

LINQ Programming

About the Author

Joe Mayo has been developing software since 1986. His experience includes assorted main-frame, desktop, and web development on different operating systems, including UNIX and Windows. He got started in the early days of .NET, back in July 2000, when .NET was still a pre-alpha product. These days, Joe owns Mayo Software Consulting, Inc., serving customers with custom software development and specializing in Microsoft .NET technology.

In addition to professional activities, Joe enjoys contributing to the community. He operates the C# Station website, and he blogs, twitters, and posts to various .NET-related forums. He also has written four other .NET technology books and numerous articles. Joe is honored to have received multiple Microsoft MVP awards.

About the Technical Editor

Tony Gravagno started programming in high school in 1978 and added the .NET Framework to his toolkit in 2002. He describes Nirvana as a place where he can make a living writing cool and useful code related to his other passions—foreign language, astronomy, and other sciences. When he's not reading tech docs or running his business, Nebula Research and Development, he enjoys spending time with his wife, Joleen, and his godsons, Danny and Michael.

LINQ Programming

Joe Mayo



New York Chicago San Francisco
Lisbon London Madrid Mexico City
Milan New Delhi San Juan
Seoul Singapore Sydney Toronto

Copyright © 2009 by The McGraw-Hill Companies, Inc. All rights reserved. Except as permitted under the United States Copyright Act of 1976, no part of this publication may be reproduced or distributed in any form or by any means, or stored in a database or retrieval system, without the prior written permission of the publisher.

ISBN: 978-0-07-159784-5

MHID: 0-07-159784-0

The material in this eBook also appears in the print version of this title: ISBN: 978-0-07-159783-8

MHID: 0-07-159783-2.

All trademarks are trademarks of their respective owners. Rather than put a trademark symbol after every occurrence of a trademarked name, we use names in an editorial fashion only, and to the benefit of the trademark owner, with no intention of infringement of the trademark. Where such designations appear in this book, they have been printed with initial caps.

McGraw-Hill eBooks are available at special quantity discounts to use as premiums and sales promotions, or for use in corporate training programs. To contact a representative please visit the Contact Us page at www.mhprofessional.com.

Information has been obtained by McGraw-Hill from sources believed to be reliable. However, because of the possibility of human or mechanical error by our sources, McGraw-Hill, or others, McGraw-Hill does not guarantee the accuracy, adequacy, or completeness of any information and is not responsible for any errors or omissions or the results obtained from the use of such information.

TERMS OF USE

This is a copyrighted work and The McGraw-Hill Companies, Inc. (“McGraw-Hill”) and its licensors reserve all rights in and to the work. Use of this work is subject to these terms. Except as permitted under the Copyright Act of 1976 and the right to store and retrieve one copy of the work, you may not decompile, disassemble, reverse engineer, reproduce, modify, create derivative works based upon, transmit, distribute, disseminate, sell, publish or sublicense the work or any part of it without McGraw-Hill’s prior consent. You may use the work for your own noncommercial and personal use; any other use of the work is strictly prohibited. Your right to use the work may be terminated if you fail to comply with these terms.

THE WORK IS PROVIDED “AS IS.” MCGRAW-HILL AND ITS LICENSORS MAKE NO GUARANTEES OR WARRANTIES AS TO THE ACCURACY, ADEQUACY OR COMPLETENESS OF OR RESULTS TO BE OBTAINED FROM USING THE WORK, INCLUDING ANY INFORMATION THAT CAN BE ACCESSED THROUGH THE WORK VIA HYPERLINK OR OTHERWISE, AND EXPRESSLY DISCLAIM ANY WARRANTY, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. McGraw-Hill and its licensors do not warrant or guarantee that the functions contained in the work will meet your requirements or that its operation will be uninterrupted or error free. Neither McGraw-Hill nor its licensors shall be liable to you or anyone else for any inaccuracy, error or omission, regardless of cause, in the work or for any damages resulting therefrom. McGraw-Hill has no responsibility for the content of any information accessed through the work. Under no circumstances shall McGraw-Hill and/or its licensors be liable for any indirect, incidental, special, punitive, consequential

or similar damages that result from the use of or inability to use the work, even if any of them has been advised of the possibility of such damages. This limitation of liability shall apply to any claim or cause whatsoever whether such claim or cause arises in contract, tort or otherwise.

