and supports LINQ over XML queries. LINQ to XSD is an extension to LINQ to XML that permits you to program with LINQ to XML but explicitly leave out the calls to the `XDocument` and `XElement` methods. For example, LINQ to XML support queries like this:

```
(from item in purchaseOrder.Elements("Item")
    select (double)item.Element("Price")
        * (int)item.Element("Quantity")
    ).Sum();
```

LINQ to XSD supports querying the same XML document but with a more natural, IntelliSense supported and object-oriented style of writing the LINQ queries. With LINQ to XSD, the preceding query can be written as follows:

IN THIS CHAPTER

- ▶ Understanding the Basic Design Goals of LINQ to XSD
- ▶ Programming with LINQ to XSD

```
(from item in purchaseOrder.Item
 select item.Price * item.Quantity
).Sum();
```

Notice that the typecasting is gone, as well as the explicit method calls to methods in the System.Xml.Linq namespace, like Elements.

LINQ to XSD uses XDocument as its backing store instead of the XSD.exe utility's use of XmlDocument. Thus, the calls are made, but the LINQ to XML plumbing is code generated for you into typed, wrapper objects that handle the LINQ to XML plumbing calls.

It is important to note that this technology is early development and likely to change, but this chapter shows you a little about the basic design goals and walks through an example based on the Alpha 0.2 release from February 2008.

Understanding the Basic Design Goals of LINQ to XSD

A popular concept (apparently at Microsoft at least) right now is the concept of mitigating impedance mismatches. You can think of impedance mismatches as productivity speed bumps. For example, when twisting database tables into objects, programmers are impeded because the shape of normalized tables and object hierarchies seldom match. The impedance mismatch for XML to objects is getting the data from XML text documents into objects and doing something with the data in an object-oriented way. For example, in LINQ to XML, everything is a string, so you have to typecast. Nodes have to be requested by chaining method calls. All this plumbing impedes forward motion—a speed bump.

The basic design objective of LINQ to XSD (or LINQ to XML for objects) is to hide the XML plumbing, so the developer can focus on desired outcomes, for example, summing the cost of an order.

Some basic observations about LINQ to SQL (paraphrasing the present literature on the subject) are that the shape of the untyped code is preserved in the typed code. In the fragment in the chapter opener, you see that the general structure of the LINQ to XML query didn't really change that much in the LINQ to XSD version. Untyped, string-encoded access is eliminated with LINQ to XSD, removing the need for typecasting everything that is not to be treated as a string. In addition, XML namespaces are replaced by proper common Language Runtime (CLR) namespaces. This means you don't have to concatenate string namespaces, but rather you use instance-oriented mapping; that is, LINQ to XSD generated types have namespaces and classes and the member-of dot-operator (.) resolves these references.

The general design goals are to cover all of XML Schema. Create mappings that are predictable and comprehensible, facilitate round-tripping, avoid relying on customizations, convey schema intent into the object models as much as possible, and derive classes that are as close to the expectations of the programmer as possible.

In an alpha product, objectives are likely to change and some of the objectives might not be complete yet, but the general idea of making LINQ to XML more natural by eliminating the XML plumbing is probably a safe bet.

Programming with LINQ to XSD

You can try the samples in the LINQ to XSD download, and a new one was created for this book. The example in this section uses a list of luncheon menu items from Dusty's Cellar in Okemos and, using LINQ to XSD, converts the XML to typed objects and a LINQ query is written against the code-generated objects and the underlying XML file.

From 1,000-foot view, the following are the steps for using LINQ to XSD:

1. Download and install the LINQ to XSD preview.
2. Create a LINQ to XSD Preview Console Application.
3. Define the content of an XML file.
4. Define the content of an XML Schema file.
5. Compile the project to generate the typed XML wrapper classes.
6. Query against the generated code with LINQ.

Downloading and Installing the LINQ to XSD Preview

You can stay on top of the latest developments for LINQ to XSD at Microsoft's XML Team WebLog at <http://blogs.msdn.com/xmlteam/archive/2008/02/21/linq-to-xsd-alpha-0-2.aspx>. To download and install the Alpha 0.2 preview used for the demo in this chapter, you can go to this link <http://www.microsoft.com/downloads/details.aspx?FamilyID=a45f58cd-fcfc-439e-b735-8182775560af&displaylang=en>, Google for LINQ to XSD Preview, or follow the link at the XML Team's WebLog. (Basically, www.microsoft.com/downloads/ will get you most of the way there for almost any Microsoft product.)

The installation process is a standard install process. Like most of the Microsoft Visual Studio add-ins, the LINQ to XSD installation adds project templates to the Visual Studio folder, and these elements show up in the Add New Item and Projects dialog boxes as selectable templates. (For more information on building templates, check out my article "Creating Project Templates in .NET" at <http://www.informit.com>.)

Creating a LINQ to XSD Preview Console Application

JavaScript files, a common Wizard DLL, wizard (*.vsz), and Visual Studio directory (*.vsdir) files all work together to support all of the various project templates. Unless you are writing custom wizards, you can let Visual Studio manage all this plumbing and infrastructure.

From an examination of a LINQ to XSD project, it looks like all the new template does is add a reference to the `Microsoft.Xml.Schema.Linq.dll` assembly, but in general, it seems to work best to let the various wizards create project templates for you. To create a LINQ to XSD Preview project, follow these steps:

1. Open Visual Studio (after installing the LINQ to XSD Preview).
2. Select File, New Project, LINQ to XSD Preview, LINQ to XSD Console Application to create a LINQ to XSD console project.

The net result is that you will have a basic console application with a reference to the LINQ to XSD assembly.

Defining the XML Context

The next step is to define some XML content. Spending a fair amount of my meager disposable income on cigars and tasty tidbits from Dusty's Cellar in Okemos, the sample XML in this section contains the luncheon menu for the restaurant.

Select Project, Add New Item and add an XML file to the console application project. Figure 23.1 shows the Add New Item dialog box with the XML File template selected and Listing 23.1 shows the XML file created from the menu items.

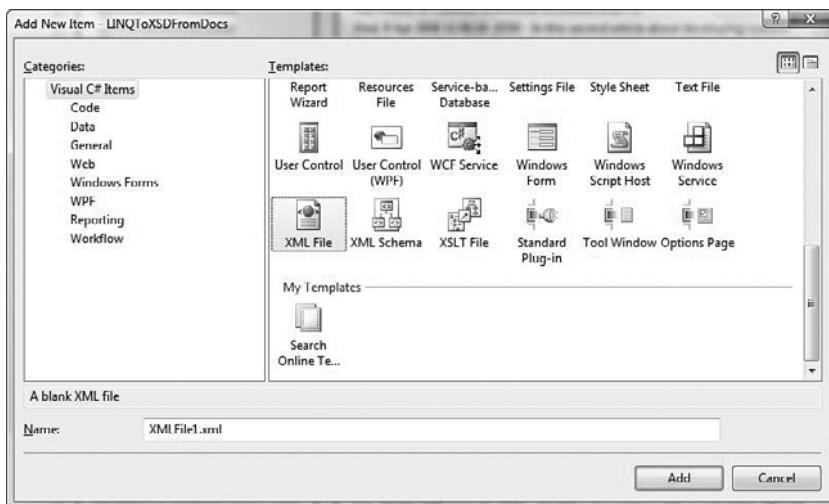


FIGURE 23.1 The Add New Item template dialog box is your friend.

LISTING 23.1 A Sample XML File Describing a Luncheon Menu

```
<?xml version="1.0" encoding="utf-8" ?>
<Dustys xmlns="http://www.dustyscellar.com/TuscanLuncheon">
  <Luncheon>
    <LuncheonId>L1</LuncheonId>
    <Price>12.00</Price>
    <LimitedByDayOfWeek></LimitedByDayOfWeek>
    <Item>
      <Name>Chicken Caesar Salad</Name>
      <Description>Tossed in mixed greens with Key Lime Caesar dressing</Description>
    </Item>
    <Item>
      <Name>Salmon Salad</Name>
      <Description>With assorted vegetables and Balsamic Vinaigrette</Description>
```

LISTING 23.1 Continued

```
</Item>
<Item>
    <Name>Torta Rustica</Name>
    <Description> Our signature dish made with brioche
    ↪ dough filled with smoked turkey, red peppers, provolone,
    ↪ Gournay cheese and artichoke hearts</Description>
    </Item>
</Luncheon>
<Luncheon>
    <LuncheonId>L2</LuncheonId>
    <Price>21.00</Price>
    <LimitedByDayOfWeek></LimitedByDayOfWeek>
    <Item>
        <Name>Chicken Breast</Name>
        <Description>Stuffed with herb garlic cheese,
    ↪ sun dried tomatoes and spinach</Description>
    </Item>
    <Item>
        <Name>Grilled 8oz Sirloin Steak</Name>
        <Description>With Pinoit Noir demi glaze</Description>
    </Item>
    <Item>
        <Name>Spinach Linguine</Name>
        <Description>With Tarragon Marsala cream, bacon,
    ↪ wild mushrooms, shallots and Romano cheese</Description>
    </Item>
</Luncheon>
<Luncheon>
    <Luncheon>F</Luncheon>
    <Price>12.00</Price>
    <LimitedByDayOfWeek>Sunday</LimitedByDayOfWeek>
    <Item>
        <Name>French Toast</Name>
        <Description>Stuffed with seasonal berries</Description>
    </Item>
    <Item>
        <Name>Scrambled Eggs</Name>
        <Description>With bacon or sausage</Description>
    </Item>
    <Item>
        <Name>Spinach Crepe</Name>
        <Description></Description>
    </Item>
    <Item>
        <Name>Torta Rustica</Name>
```

LISTING 23.1 Continued

```
<Description> Our signature dish made with brioche  
➥ dough filled with smoked turkey, red peppers, provolone,  
➥ Gournay cheese and artichoke hearts</Description>  
</Item>  
</Luncheon>  
</Dustys>
```

Any XML file will do. The key to using LINQ to XSD is to define an XML Schema file, its contents, and set the build option to use the `LinqToXSD.exe` compiler that is downloaded with the LINQ to XSD Preview package.