To June Blossom Mayo

Contents at a Glance

Part I LINQ Essentials

[1 Introducing LINQ](#)

[2 Using LINQ to Objects](#)

[3 Handling LINQ to SQL with Visual Studio](#)

Part II LINQ to Any Data Source

[4 Working with ADO.NET Through LINQ to DataSet](#)

[5 Programming Objects with LINQ to Entities](#)

[6 Programming Objects with LINQ to XML](#)

[7 Automatically Generating LINQ to SQL Code with SqlMetal](#)

Part III Extending LINQ

[8 Digging Into Expression Trees and Lambdas](#)

[9 Constructing New Code with Extension Methods](#)

[10 Building a Custom LINQ Provider—Introducing LINQ to Twitter](#)

[11 Designing Applications with LINQ](#)

[12 Concurrent Programming with Parallel LINQ \(PLINQ\)](#)

[A Standard Query Operator Reference](#)

[Index](#)

Contents

[Foreword](#)

[Acknowledgments](#)

[Introduction](#)

[Part I LINQ Essentials](#)

[1 Introducing LINQ](#)

[Getting Started](#)

[Example Data](#)

[Viewing Assembly Contents and Intermediate Language](#)

[Auto-Implemented Properties](#)

[Using Auto-Implemented Properties](#)

[A Few More Tips](#)

[In Case You're Curious](#)

[Partial Methods](#)

[How Are Partial Methods Used?](#)

[A Few More Tips](#)

[In Case You're Curious](#)

[Extension Methods](#)

[Creating Extension Methods](#)

[Using Extension Methods](#)

[Extension Method Precedence](#)

[In Case You're Curious](#)

[Object Initializers](#)

[Using Object Initializers](#)

[In Case You're Curious](#)

[Collection Initializers](#)

[Implementing Collection Initializers](#)

[Collection Initializers for Dictionary Collections](#)

[Lambda Expressions](#)

[How to Use a Lambda Expression](#)

[Replacing Anonymous Methods and Delegates](#)

[A Few Tips on Lambda Expressions](#)

[Implicitly Typed Local Variables](#)

[Query Syntax](#)

[A First Look at a Very Simple Query](#)

[Using Sequences from a Query](#)

[Anonymous Types](#)

[A Simple Anonymous Type](#)

[Anonymous Types Are Strongly Typed](#)

[In Case You're Curious](#)

[Summary](#)

[2 Using LINQ to Objects](#)

[LINQ to Objects Essentials](#)

[LINQ to Objects Benefits](#)

[The Example Collections](#)

[Implementing Projections](#)

[Collection Object Projections](#)

[LINQ to Objects Queries Return IEnumerable<T>](#)

[Selecting a Single Field or Property](#)

[Projecting into Anonymous Types](#)

[Projecting into Custom Types](#)

[Filtering Data](#)

[Calculating Intermediate Values](#)

[Sorting Query Results](#)

[Sorting a Single Property](#)

[Sorting Multiple Properties](#)

[Managing Sort Direction](#)

[Grouping Sets](#)

[Grouping by a Single Property](#)

[Grouping by Multiple Properties](#)

[Accessing Grouped Objects](#)

[Joining Objects](#)

[Joining Objects](#)

[Performing Left Joins](#)

[Performing Group Joins](#)

[Joining with Composite Keys](#)

[Performing Select Many Joins](#)

[Flattening Object Hierarchies](#)

[Performing Cross-Joins](#)

[Querying Non-IEnumerable<T> Collections](#)

[A Practical Example of LINQ to Objects](#)

[Summary](#)

3 Handling LINQ to SQL with Visual Studio

[LINQ to SQL Capabilities](#)

[Introducing the Example Database](#)

[Understanding the DataContext](#)

[Creating a DataContext with the VS 2008 LINQ to SQL Designer](#)

[Database Connection Management](#)

[Object Mapping](#)

[Object Tracking](#)

[Defining Entity Relationships](#)

[Adding Multiple Entities to a DataContext](#)

[Demonstrating Entity Associations](#)

[Handling Multiple Relationships Between Two Entities](#)

[Examining Association Attributes](#)

[A Closer Look at Queries](#)

[Querying with the DataContext](#)

[Examining Generated SQL](#)

[Examining Query Results](#)

[Inserting, Updating, and Deleting](#)

[Inserting New Data](#)

[Updating Existing Data](#)

[Deleting Existing Data](#)

[Attaching Objects](#)

[Implementing Stored Procedures and Functions](#)

[Stored Procedures](#)

[Functions](#)

[Executing Raw SQL](#)

[Entity Inheritance](#)

[Summary](#)

Part II LINQ to Any Data Source

4 Working with ADO.NET Through LINQ to DataSet

[Setting Up the DataSet](#)

[Querying a DataSet](#)

[Querying DataSet Tables](#)

[Reading DataRow Results](#)

[Modifying a DataSet](#)

[Working with Strongly Typed DataSets](#)

[Creating a Strongly Typed DataSet](#)

[Querying a Strongly Typed DataSet](#)

[Special DataSet Operations](#)

[Creating Data Views from LINQ to DataSet Queries](#)

[Copying LINQ to DataSet Queries to DataTables](#)

[Summary](#)

5 Programming Objects with LINQ to Entities

[Introduction to Entities](#)

[The ADO.NET Entity Framework \(AEF\) Architecture](#)

[Creating an Entity Data Model \(EDM\)](#)

[Inside the Entity Data Model \(EDM\)](#)

[Accessing the EDM with theObjectContext](#)

[Examining EDM Entity Types](#)

[Top-Down EDM Design](#)

[What Is the Top-Down Design Approach?](#)

[Setting Up an EDM Using Top-Down Design](#)

[Creating New Entities Yourself](#)

[Mapping Entities to the Database](#)

[Querying the EDM with LINQ to Entities](#)

[Summary](#)

6 Programming Objects with LINQ to XML

[The LINQ to XML Objects](#)

[Creating XML Documents](#)

[Functional Construction](#)

[Saving XML Documents](#)

[Retrieving XML from Various Sources](#)

[Creating XML from Strings](#)

[Reading XML from Streams](#)

[Reading XML from Files](#)

[Reading XML from URIs](#)

[Reading XML from XmlReaders](#)

[Working with LINQ to XML Namespaces](#)

[Creating and Using a Namespace](#)

[Creating Namespace Prefixes](#)

[Working with Documents](#)

[Adding Declarations, Text, and Processing Instructions](#)

[Including Declarations](#)

[Defining CDATA Text](#)

[Handling Processing Instructions](#)

[Validating XML](#)

[Queries and Axes](#)

[Basic LINQ to XML Queries](#)

[LINQ to XML Axis Methods](#)

[Axis Method Filters](#)

[Manipulating XML](#)

[Adding New Elements](#)

[Modifying Existing Elements](#)

[Removing Elements](#)

[Summary](#)

7 Automatically Generating LINQ to SQL Code with SqlMetal

[Using SqlMetal.exe](#)

[Database Connection Options](#)

[Working with DBML Files](#)

[SqlMetal Source Code Options](#)

[Implementing External Mapping Files](#)

[Generating an External Mapping File with SQLMetal.exe](#)

[Using the External Mapping File in Your Code](#)

[Summary](#)

Part III Extending LINQ

8 Digging Into Expression Trees and Lambdas

[Working with Lambda Expressions](#)

[More Lambda Examples](#)