Defining the XML Schema File

Again, return to the Add New Item dialog box and add an XML Schema file to your project. An XML file defines the data in a self-describing way. The XML Schema defines the schema or structure of the XML file. The separation of these two items supports changing the structure of the XML data by applying different schemas.

XML Schema files are generally a high level of abstraction that describes what XML documents conforming to a particular schema should contain. Schema documents support adding additional information above and beyond what an XML document itself will support, like adding regular expressions. Listing 23.2 contains a schema file for our example.

LISTING 23.2 An XML Schema File (.xsd) for the Tuscan Luncheon Menu Data in the XML File in Listing 23.1

```
<xss:schema  
  xmlns:xss="http://www.w3.org/2001/XMLSchema"  
  targetNamespace="http://www.dustyscellar.com/TuscanLuncheon"  
  xmlns="http://www.dustyscellar.com/TuscanLuncheon"  
  elementFormDefault="qualified">  
  
<xss:element name="Dustys">  
  <xss:complexType>  
    <xss:sequence>  
      <xss:element ref="Luncheon"  
                  minOccurs="0" maxOccurs="unbounded" />  
    </xss:sequence>  
  </xss:complexType>  
</xss:element>  
  
<xss:element name="Luncheon">  
  <xss:complexType>  
    <xss:sequence>
```

LISTING 23.2 Continued

```
<xs:element name="LuncheonId" type="xs:double"/>
<xs:element name="Price" type="xs:double"/>
<xs:element name="LimitedByDayOfWeek" type="xs:string"/>
<xs:element ref="Item"
    minOccurs="0" maxOccurs="unbounded"/>
</xs:sequence>
</xs:complexType>
</xs:element>

<xs:element name="Item">
<xs:complexType>
<xs:sequence>
<xs:element name="Name" type="xs:string"/>
<xs:element name="Description" type="xs:string"/>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:schema>
```

The XSD Schema file indicates that documents that support this schema contain an element `Dustys`. `Dustys` contains a list of luncheon items, and the luncheon items contain a `LuncheonId`, `Price`, `LimitedByDayOfWeek`, and an `Item` element. The schema also defines the data types for these elements. `Item` is defined as containing a `Name` and `Description`.

Adding a Regular Expression to a Schema File

If you wanted to further constrain XML files to those that match the schema, you could add a regular expression that enforces values for the `LimitedByDayOfWeek` element. As defined in Listing 23.2, any string could be placed in this element, but what you really want are valid days of the week or perhaps an empty string (refer to Listing 23.3).

LISTING 23.3 Defining a Regular Expression Element That Scans the Day of Week Element for Acceptable Values

```
<xs:schema
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.dustyscellar.com/TuscanLuncheon"
  xmlns="http://www.dustyscellar.com/TuscanLuncheon"
  elementFormDefault="qualified">

<xs:element name="Dustys">
<xs:complexType>
<xs:sequence>
<xs:element ref="Luncheon"
```

LISTING 23.3 Continued

```

        minOccurs="0" maxOccurs="unbounded" />
    </xs:sequence>
</xs:complexType>
</xs:element>
<xs:element name="Luncheon">
<xs:complexType>
<xs:sequence>
<xs:element name="LuncheonId" type="xs:string"/>
<xs:element name="Price" type="xs:double"/>
<xs:element name="LimitedByDays">
<xs:complexType>
<xs:sequence minOccurs="0" maxOccurs="unbounded">
<xs:element name="LimitedByDayOfWeek" type="acceptable-weekdays" />
</xs:sequence>
</xs:complexType>
</xs:element>

<xs:element ref="Item"
            minOccurs="0" maxOccurs="unbounded" />
</xs:sequence>
</xs:complexType>
</xs:element>
<xs:simpleType name="acceptable-weekdays">
<xs:restriction base="xs:string">
<xs:pattern value="(Sunday|Monday|Tuesday|Wednesday|Thursday|Friday|
➥ Saturday)"/>
</xs:restriction>
</xs:simpleType>

<xs:element name="Item">
<xs:complexType>
<xs:sequence>
<xs:element name="Name" type="xs:string"/>
<xs:element name="Description" type="xs:string"/>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:schema>
```

Because the new schema in Listing 23.3 indicates that there can now be multiple days that a luncheon is limited to, you will need to change the XML document. Listing 23.4 contains a revised XML document that complies with the schema in Listing 23.3.

LISTING 23.4 If You Use the XML Schema in Listing 23.3, Modify the XML Document to Conform to the List of Possible Weekdays

23

```
<?xml version="1.0" encoding="utf-8" ?>
<Dustys xmlns="http://www.dustyscellar.com/TuscanLuncheon">
  <Luncheon>
    <LuncheonId>L1</LuncheonId>
    <Price>12.00</Price>
    <LimitedByDays>
      </LimitedByDays>
    <Item>
      <Name>Chicken Caesar Salad</Name>
      <Description>Tossed in mixed greens with Key Lime Caesar dressing</Description>
    </Item>
    <Item>
      <Name>Salmon Salad</Name>
      <Description>With assorted vegetables and Balsamic Vinaigrette</Description>
    </Item>
    <Item>
      <Name>Torta Rustica</Name>
      <Description> Our signature dish made with brioche
      ↳ dough filled with smoked turkey, red peppers, provolone,
      ↳ Gournay cheese and artichoke hearts</Description>
    </Item>
  </Luncheon>
  <Luncheon>
    <LuncheonId>L2</LuncheonId>
    <Price>21.00</Price>
    <LimitedByDays></LimitedByDays>
    <Item>
      <Name>Chicken Breast</Name>
      <Description>Stuffed with herb garlic cheese,
      ↳ sun dried tomatoes and spinach</Description>
    </Item>
    <Item>
      <Name>Grilled 8oz Sirloin Steak</Name>
      <Description>With Pinoit Noir demi glaze</Description>
    </Item>
    <Item>
      <Name>Spinach Linguine</Name>
      <Description>With Tarragon Marsala cream, bacon,
      ↳ wild mushrooms, shallots and Romano cheese</Description>
    </Item>
  </Luncheon>
```

LISTING 23.4 Continued

```
<Luncheon>
  <LuncheonId>F</LuncheonId>
  <Price>12.00</Price>
  <LimitedByDays>
    <LimitedByDayOfWeek>Saturday</LimitedByDayOfWeek>
    <LimitedByDayOfWeek>Sunday</LimitedByDayOfWeek>
  </LimitedByDays>
  <Item>
    <Name>French Toast</Name>
    <Description>Stuffed with seasonal berries</Description>
  </Item>
  <Item>
    <Name>Scrambled Eggs</Name>
    <Description>With bacon or sausage</Description>
  </Item>
  <Item>
    <Name>Spinach Crepe</Name>
    <Description></Description>
  </Item>
  <Item>
    <Name>Torta Rustica</Name>
    <Description> Our signature dish made with brioche
    ↵ dough filled with smoked turkey, red peppers, provolone,
    ↵ Gournay cheese and artichoke hearts</Description>
  </Item>
</Luncheon>
</Dustys>
```

Notice that Listing 23.4 now has a `LimitedByDays` element and this element contains the child elements `LimitedByDayOfWeek`.

After you have chosen the form of your XML and schema documents, you set the `BuildAction` of the schema document to `LinqToXsdSchema` and compile the solution. To set the build action, complete the following steps:

1. Open the Solution Explorer.
2. Right-click the XSD document and select Properties.
3. Change the schema document's Build Action to `LinqToXsdSchema` (see Figure 23.2).
4. Compile the solution.

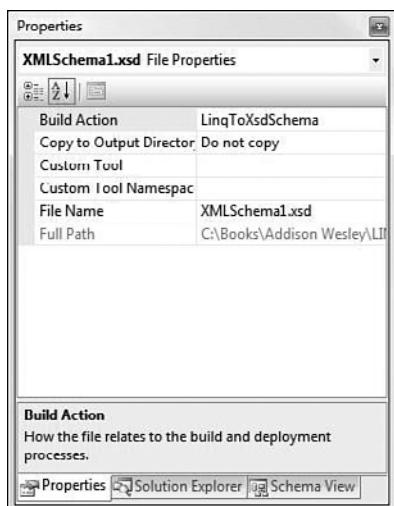


FIGURE 23.2 Remember to set the `BuildAction` property of the schema document to `LinqToXsdSchema`.

When you build the solution, the `LinqToXsdSchema` tool generates the XML wrapper classes, including the namespace. Figure 23.3 shows the Object Browser (F2) with the generated elements for the original XML and schema documents (from Listing 23.1 and 23.2).