[Lambdas and the Func Delegates](#)

[Building Expression Trees](#)

[Expression Tree and Lambda Expression Conversions](#)

[Expression Tree Internals](#)

[Dynamic LINQ Queries with Expression Trees](#)

[The Where Or Problem](#)

[A Dynamic Query Building Algorithm](#)

[Executing the Dynamic Query](#)

[Summary](#)

[9 Constructing New Code with Extension Methods](#)

[Relating Query Expressions to Extension Methods](#)

[Matching Extension Methods](#)

[How Query Expressions Translate](#)

[Extension Method Composability](#)

[Building a Custom Extension Method](#)

[Summary](#)

[10 Building a Custom LINQ Provider—Introducing LINQ to Twitter](#)

[Introducing LINQ to Twitter](#)

[Overview of the LINQ Provider Development Process](#)

[.NET Framework Interfaces](#)

[Implementation Types](#)

[Data Source Communication](#)

[Additional Information Resources](#)

[Implementing LINQ Provider Interfaces](#)

[Understanding the IQueryable<T> Interface](#)

[Understanding the IQueryProvider Interface](#)

[Understanding IOrderedQueryable<T>](#)

[Implementing IQueryable<T>](#)

[Implementing IQueryProvider](#)

[Building TwitterContext](#)

[TwitterContext Instantiation](#)

[Exposing IQueryable<T>](#)

[Execution—The Bird's Eye View](#)

[Transforming the Query](#)

[Communicating with Twitter](#)

[Summary](#)

[11 Designing Applications with LINQ](#)

[Introduction to the ASP.NET LinqDataSource](#)

[N-Layer Architecture with LINQ](#)

[Creating the Data Access Layer \(DAL\)](#)

[Creating the Business Logic Layer \(BLL\)](#)

[Creating the User Interface Layer \(UI\)](#)

[Handling Data Concurrency](#)

[Implementing Pessimistic Concurrency with LINQ](#)

[Optimistic Concurrency](#)

[Working with Deferred Execution](#)

[Working with Deferred Loading](#)

[What Is Good about Deferred Loading?](#)

[What Could Be Problematic about Deferred Loading?](#)

[Resolving Deferred Loading Problems](#)

[Summary](#)

[12 Concurrent Programming with Parallel LINQ \(PLINQ\)](#)

[System Requirements and Setup for PLINQ](#)

[Running PLINQ Queries](#)

[Examining PLINQ Types](#)

[Parallelizing a Query](#)

[Two Threads Are Not Always Better than One](#)

[Choosing What to Parallelize](#)

[Setting the Degree of Parallelism](#)

[Ordering PLINQ Results](#)

[Additional PLINQ Methods](#)

[PLINQ Performance Testing](#)

[Analyzing Performance](#)

[Considering the Impact of Hardware](#)

[Handling PLINQ Exceptions](#)

[Summary](#)

[A Standard Query Operator Reference](#)

[Setting Up the Examples](#)

[Setting Up LINQ to Objects](#)

[Setting Up LINQ to SQL](#)

[Organization of the Examples](#)

[Alphabetical Standard Query Operator Reference](#)

[Aggregate](#)

[All](#)

[Any](#)

[AsEnumerable](#)

[AsQueryable](#)

[Average](#)

[Cast](#)

[Concat](#)

[Contains](#)

[Count](#)

[DefaultIfEmpty](#)

[Distinct](#)

[ElementAt](#)

[ElementAtOrDefault](#)

[Empty](#)

[Except](#)

[First](#)

[FirstOrDefault](#)

[GroupBy](#)

[GroupJoin](#)

[Intersect](#)

[Join](#)

[Last](#)

[LastOrDefault](#)

[LongCount](#)

[Max](#)

[Min](#)

[OfType](#)

[OrderBy](#)

[OrderByDescending](#)

[Range](#)

[Repeat](#)

[Reverse](#)

[Select](#)

[SelectMany](#)