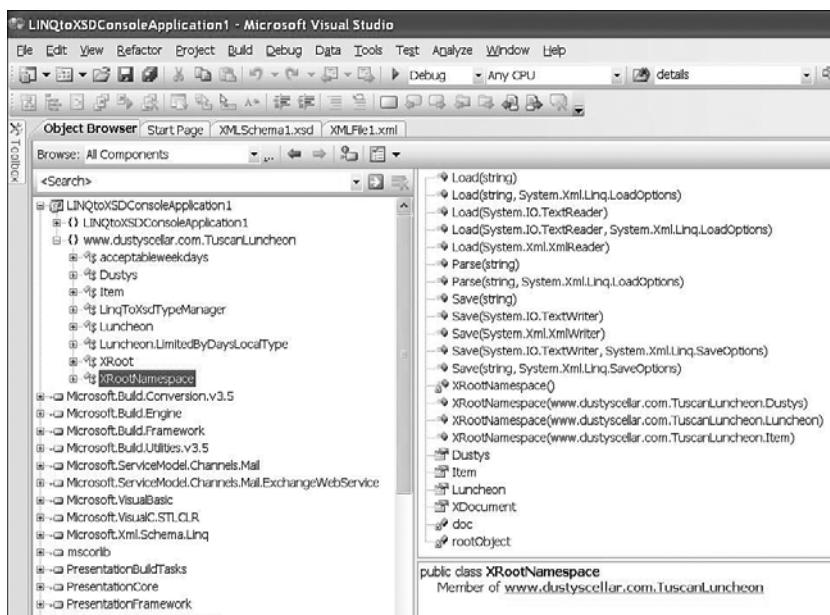


FIGURE 23.3 The generated elements from the Object Browser for the XML and schema from Listings 23.1 and 23.2.

Querying with LINQ to XML for Objects

After you have finished the modest amount of preparation work, you can write queries against the XML document using all of the features in regular LINQ to Objects.

IntelliSense works (see Figure 23.4) as does the features discussed in Part II of this book.

Listing 23.5 demonstrates a query based on the XML and schema in Listings 23.1 and 23.2, and Listing 23.6 shows a revised query using the XML and schema from Listings 23.3 and 23.4, incorporating the regular expression element.

LISTING 23.5 A LINQ Query That Hits the Wrapper Classes and XML Document for the XML and Schema from Listings 23.1 and 23.2

```
using System;
using System.Linq;
using System.Collections.Generic;
using System.Text;
using System.Xml;
using System.Xml.Linq;
using www.dustyscellar.com.TuscanLuncheon;

namespace DustysTuscanLuncheonMenu
{
    class Program
    {
        static void Main(string[] args)
        {
            var dustys = Dustys.Load("../Menu.xml");
            var menu = from lunchMenu in dustys.Luncheon
                       from item in lunchMenu.Item
                       let cost = lunchMenu.Price * 8 * 1.06
                       where lunchMenu.LimitedByDayOfWeek == "Sunday"
                       select new {MenuItem=item.Name,
                                  Description=item.Description,
                                  Cost=string.Format("{0:C}", cost)};

            Array.ForEach(menu.ToArray(), m=>Console.WriteLine(m));
            Console.ReadLine();
        }
    }
}
```

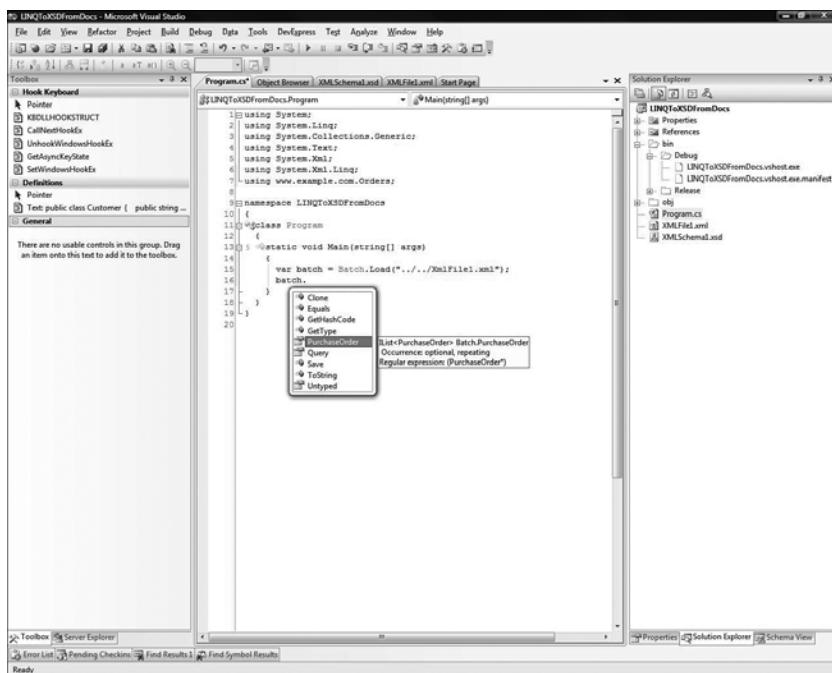


FIGURE 23.4 IntelliSense picks up the namespaces, classes, and members generated in the wrapper class for LINQ to XSD (also called LINQ to XML for Objects).

LISTING 23.6 A Sample Program That Uses the Revised Schema and XML for Listings 23.3 and 23.4, Incorporating the Collection of Luncheon Days and the Regular Expression

```

using System;
using System.Linq;
using System.Collections.Generic;
using System.Text;
using System.Xml;
using System.Xml.Linq;
using www.dustyscellar.com.TuscanLuncheon;

namespace DustysTuscanLuncheonMenu
{
    class Program
    {
        static void Main(string[] args)
        {
            var dustys = Dustys.Load("../Menu.xml");
            var menu = from lunchMenu in dustys.Luncheon
                       from item in lunchMenu.Item

```

LISTING 23.6 Continued

```
let cost = lunchMenu.Price * 8 * 1.06
let days = lunchMenu.LimitedByDays
where days.LimitedByDayOfWeek.Contains("Sunday")
select new {MenuItem=item.Name,
    Description=item.Description,
    Cost=string.Format("{0:C}", cost)};

Array.ForEach(menu.ToArray(), m=>Console.WriteLine(m));
Console.ReadLine();

}
```

If you are curious, the LinqToXsd tool generates classes containing a default constructor, properties for children, Load, Save, and Clone members, and an explicit cast operator for the XElement type. You can view this information in the Object Browser or a tool like Reflector.

LINQ to XSD is still in its incubation phase. This means that some of the details here might not work in future versions. However, the basic design goals are to support a more natural interaction with XML documents and maintain a higher level of fidelity between the intent of the schema—like supporting restrictions and regular expressions—between the XML Schema and the code-generated wrapper classes.

Summary

The way the .NET Framework is layering complexity into the framework on top of cool technologies like the CodeDOM and LINQ to XML (with LINQ to XSD) is inspired genius. This takes some of the complexity out of programming and using technologies like XML and XSD for us beleaguered programmers, letting us focus on solving problems rather than wrangling with technologies.

Check out the weblogs for more on LINQ to XSD. Everything in this book except LINQ to Entities (Chapter 17, “Introducing ADO.NET 3.0 and the Entity Framework”) and LINQ to XSD is ready for prime time. Enjoy.

Index

NUMBERS

101 LINQ Samples by Microsoft website, 203

A

Action delegate, Lambda Expressions, 104-106

Active Directory, 243

defining as data source, 248-252

helper attributes, 257-259

IQueryable provider

creating, 245-246

Smet, Bart De implementation, 245

IQueryProvider interface, implementing, 246-248

LINQ query conversions, 252-254

property assignments, 257

querying, 243-244, 260-262

RFCs, 244

schema entities, defining, 259-260

search filters, creating from Where Lambda Expressions, 254-256

Active Directory Services Interfaces, 259

Add New Item template dialog, 488

ADO.NET

Entity Framework, 383
 conceptual data models, 385
 downloading, 387
 Entity Data Models (EDMs), 386
 Go Live estimation date, 388
 relational database solutions, 385
 StockHistory database, 401-411
 web resources, 387-388
 objects, filling, 377

ADO.NET 2.0

obtaining stock quotes, updating the database, 397-401
 StockHistory database
 complete script, 394-397
 defining, 389-390
 foreign keys, adding, 393-394
 quotes, adding, 390-392

ADSI (Active Directory Services Interfaces), 259

Aggregate method, 151-153

aggregate operations

aggregation, 151-153
 averages, 154-157
 finding minimum and maximum elements, 157-159
 median values, 163-165
 overview, 151
 summing query results, 162-163

aggregation, 151-153

AJAX, 22

All, 124-125

Allow access dialog (Outlook), 242
annotating nodes, 433-434
anonymous methods
 CancelKeyPress events, 25
 delegate keyword, 25
 generic, 26-27
 nested recursion, 27-28
 regular method comparisons, 25
anonymous types
 arrays, initializing, 7-8
composite
 behaviors, adding, 9
 creating, 9
 methods, adding, 10-12
databinding, 18, 22
 binding statement, 20
 elements, editing, 21
 requesting stock quotes from Yahoo!, 19-20
 defining, 7
 equality, testing, 23
 hierarchical data shaping support, 24
 indexes in for statements, 12-14
 IntelliSense support, 6
 LINQ query examples, 24-25
 object initialization, 34-36
 overview, 6-7
 returning from functions, 17-18
 using statements, 14-16
 var keyword, 5
Any, 124-125
API methods for raw device contexts, 201

arrays

- anonymous types, initializing, 7-8
- Blackjack game, shuffling a deck of cards, 196-199
- initial capping words, 202-203
- select indexes, shuffling/unsorting, 194-195
- ToArray conversion operator, 51-53
- ascending order, sorting information in, 138-139**
- AsEnumerable conversion operator, 55-56**
- AsEnumerable method, 278-280**
- assigning Lambda Expressions, predefined generic delegates, 101
- AssociationAttribute, 301**
- associations, adding to databases, 402
- attributes**
 - Active Directory helper, 257-259
 - AssociationAttribute, 301
 - InheritanceMappingAttribute, 295
 - XElement class
 - adding to, 422
 - deleting from, 423
 - XML documents, querying, 420-421
- auto-implemented properties, 34**
- automatic properties**
 - creating custom objects with, 169
 - Lambda Expressions, 102-103
- AutoSync property (ColumnAttribute class), 272**
- Average method, 154-157**
- averages, computing, 157**
 - average file size, 154
 - average length of words, 156
 - simple averages, 154

B

- behaviors, adding to anonymous composite types, 9**
- Bill Blogs in C# website, 203**
- BinaryTree class, yield return, 89-93**
- binding control events to Lambda Expressions, 109-110**
- Blackjack game, 195-199**
- Bonaparte, Napoleon, 137**
- broadcast-listeners, 51-53**
- business objects, returning, 190-193**

C

- C#, Active Directory queries, 243-244**
- calculated values, projecting new types, 200**
- calling user-defined functions, 363-366**
- CanBeNull property (ColumnAttribute class), 272**
- Cartesian joins, 228**
- CAS (code access security), 205**
- Cast conversion operator, 54-55**
- ChangeConflicts collection, 372**
- child elements, LINQ to XML and XPath comparison, 441**
- classes**
 - closures, 117-119
 - ColoredPoint, 32-33
 - ColoredPointList, 37
 - ColumnAttribute, properties, 269-273
 - Customer, 460

DataAccess, 191
 DataContext, 305, 356-357
 DirectoryAttributeAttribute, 258-259
 DirectoryQuery, 252-254
 DirectorySchemaAttribute, 257-258
 EntitySet, adding as properties, 300-305
 EventLog, 206
 Hypergraph, 39-46
 implementing, compound type initialization, 32-34
 IOrderedQueryable, 248-252
 LINQ to SQL Class designer, 285-286
 LINQ to SQL Object Relational Designer generated, customizing, 299-300
 MAPIFolder, 242
 mapping to tables, 269-272
 Person, 36
 StackFrame, 27
 StockHistoryEntities, 405
 Supplier, 190
 TransactionScope, 366
 WebClient, 400
 XDocument, loading XML documents, 416-419
 XElement
 adding/deleting attributes, 422-423
 node annotations, 433-434
 XML documents, loading, 420
 XmlWriter
 overview, 464-465
 XML files, creating, 465-467

clauses
 from, joins, 211-212
 group by, 145-149
 let, XML intermediate values, 432-433
 Where, XML documents, 429
closures, 117-119
code, compiling Lambda Expressions as
 assigned to Expression<TDelegate>
 instance emits IL example, 114, 116
 expression tree exploration, 116-117
code access security (CAS), 205
CodeDOM, 485
collections
 ChangeConflicts, 372
 initializing, 36, 39
ColoredPoint class, 32-33
ColoredPointList class, 37
ColumnAttribute class, 269-273
columns, ignoring for conflict checks, 371
COM (Component Object Model), 239
comma-delimited text files, 453
CompareTo method, 159
compiling Lambda Expressions as code/data
 assigned to Expression<TDelegate>
 instance emits IL example, 114, 116
 expression tree exploration example, 116-117
Component Object Model (COM), 239
composite anonymous types
 behaviors, adding, 9
 creating, 9
 methods, adding, 10-12

- composite keys, defining joins**, 237
- composite resultsets, creating**, 182-183
- compound initialization**
 - anonymous types, 34-36
 - collections, 36, 39
 - named types, 31
 - auto-implemented properties, 34
 - classes, implementing, 32-34
 - default constructor and property assignment, 30
 - purpose-defined constructor, 30
- Concat, extension methods**, 132-133
- concurrency conflicts**, 368
 - catching
 - comparing member conflict states, 373-374
 - entities/tables associated with conflict, 372-373
 - ignoring columns for conflict checks, 371
 - retrieving conflict information, 372
 - handling, SubmitChanges method, 369-371
 - resolving, 375-376
- concurrency control**, 368
- Configure Behavior dialog**, 362
- conflicts**
 - catching
 - conflict information, retrieving, 372
 - entities/tables associated with conflict, 372-373
 - ignoring columns for conflict checks, 371
 - member conflict states, comparing, 373-374
- concurrency conflicts**, 368
- concurrency control**, 368
- handling, SubmitChanges method**, 369-371
- optimistic concurrency**, 368
- pessimistic concurrency**, 368
- resolving**, 375-376
- console applications, creating LINQ to XSD Preview**, 487
- contacts (Outlook), adding email addresses**, 240-241
- control events, binding to Lambda Expressions**, 109-110
- conversion operators**
 - AsEnumerable, 55-56
 - Cast, 54-55
 - OfType, 54
 - ToArray, 51-53
 - ToDictionary, 57-58
 - ToList, 56-57
 - ToLookup, 58-59
- converting CSV files to XML**, 454-456
- Count method**, 157
- counting elements**, 157
- "Creating Project Templates in .NET" website**, 487
- cross joins, implementing**, 228, 236
 - Northwind database customers and products example, 229-231
 - SQL as LINQ query example, 231-236
- CSV files (comma-delimited text files), XML**
 - converting to, 454-456
 - creating from, 457-458
- currying**, 119-120

How can we make this index more useful? Email us at indexes@samspublishing.com

- Customer class, 460**
 - customizing**
 - joins, 214
 - defining, 215-218
 - multiple predicates, 219-220
 - temporary range variables, 220-223
 - LINQ to SQL Object Relational Designer
 - generated classes, 299-300
 - objects, instantiating, 170-171
 - select statement predicates, 190
- D**
- data**
 - compiling Lambda Expressions as
 - assigned to Expression<TDelegate>
 - instance emits IL example, 114-116
 - expression tree exploration, 116-117
 - LINQ to SQL
 - adding to, 349-352
 - deleting from, 352-354
 - updating in, 354-355
 - data access layers, writing, 384**
 - DataAccess class, 191**
 - databases**
 - creating with LINQ to SQL, 305-307
 - relational model
 - C# programming problems, 384-385
 - data access layers, 384
 - Entity Framework solution, 385
 - StockHistory (ADO.NET 2.0)**
 - complete script, 394-397
 - defining, 389-390
 - foreign keys, adding, 393-394
 - obtaining stock quotes, updating the database, 397-401
 - quotes, adding, 390-392
 - StockHistory (ADO.NET Entity Framework)**
 - associations, adding, 402
 - creating EDMs, 401-402
 - LINQ to XML and LINQ to Entities, 407-411
 - querying EDMs with Entity SQL, 402-405
 - querying EDMs with LINQ to Entities, 405-406
 - databinding**
 - anonymous types, 18
 - binding statement, 20
 - elements, editing, 21
 - requesting stock quotes from Yahoo!, 19-20
 - bindability, 345
 - IEnumerable interface, 345
 - listing example, 345-347
 - DataContext class, 305, 356-357**
 - DataContext object, 275-277**
 - DataContextMapping property, 372**
 - DataSets**
 - DataTables**
 - querying with Where clause, 280-281
 - selecting data from, 278-280
 - sorting against, 282

joins, defining with, 282-284
LINQ to DataSets
 equijoins, 310-312
 left outer joins, 313-315
 nonequijoins, 312-313
 right joins, 315-317
 overview, 277-278
 partitioning methods, 282
DbType property (ColumnAttribute class), 272
de Gaulle, Charles, 151
debugging
 stored procedures, 392
 XSLT documents, 450
Decorator Structural pattern, 61
DefaultIfEmpty method, 127, 331
defining
 Active Directory schema entities, 259-260
 anonymous types, 7
 exclusive sets, 177-181
 generic extension methods, 69-70, 73
 joins
 based on composite keys, 237
 cross joins, 228-236
 group joins, 224-226
 left outer joins, 226-228
 with DataSets, 282-284
 nonequijoins, 215-218
 multiple predicates, 219-220
 temporary range variables, 220-223
 partial methods, 79-84
 stored procedures, 358-360
 tables, 266-269
 XML as strings, 424-425
delegate keyword, anonymous methods, 25
delegates, Lambda Expressions
 Action, 104-106
 Predicate<T>, 108-109
deleting
 attributes, XElement class, 423
 data, LINQ to SQL, 352-354
descending keyword, 138-140
descending order, sorting information in, 139-141
design goals, LINQ to XSD, 486
dictionaries, list conversions, 57-58
DirectoryAttributeAttribute class, 258-259
DirectoryQuery class, 252-254
DirectorySchemaAttribute class, 257-258
distinct elements
 customer objects
 defining Order object, 169
 instantiating, 170-171
 distinct lists of cities, sorting/returning, 173-177
 finding, 167
 IEqualityComparer interface, implementing, 171-172
 median grade, determining from list of numbers, 167-168
 object dumper, implementing, 172-173
Distinct method, 167

DLLs (Dynamic Link Libraries),
importing, 200

documents

XML documents

- creating from Yahoo! stock quotes, 426-427
- defining as strings, 424-425
- element navigation based on context, 430-431
- filtering, 429
- functional construction, 450-451
- intermediate values, 432-433
- loading, 415-416
- missing data, 425-426
- namespaces, 427-428
- nested queries, 428-429
- node annotations, 433-434
- querying, 416-421
- sorting, 431

XSLT documents, debugging, 450

downloading

Entity Framework, 387-388

LINQ to XSD, 487

duct typing, 64

Dump, overloading extension methods,
62-63, 67-68

dynamic programming, Lambda Expressions

- code/data, compiling as, 114-117
- OrderBy<T> method, 113
- Select<T> method, 110-112
- Where<T> method, 112-113

E

EDMs (Entity Data Models), 386

- creating, 401-402
- querying
- Entity SQL, 402-405
- LINQ to Entities, 405-406

element operations, 131-132

elements

- child elements, LINQ to XML and XPath comparison, 441
- counting, aggregate operations, 157
- distinct elements
 - defining Order object in customer objects, 169
 - determining median grade from list of numbers, 167-168
 - finding, 167
 - implementing IEqualityComparer interface, 171-172
 - implementing object Dumper, 172-173
 - instantiating customer objects, 170-171
 - sorting/returning distinct lists of cities, 173-177
 - editing, databinding, 21
 - filtering, LINQ to XML and XPath comparison, 442-443
 - maximum elements, finding, 157-159
 - minimum elements, finding, 157-159
 - navigation based on context, XML documents, 430-431

obtaining specific elements from sequences, 131-132

sibling elements, LINQ to XML and XPath comparison, 442

ElementsAfterSelf method, 430-431

email addresses, adding to Outlook contacts, 240-241

embedded LINQ queries, XML with, 458

- console application, 460-461
- Customer class example, 459-460
- literal XML with embedded expressions and LINQ, 461-462

Empty, 127

entities

- Active Directory schema, defining, 259-260
- associated with conflict, 372-373
- LINQ to Entities
 - EDMs, querying, 405-406
 - StockHistory database, UpdatePriceHistory method, 407-411
- nullable, 290-293

Entity Data Models, 386

Entity Framework (ADO.NET), 383

- conceptual data models, 385
- downloading, 387
- EDMs, 386
- Go Live estimation date, 388
- relational database solutions, 385
- StockHistory database
 - adding associations, 402
 - creating EDMs, 401-402
 - LINQ to XML and LINQ to Entities, 407-411

querying EDMs with Entity SQL, 402-405

querying EDMs with LINQ to Entities, 405-406

web resources

- Entity SQL blog, 387
- samples, 388
- Wikipedia, 387

Entity SQL

- blog, 387
- EDMs, querying, 402-405
- website, 405

EntitySet classes, adding as properties, 300-305

equality testing, 23, 129-130

Equals method, anonymous types, 23

equijoins, 214

- LINQ to Datasets, 310-312
- LINQ to SQL, 317-321

esoterism, 27

Euclidean algorithm example, 186-189

EventLog class, 206

Except method, 177-181

exclusive sets, defining, 177-181

Expression property (ColumnAttribute class), 272

expression trees, 116

expressions

- Lambda
 - assigning to predefined generic delegates, 101
 - automatic properties, 102-103
 - capturing as generic actions, 104-106

How can we make this index more useful? Email us at indexes@samspublishing.com

- capturing as generic predicates, 108-109
- control events, binding, 109-110
- currying, 119-120
- delegate role listing, 100
- reading, 103-104
- string searches, 106-107
 - Where, 254-256
- regular, adding to XML Schema files, 491-494
- Extensible Stylesheet Language Transformations, 437**
- extension methods, 61-63, 151**
 - Concat, 132-133
 - defining generic extension methods, 69-70, 73
 - defining with return type, 64-65
 - implementing, 64-67
 - LINQ, 73-77
 - overloading, 67-68
 - SequenceEqual, 130
 - “talking” string extension methods, 78-79
 - uses for, 63-64
 - Where, 73, 76
- F**
- Feynman, Richard, 179**
- Fibonacci numbers, 8, 177**
- Field method, 280-281**
- filtering**
 - elements, LINQ to XML and XPath comparison, 442-443
 - information, 122-124
 - OfType filters, 122-124
 - XML documents, 429
- finding**
 - distinct elements, 167
 - defining customer Order object, 169
 - determining median grade from list of numbers, 167-168
 - implementing IEqualityComparer interface, 171-172
 - implementing object Dumper, 172-173
 - instantiating custom objects, 170-171
 - sorting and returning distinct list of cities, 173-177
 - minimum and maximum elements, 157-159
 - for statements, anonymous type indexes, 12-14
 - foreign keys, adding to databases, 393-394
 - from clauses, joins, 211-212
 - function pointers, listings
 - anonymous delegate, 99
 - delegates in C#, 99
 - FunctionPointer definition, 98
 - Lambda Expression playing the delegate role, 100
 - functional construction, 443, 450-451
 - functions
 - anonymous types, returning, 17-18
 - ProductsUnderThisUnitPrice, 363-366
 - user-defined, calling, 363-366

G

GDI+, API methods for raw device contexts, 201

generation operations

DefaultIfEmpty, 127

Empty, 127

Range, 127

Repeat, 128-129

generic anonymous methods, 26-27

generic extension methods, defining, 69-70, 73

GetData method, 476

GetPoints method, 38

group by clause, 145-149

group joins

defining, 224-226

LINQ to SQL, 321-331

grouping information, 145-150

GroupJoin method, 321-331

H

helper attributes, Active Directory, 257-259

Hypergraph class, 39-46

broadcast-listener, 53

ColoredPoint class, 32-33

compound type initialization of objects

default constructor and property assignment, 30

Paint Event handler, 31

Pen object, 30

HypergraphController user control, 47-50

IHypergraph interface, 46-47

images, saving to files, 51-52

subject and observer interfaces, 50

I

IBindingList interface, databinding, 345

IComparable interface, 159

IDataReader interface methods, 215-218

IEnumerable{T}, 94

IEnumerable interface

AsEnumerable conversion operator, 55

databinding, 345

IEqualityComparer interface, implementing, 171-172

IHypergraph interface, 46-47

IL (Intermediate Language), 7

ILDASM (Intermediate Language Disassembler), 7

importing DLLs, 200

Inbox (Outlook), reading, 240-241

indexes

anonymous type, for statements, 12-14

select, shuffling/unsorting arrays, 194-195

SelectMany method, 207

SelectMany methods, 208

information filtering, 122-124

inheritance hierarchies, LINQ to SQL

Object Relational Designer, creating with, 298

single-table mapping, 294-298

- InheritanceMappingAttribute, 295**
- initial capping words (arrays), 202-203**
- initializing**
 - anonymous type arrays, 7-8
 - collections, 36, 39
 - objects with
 - anonymous types, 34-36
 - named types, 30-34
- inner joins, 213-214**
- InnerGetQuote method, 20**
- InsertCustomer methods, 362-363**
- InsertQuote stored procedure, 390-392**
- IntelliSense, anonymous types support, 6**
- interfaces**
 - IBindingList, databinding, 345**
 - IComparable, 159**
 - IDataReader, methods, 215-218**
 - IEnumerable**
 - AsEnumerable conversion operator, 55
 - databinding, 345
 - IEqualityComparer, implementing, 171-172**
 - IHypergraph, 46-47**
 - IQueryProvider, implementing, 246-248**
 - projecting interfaces, support for, 159-161
- Intermediate Language, 7**
- Intermediate Language Disassembler, 7**
- intermediate values, XML documents, 432-433**
- Intersect method, 177-181**
- IOrderedQueryable class, 248-252**
- IQueryable, 73**
- IQueryable provider**
 - creating, 245-246
 - Smet, Bart De implementation, 245
- IQueryProvider interface, implementing, 246-248**
- IsDbGenerated property (ColumnAttribute class), 273**
- IsDiscriminator property (ColumnAttribute class), 273**
- IsPrimaryKey property (ColumnAttribute class), 273**
- IsVersion property (ColumnAttribute class), 273**

J

joins

- based on composite keys, 237
- cross
 - implementing, 228, 236
 - Northwind database customers and products example, 229-231
 - SQL as LINQ query example, 231-236
- DataSets, defining with, 282-284
- equijoins, 214
 - LINQ to DataSets, 310-312
 - LINQ to SQL, 317-321
- group
 - defining, 224-226
 - LINQ to SQL, 321-331
- inner, 213-214

left, LINQ to SQL, 331-340

left outer, 224

implementing, 226-228

LINQ to DataSets, 313-315

multiple from clauses, 211-212

nonequijoins, 214

defining, 215-218

LINQ to DataSets, 312-313

multiple predicates, 219-220

temporary range variables, 220-223

right joins, LINQ to DataSets, 315-317

delegate role listing, 100

dynamic programming

compiling as code/data, 114-117

OrderBy<T> method, 113

Select<T> method, 110-112

Where<T> method, 112-113

predefined generic delegates, assigning to, 101

reading, 103-104

string searches, 106-107

Where, converting to Active Directory search filters, 254-256

LDAP (Lightweight Directory Access Protocol), 244

left joins, LINQ to SQL, 331-340

left outer joins, 224

implementing, 226-228

LINQ to Datasets, 313-315

Leonardo of Pisa, 177

let clause, XML intermediate values, 432-433

LINQ (Language INtegrated Query), 121

constructing queries, 122

equality testing, 129-130

extension methods, 73-77

LINQ to DataSets, joins

equijoins, 310-312

left outer joins, 313-315

nonequijoins, 312-313

right joins, 315-317

LINQ to Entities

EDMs, querying, 405-406

StockHistory database

UpdatePriceHistory method, 407-411

K–L

Kernighan, Brian, 98

keys

composite, 237

foreign, adding to databases, 393-394

keywords

delegate, anonymous, 25

descending, 138-140

orderby, 137

var, anonymous types, 5

Lambda Expressions

automatic properties, 102-103

capturing as

generic actions, 104-106

generic predicates, 108-109

closures, 117-119

control events, binding, 109-110

currying, 119-120

LINQ to SQL

data
 adding, 349-352
 deleting, 352-354
 updating, 354-355
 databases, creating, 305-307
 databinding
 bindability, 345
 IEnumerable interface, 345
 listing example, 345-347
 inheritance hierarchies
 creating with Object Relational Designer, 298
 single-table mapping, 294-298
 joins
 equijoins, 317-321
 group, 321-331
 left, 331-340
 n-tier applications, 376
 client with reference to the service, 380-381
 service contract for serializing Customer objects, 377-379
 service contract, implementing, 379
 WCF middle tier, 377
 Object Relational Designer generated classes, customizing, 299-300
 views, querying, 342-344
 Visual Designer, mapping stored procedures, 360-363

LINQ to SQL Class designer, 285-286**LINQ to XML**

node annotations, 433-434
 StockHistory database
 UpdatePriceHistory method, 407-411

XML documents

creating from Yahoo! stock quotes, 426-427
 element navigation based on context, 430-431
 filtering, 429
 intermediate values, 432-433
 namespaces, 427-428
 nested queries, 428-429
 sorting, 431
 XPath, compared, 438
 child elements, 441
 filtering elements, 442-443
 namespaces, 439-441
 sibling elements, 442
 XSLT, compared, 443
 debugging XSLT documents, 450
 HTML documents, 444-449

LINQ to XSD

design goals, 486
 downloading/installing, 487
 object queries, 496-498
 overview, 485
 Preview console applications, creating, 487
 regular expressions added to XML Schema files, 491-494
 XML files, defining, 488-490
 XML Schema files, defining, 490-491

listings

Active Directory
 DirectorySchemaAttribute class, 257
 LINQ query conversions to Active Directory queries, 253-254
 property assignments, 257

querying, 260-262
 schema entities, 259-260
 search filters created with Where Lambda Expressions query, 254-256

Active Directory queries with straight C# code, 243-244

anonymous methods handling CancelKeyPress event, 25

anonymous types
 adding behaviors to, 10
 equality testing, 23
 indexes in for statements, 12-13
 initializing, 7, 35-36
 returning from functions, 17
 using statements, 14-15

AsEnumerable conversion operator, 55

ASP for AJAX page, 21

behaviors, adding to anonymous type, 10

Blackjack game
 jack namespace, 439
 shuffling a deck of cards, 196-199
 statistics saved to XML file, 438

Cast conversion operator, 54

composite anonymous types, 9

concurrency conflicts
 comparing member conflict states, 373
 conflict information, retrieving, 372
 entities/tables associated with conflict, 372
 handling, 369
 ignoring columns for conflict checks, 371
 resolving, 375

data
 adding, 350-352
 deleting, 352-354
 updating, 354

databinding with LINQ to SQL, 345-347

DataContext class, 356-357

DirectoryAttributeAttribute class, 258-259

EntitySet classes as properties, adding, 301-305

function pointers
 anonymous delegate, 99
 delegates in C#, 99
 FunctionPointer definition, 98
 Lambda Expression playing the delegate role, 100

functional construction, 451

GDI API methods for raw device contexts, 201

generic anonymous methods, 26

Hypergraph
 broadcast-listener, 53
 ColoredPoint class, 32-33
 ColoredPointList class, 37
 default constructor and property assignment, 30
 Hypergraph class, 39-46
 HypergraphController user control, 47-50
 IHypergraph interface, 46-47
 named types, 31
 purpose-defined constructor, 30
 saving images to files, 51-52
 subject and observer interfaces, 50

IOrderedQueryable class, 249-252

How can we make this index more useful? Email us at indexes@samspublishing.com

- IQueryProvider interface, 246
- joins
 - based on composite keys, 237
 - cross join for Northwind database customers and products, 229-231
 - cross join SQL as LINQ query, 231-236
 - group joins, 224-226
 - inner, 213-214
 - left outer joins, 227
 - multiple from clauses, 211
 - nonequijoins with multiple predicates, 219
 - nonequijoins with temporary range variables, 220-223
 - nonequijoins, defining, 215-218
- Lambda Expressions
 - assigned to Expression<TDelegate> instance emits IL example, 114-116
 - assigning to predefined generic delegates, 101
 - automatic properties, 102
 - capturing as generic actions, 104-105
 - capturing as generic predicates, 108
 - closures, 118
 - control events, binding, 109
 - currying, 119
 - demonstrating explicit argument types, 103
 - expression tree exploration, 116-117
 - OrderBy<T> method, 113
 - Select<T> method, 110-111
 - string searches, 106-107
 - Where<T> method, 112
- LINQ to DataSets
 - equijoins, 310-311
 - left outer joins, 314-315
 - nonequijoins, 312-313
 - right outer joins, 316
- LINQ to SQL
 - creating databases, 305
 - customizing Object Relational Designer generated classes, 299-300
 - equijoins, 317-321
 - group joins, 321-331
 - inheritance hierarchies, 294-298
 - left joins, 331-340
 - regular expressions added to XML Schema files, 492-494
- LINQ to XML and XPath comparison
 - child elements, 441
 - filtering elements, 442
 - namespaces, 440
 - sibling elements, 442
- LINQ to XML and XSLT comparison, HTML documents, 444-449
- LINQ to XSD
 - queries, 496-497
 - XML files, creating, 488-490
 - XML Schema files, creating, 490
- n-tier applications with LINQ to SQL
 - client with reference to the service, 380-381
 - service contract for serializing Customer objects, 377-379
 - service contract, implementing, 379
- nested recursive anonymous generic methods, 27

nullable type entities, 290-293
Oftype conversion operator, 54
Outlook
 updating contacts, 240-241
 Inbox, 240-241
PetCemetery.XML file, 417-419
query examples with anonymous types, 24
requesting stock quotes from Yahoo!, 19-20
select statements
 customizing predicates, 190
 function call effects, 186-189
 indexes for shuffling/unsorting arrays, 194-195
 initial capping words in arrays, 202-203
 projecting types, 203
 returning custom business objects, 191-193
SelectMany methods
 comparing Windows Registry sections, 206-207
 indexes, 207
 projecting types, 203
SQL to XML conversions, 473-475
 Northwind **DataContext** example, 472
 Northwind object-relational map example, 471-472
 TreeView output of XML document, 476
SQL updates from XML
 examining inserted data, 481
 inserting data, 480-481
 osql.exe scripting output, 482-483
 sample XML file, 478-479
StockHistory database
 adding quotes, 390-392
 Company table, 390
 complete script, 394-395, 397
 foreign keys, adding, 394
 LINQ to XML and LINQ to Entities, 407-411
 obtaining stock quotes to update the database, 397-401
 PriceHistory table, 390
 querying EDMs with Entity SQL, 404-405
 querying EDMs with LINQ to Entities, 405
stored procedures
 defining, 358-360
 mapping with LINQ to SQL Visual Designer, 362-363
 UpdateCustomer example, 357-358
ToDictionary conversion operator, 57
ToList conversion operator, 56
ToLookup conversion operator, 58
transactions, deleting parent/child rows, 366-368
user-defined functions, calling, 363-365
views
 building with SQL Server, 342
 querying with LINQ to SQL, 342-344
XElement class, adding/deleting attributes, 422-423
XML
 creating from CSV files, 454-456
 defining as strings, 424
 missing data, 425-426
 text files, creating, 457-458

How can we make this index more useful? Email us at indexes@samspublishing.com

- XML documents
- creating from Yahoo! stock quotes, 426-427
 - element navigation based on context, 430-431
 - filtering, 429
 - intermediate values, 432-433
 - namespaces, 427-428
 - nested queries, 428-429
 - node annotations, 433-434
 - querying, 416-417, 420-421
 - sorting, 431
- XML with embedded LINQ queries in VB
- console application, 460-461
 - Customer class example, 459-460
 - literal XML with embedded expressions and LINQ, 461-462
- XmlWriter class for creating XML files, 465-467
- lists, converting**
- dictionaries, to, 57-58
 - query results to, 56-57
- literal XML in VB with embedded expressions and LINQ, 461-462**
- LongCount method, 157**
- lookups, IEnumerable object conversions, 58-59**
- luncheon menu example**
- luncheon days collection and regular expression incorporation, 497
 - possible weekdays XML document, 492-494
 - XML file, 488
 - XML Schema file, 490
- M**
- MAPIFolder class, 242**
- mapping**
- classes to tables, 269-272
 - LINQ to SQL inheritance hierarchies
 - creating with Object Relational Designer, 298
 - single-table mappings, 294-298
 - stored procedures, LINQ to SQL Visual Designer, 360-363
- Max method, 157-159**
- maximum elements, finding, 157-159**
- McCarthy, Dan, 177**
- Median method, 163-165**
- median grade, determining from list of numbers, 167-168**
- median values, 163-165**
- member conflict states, comparing, 373-374**
- methods**
- Aggregate, 151-153
 - anonymous composite types, adding to, 10-12
 - anonymous methods
 - CancelKeyPress events, 25
 - delegate keyword, 25
 - generic, 26-27
 - nested recursion, 27-28
 - regular method comparisons, 25
 - API for raw device contexts, 201
 - AsEnumerable, 278-280
 - Average, 154-157
 - CompareTo, 159

Count, 157
DefaultIfEmpty, 331
Distinct, 167
ElementsAfterSelf, 430-431
Equals, anonymous types, 23
Except, 177-181
extension methods, 61-63, 151
 Concat, 132-133
 defining generic extension methods, 69-70, 73
 defining with return type, 64-65
 implementing, 64-67
 LINQ, 73-77
 overloading, 67-68
 SequenceEqual, 130
 “talking” string extension methods, 78-79
 uses for, 63-64
 Where, 73, 76
Field, 280-281
GetData, 476
GetPoints, 38
GroupJoin, 321-331
IDataReader interface, 215-218
InnerGetQuote, 20
InsertCustomer, 362-363
Intersect, 177-181
LongCount, 157
Max, 157-159
Median, 163-165
Min, 157-159
ObjectChangeConflict.Resolve, 375
OrderBy<T>, Lambda Expressions, 113
OrderByDescending, 140
partial methods, 79-84
partitioning methods, 282
ReadSuppliers, 191
Reverse, 144-145
Select<T>, Lambda Expressions, 110-112
SelectMany
 indexes, 207-208
 types, projecting, 203-205
Windows Registry sections, comparing, 206-207
SubmitChanges, 369-371
Sum, 162-163
ThenBy, 138
ThenByDescending, 141
ToLookup, 150
Union, 182-183
Update, databinding anonymous types, 20
UpdatePriceHistory, 400
Where<T>, Lambda Expressions, 112-113
Microsoft Intermediate Language, 7
Microsoft XML Team WebLog website, 487
Min method, 157-159
minimum elements, finding, 157-159
missing data (XML), 425-426
MSIL (Microsoft Intermediate Language), 7
MyPoint property, 34

How can we make this index more useful? Email us at indexes@samspublishing.com

N

n-tier applications, 376

- client with reference to the service, 380-381
- service contracts
 - implementing, 379
 - serializing Customer objects, 377-379
- WCF middle tier, 377

Name property (ColumnAttribute class), 273

named types, object initialization, 31

- auto-implemented properties, 34
- classes, implementing, 32-34
- default constructor and property assignment, 30
- purpose-defined constructor, 30

namespaces

- LINQ to XML and XPath comparison, 439-441
- XML documents, 427-428

nanotechnology, 179

Napoleon, 137

nested queries, XML documents, 428-429

nested recursive anonymous generic methods, 27-28

New Association dialog, 402

nodes

- annotations, 433-434
- XComment, SQL to XML conversions, 475

nonequijoins, 214

- defining, 215-218

LINQ to Datasets, 312-313

multiple predicates, 219-220

temporary range variables, 220-223

Northwind Customers table object-relational map, 472

Northwind database

cross join of customers and products, 229-231

customers

adding, 350-352

deleting, 352-354, 366-368

table object-relational map, 471

Customers table object-relational map, 471

data, updating, 354

DataContext example, 472

examining inserted data, 481

InsertCustomer methods, 362-363

inserting data, 480-481

new customers XML file, 478-479

orders, deleting, 366-368

osql.exe scripting output, 482-483

ProductsUnderThisUnitPrice function, 363-366

stored procedure for CustomerIDs, 358-360

UpdateCustomer stored procedures, 357-358

views

Orders/Order Details tables, 342

querying, 342-344

nullable types, 289-293

O

Object Relational Designer, LINQ to SQL

generated classes, customizing, 299-300

inheritance hierarchies, 298

object-relational maps, XML conversions, 470-472

ObjectChangeConflict.Resolve method, 375

objects

ADO.NET, filling with, 377

compound initialization with anonymous types, 34-36

compound initialization with named types, 31

auto-implemented properties, 34

classes, implementing, 32-34

default constructor and property type, 30

purpose-defined constructor, 30

custom business, returning, 190-193

custom objects, instantiating, 170-171

DataContext, 275-277

LINQ to XML queries, 496-498

object dumper, implementing, 172-173

Order, defining, 169

tables

defining, 266-269

mapping classes to, 269-272

XNamespace, 427-428

OfType conversion operator, 54

OfType filter, 122-124

operations

element operations, 131-132

generation operations

DefaultIfEmpty, 127

Empty, 127

Range, 127

Repeat, 128-129

optimistic concurrency, 368

Order object, defining, 169

orderby keyword, 137

OrderBy<T> method, Lambda Expression, 113

OrderByDescending method, 140

osql.exe command line, examining inserted data, 482-483

Outlook

Allow access dialog, 242

contacts, adding email addresses, 240-241

Inbox/contacts, reading, 240-241

instances, creating, 242

overloading extension methods, 67-68

P

partial methods, defining, 79-84

partitioning, 282

Skip, 126-127

Take, 126-127

Person class, 36

pessimistic concurrency, 368

- PetCemetary.XML file example, 417-419**
- phishing, 205**
- Predicate<T> delegate, Lambda Expressions, 108-109**
- predicates**
 - nonequijoins, defining, 219-220
 - select statements, customizing, 190
- prime number algorithm examples, 186-189**
- PRINT statements, debugging stored procedures, 392**
- ProductsUnderThisUnitPrice function, 363-366**
- profiling code, yield return, 93-94**
- programming**
 - anonymous types
 - arrays, initializing, 7-8
 - composite, 9-12
 - composite, creating, 9
 - defining, 7
 - indexes in for statements, 12-14
 - returning from functions, 17-18
 - using statements, 14-16
 - dynamic programming, Lambda Expressions, 110
- LINQ to XSD**
 - downloading/installing, 487
 - object queries, 496-498
 - Preview console applications, creating, 487
 - regular expressions added to XML Schema files, 491-494
 - XML files, defining, 488-490
 - XML Schema files, defining, 490-491
- "Programming for Fun and Profit—Using the Card.dll" website, 439**
- projecting interfaces, support for, 159-161**
- projecting new types, 200, 203-205**
- projections, 35, 203**
- properties**
 - Active Directory, assigning, 257
 - auto-implemented, 34
 - automatic properties
 - creating custom objects with, 169
 - Lambda Expressions, 102-103
 - ColumnAttribute class, 269-273
 - DataContextMapping, 372
 - EntitySet classes as, 300-305
 - MyPoint, 34
- providers, IQueryabble**
 - creating, 245-246
 - Smet, Bart De implementation, 245

Q

- quantifiers, 126**
 - All, 124-125
 - Any, 124-125
- querying**
 - Active Directory, 243-244, 252-254, 260-262
 - converting
 - results to lists, 56-57
 - to Active Directory queries, 252-254
 - EDMs
 - Entity SQL, 402-405
 - LINQ to Entities, 405-406

embedded with XML in VB, 458
console application, 460-461
Customer class example, 459-460
literal XML with embedded
expressions and LINQ, 461-462
joins
 based on composite keys, 237
 cross, 228-236
 equijoins, 214
 group, 224-226
 inner, 213-214
 left outer, 224-228
 multiple from clauses, 211-212
 nonequijoins, 214-223
LINQ queries, constructing, 122
LINQ to XSD, 496, 498
LINQ with anonymous types, 24-25
nested, LINQ to XML, 428-429
results, summing, 162-163
text, viewing, 273-275
views, LINQ to SQL, 342, 344
XML documents
 attributes, 420-421
 XDocument class, 416-419
 XElement class, 420

R

Range, 127
range variables, defining nonequijoins,
220-223
ReaderHelper, 73

ReadSuppliers method, 191
Registry
 overview, 205
 two section comparison, 206-207
regular expressions, adding to XML Schema
files, 491-494
relational data, connecting to, 275-277
relational database models
 C# programming problems, 384-385
 data access layers, 384
 Entity Framework solution, 385
Repeat, 128-129
requesting stock quotes from Yahoo!, 19-20
Requests for Comments, 244
resolving conflicts, 375-376
resources (web), ADO.NET Entity Framework
 downloads, 387
 Entity SQL blog, 387
 samples, 388
 Wikipedia, 387
results of queries, summing, 162-163
resultsets, creating composite resultsets,
182-183
return type, defining extension methods,
64-65
Reverse method, 144-145
reversing item order, 144-145
RFCs (Requests for Comments), 244
right joins, LINQ to Datasets, 315-317
Ritchie, Dennis, 98
rules for yield return, 88

S**Santana, Carlos, 119****ScottGu's Blog website, 203****secondary sorts, 141-144****security, CAS (code access security), 205****select indexes, shuffling/unsorting arrays, 194-195****select statements**

custom business objects, returning, 190-193

function call effects, 186-189

initial capping words in arrays, 202-203

predicates, customizing, 190

types, projecting, 203-205

Select<T> method, Lambda Expression, 110-112**SelectMany method**

indexes, 207-208

types, projecting, 203-205

Windows Registry sections comparisons, 206-207

SequenceEqual, 130**sequences, appending with Concat, 132-133****services oriented architecture, 462****set operations**

composite resultsets, creating, 182-183

distinct elements, finding, 167

defining custom Order object, 169

determining median grade from list of numbers, 167-168

implementing IEqualityComparer interface, 171-172

implementing object dumper, 172-173**instantiating custom objects, 170-171****sorting and returning distinct list of cities, 173-177****exclusive sets, defining, 177-181****overview, 167****shaping, 35****shared source code, 86****shuffling a deck of cards (Blackjack game), 196-199****sibling elements, LINQ to XML and XPath comparison, 442****sieve of Atkin algorithm, 189****sieve of Eratosthenes algorithm example, 186-189****single-table mapping, LINQ to SQL inheritance hierarchies, 294-298****Skip, partitioning, 126-127****SOA (services oriented architectures), 462****sorting**

against DataTables, 282

distinct list of cities, 173-177

information

in ascending order, 138-139

in descending order, 139-141

overview, 137

reversing order of items, 144-145

secondary sorts, 141-144

XML queries, 431

source code (shared), 86**Space Invaders website, 463****sprocs (stored procedures), 223**

SQL (Structured Query Language)**LINQ to SQL**

- adding data, 349-352
- customizing Object Relational Designer generated classes, 299-300
- databases, creating, 305-307
- databinding, 345-347
- deleting data, 352-354
- equijoins, 317-321
- group joins, 321-331
- inheritance hierarchies, 294-298
- left joins, 331-340
- n-tier applications, 376-381
- querying views, 342-344
- updating data, 354-355

LINQ to SQL Class designer, 285-286**LINQ to SQL Visual Designer**, mapping stored procedures, 360-363**statements**, executing in Visual Studio, 481**XML**, creating, 469, 473-474

- object-relational maps, defining, 470-472
- TreeView output of XML document, 475-478
- XComment node, 475

XML, updating from

- examining inserted data, 481
- inserting data, 480-481
- osql.exe scripting output, 482-483
- sample XML file, 478-479

SQL Server, building views, 340-342**SqlMetal**, 285**StackFrame class**, 27**statements**

- anonymous types, 14-16
- binding statements in, 20
- using, 14-16
- binding statements, anonymous types, 20
- for statements, anonymous type indexes, 12-14
- PRINT, debugging stored procedures, 392
- select
 - custom business objects, returning, 190-193
 - customizing predicates, 190
 - function call effects, 186-189
 - initial capping words in arrays, 202-203
 - projecting types, 203-205
- SQL, executing in Visual Studio, 481

StockHistory database**ADO.NET 2.0**

- adding foreign keys, 393-394
- adding quotes, 390-392
- complete script, 394-397
- defining, 389-390

Entity Framework (ADO.NET)

- adding associations, 402
- creating EDMs, 401-402
- LINQ to XML and LINQ to Entities, 407-411

How can we make this index more useful? Email us at indexes@samspublishing.com

- querying EDMs with Entity SQL, 402-405
 - querying EDMs with LINQ to Entities, 405-406
 - obtaining stock quotes, updating the database, 397, 399-401
 - StockHistoryEntities class, 405**
 - Storage property (ColumnAttribute class), 273**
 - stored procedures, 223**
 - debugging, 392
 - defining, 358-360
 - InsertQuote, 390-392
 - mapping, LINQ to SQL Visual Designer, 360-363
 - overview, 355
 - UpdateCustomer example, 357-358
 - strings**
 - searching, Lambda Expressions, 106-107
 - XML defined as, 424-425
 - SubmitChanges method, 369-371**
 - Sum method, 162-163**
 - summing query results, 162-163**
 - Supplier class, 190**
 - System.Linq namespace, 73**
- T**
- tables**
 - associated with conflict, 372-373
 - DataTables**
 - querying with Where clause, 280-281
 - selecting data from, 278-280
 - sorting against, 282
- U**
- Union method, 182-183**
 - Update method, databinding anonymous types, 20**

UpdateCheck property (ColumnAttribute class), 273
UpdateCustomer stored procedure, 357-358
UpdatePriceHistory methods, 400
updating
 data, LINQ to SQL, 354-355
 SQL from XML
 examining inserted data, 481
 inserting data, 480-481
 osql.exe scripting output, 482-483
 sample XML file, 478-479
user controls, HypergraphController, 47-50
user-defined functions, calling, 363-366
using statements, anonymous types, 14-16

V

var keyword, anonymous types, 5
variables (range), defining nonequijoins,
 220-223
VB (Visual Basic)
 VB Today website, 203
 XML with embedded LINQ queries, 458
 console application, 460-461
 Customer class example, 459-460
 literal XML with embedded
 expressions and LINQ, 461-462
views, 340
 querying with LINQ to SQL, 342-344
 SQL Server, building with, 340-342

Visual Designer (LINQ to SQL), mapping
stored procedures, 360-363
Visual Studio
 SQL statements, executing, 481
 stored procedures, defining, 360

W

Wagner, Bill, 80
WCF (Windows Communication
Foundation), 377
WebClient class, 400
websites, 438
 101 LINQ Samples by Microsoft, 203
 Bill Blogs in C#, 203
 Creating Project Templates in .NET
 (quotes), 487
 Entity Framework download, 387
 Entity Framework Go Live estimation
 date, 388
 Entity SQL blog, 387
 Entity SQL reference, 405
 Microsoft XML Team WebLog, 487
 Programming for Fun and Profit—Using
 the Card.dll, 439
 ScottGu's Blog, 203
 Smet, Bart De IQuerybable provider
 implementation, 245
 Space Invaders, 463
 VB Today, 203
 Wikipedia, 387
 Yahoo! stock quotes, 426

West, David, 78

Where, extension methods, 73, 76

Where clauses, XML documents, 429

Where Lambda Expressions, converting to Active Directory search filters, 254-256

Where<T> method, Lambda Expression, 112-113

Wikipedia, 387

Wilde, Oscar, 167

Windows Communication Foundation, 377

Windows Registry

 overview, 205

 two section comparison, 206-207

X-Z

XComment node, SQL to XML conversions, 475

XDocument class, loading XML documents, 416-419

XElement class

 attributes

 adding, 422

 deleting, 423

 node annotations, 433-434

 XML documents, loading, 420

XML

 .csv files, creating from, 454-456

 documents

 creating from Yahoo! stock quotes, 426-427

 defining as strings, 424-425

 element navigation based on context, 430-431

 filtering, 429

 functional construction, 450-451

 intermediate values, 432-433

 loading, 415-416

 missing data, 425-426

 namespaces, 427-428

 nested queries, 428-429

 node annotations, 433-434

 querying, 416-421

 sorting, 431

 embedded LINQ queries in VB, 458

 console application, 460-461

 Customer class example, 459-460

 literal XML with embedded expressions and LINQ, 461-462

 files, creating with

 LINQ to XSD, 488-490

 XmlWriter class, 465-467

 LINQ to XML

 StockHistory database

 UpdatePriceHistory method, 407-411

 XPath, compared, 438-443

 XSLT, compared, 443-450

 Path Language, 437

 Schema files

 creating with LINQ to XSD, 490-491

 regular expressions, adding, 491-494

 SQL, creating from, 469, 473-474

 object-relational maps, defining, 470-472

 TreeView output of XML document, 475-478

 XComment node, 475

SQL, updating
examining inserted data, 481
inserting data, 480-481
osql.exe scripting output, 482-483
sample XML file, 478-479
text files, creating, 457-458

XmlWriter class

overview, 464-465
XML files, creating, 465-467

XNamespace object, 427-428

XPath (XML Path Language), LINQ to XML, 437-438

child elements, 441
filtering elements, 442-443
namespaces, 439-441
sibling elements, 442

XPath (XML Path Language), 437

XQuery, filtering elements, 442

XSLT (Extensible Stylesheet Language Transformations), LINQ to XML, 437, 443

debugging documents, 450
HTML documents, 444-449

Yahoo! stock quotes website, 426

yield return, 85-86

BinaryTree, 89-93
demonstration of, 87-88
profiling code, 93-94
rules for, 88
yield break, 95

UNLEASHED

Unleashed takes you beyond the basics, providing an exhaustive, technically sophisticated reference for professionals who need to exploit a technology to its fullest potential. It's the best resource for practical advice from the experts, and the most in-depth coverage of the latest technologies.

OTHER UNLEASHED TITLES

Microsoft Dynamics CRM 4.0 Unleashed

ISBN-13: 978-0-672-32970-8

Microsoft Exchange Server 2007 Unleashed

ISBN-13: 978-0-672-32920-3

Microsoft Expression Blend Unleashed

ISBN-13: 978-0-672-32931-9

Microsoft ISA Server 2006 Unleashed

ISBN-13: 978-0-672-32919-7

Microsoft Office Project Server 2007 Unleashed

ISBN-13: 978-0-672-32921-0

Microsoft SharePoint 2007 Development Unleashed

ISBN-13: 978-0-672-32903-6

Microsoft Small Business Server 2008 Unleashed

ISBN-13: 978-0-672-32957-9

Microsoft SQL Server 2005 Unleashed

ISBN-13: 978-0-672-32824-4

Microsoft XNA Unleashed

ISBN-13: 978-0-672-32964-7

Silverlight 1.0 Unleashed

ISBN-13: 978-0-672-33007-0

System Center Operations Manager 2007 Unleashed

ISBN-13: 978-0-672-32955-5

VBScript, WMI and ADSI Unleashed

ISBN-13: 978-0-321-50171-4

Windows Communication Foundation Unleashed

ISBN-13: 978-0-672-32948-7

Windows PowerShell Unleashed

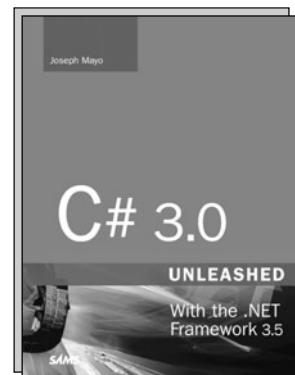
ISBN-13: 978-0-672-32953-1

Windows Presentation Foundation Unleashed

ISBN-13: 978-0-672-32891-6

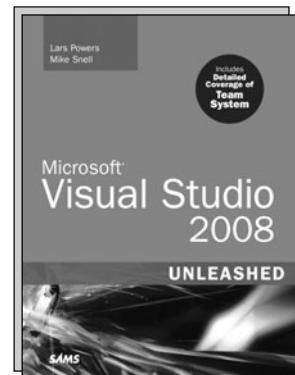
Windows Server 2008 Unleashed

ISBN-13: 978-0-672-32930-2



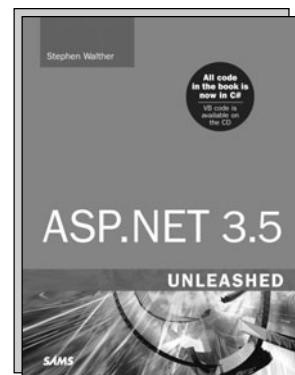
C# 3.0 Unleashed

ISBN-13: 978-0-672-32981-4



Microsoft Visual Studio 2008 Unleashed

ISBN-13: 978-0-672-32972-2



ASP.NET 3.5 Unleashed

ISBN-13: 978-0-672-33011-7

SAMS

informit.com/sams