[SequenceEqual](#)

[Single](#)

[SingleOrDefault](#)

[Skip](#)

[SkipWhile](#)

[Take](#)

[TakeWhile](#)

[ThenBy](#)

[ThenByDescending](#)

[ToArray](#)

[ToDictionary](#)

[ToList](#)

[ToLookup](#)

[Union](#)

[Where](#)

[**Index**](#)

Foreword

LINQ is a new technology that will help developers get more work done in less time using code that is easy to maintain. It provides a single, easy to understand query model for working with a wide range of data sources, from relational databases, to XML, to the generic collections in our programs. LINQ even includes an extensible data model that allows developers to query any arbitrary data source. Never before has it been so easy for developers to write queries against a wide range of data sources using code that is so clean, succinct, and flexible.

To reap these benefits, however, you will need a good map that lays out the new territory exposed by the LINQ architecture. Before they can fully understand LINQ, most developers are going to need to become acquainted with several new concepts. The basics of LINQ are easy to understand, but gaining a deep understanding of the conceptual framework behind LINQ requires some work.

This book provides developers with a clear, easy to read map that details the LINQ architecture and supplies us with a well thought out overview of the terrain. Joe Mayo is not only an excellent engineer, but also an excellent writer with an organized and very logical mind. He is the ideal guide to the new and exciting world of LINQ development.

Joe starts out this excellent book with an explanation of the fundamental syntactical elements introduced in C# 3.0 that form the foundation for LINQ. Building on these principles, he goes on to explain how to write query expressions, explaining the syntax in clear, well-written, and logically structured prose. From there, he goes on to cover all the key features of LINQ, include LINQ to SQL, LINQ to XML, and LINQ to DataSets.

The latter half of the book contains an excellent and highly valuable chapter on creating LINQ providers. Near the end of the text, Joe gifts us with an excellent chapter on how to design LINQ applications. This guide to many important LINQ best practices will no doubt become one of the most utilized and favorite sections of this very useful book.

I have read several LINQ books. This text is the best laid out, and the most thoughtful and thorough examination of this subject that I have seen. The book that you hold in your hands is a superior guide to a new world of programming. LINQ is both a practical tool that will help us get our work done more quickly, and also a beautifully designed tool that yields many wonders for those who take the time to study it in-depth. This book is an excellent guide to that wondrous new world, written by an expert working at the peak of his powers. We are all lucky to have such a great new guide to this exciting territory.

—Charlie Calvert
C# Community Program Manager, Microsoft

Acknowledgments

First, I would like to thank my wife, Maytinee, for her patience and support. Thanks to Charlie Calvert and Rafael Munoz of Microsoft for recommending me to McGraw-Hill. And thanks to Charlie Calvert for writing the Foreword, which I appreciate.

I would also like to thank the people at McGraw-Hill: executive editor Jane Brownlow for her leadership on this task; Jennifer Housh and Carly Stapleton, who helped coordinate chapters; and Arushi Chawla, who coordinated copy editing and page proofs. Thanks to Janet Walden for editorial supervision. I also appreciate Jan Jue's copy editing expertise, which helped in so many ways.

Thanks to Tony Gravagno who was the technical editor. He gracefully surfaced important issues that I overlooked and was instrumental in ensuring a higher-quality book.

Introduction

C# 3.0 and .NET 3.5 introduce a new set of language features and APIs for working with data called Language Integrated Query (LINQ). While many other APIs are already available to work with data, LINQ has the special features of being integrated into the C# and VB.NET programming languages, introduces functional programming, and offers a common way to query data, regardless of data source.

Without going into too much detail, which you'll learn in the rest of the book, here's a quick example of a LINQ query. The following query should be familiar if you're accustomed to using SQL. For a collection named *Products*, we want to retrieve the *Name* from all of the products whose *Color* is *Red*. The *IEnumerable<string>* is a collection that you can iterate through with a *foreach* loop:

```
IEnumerable<string> prodNames =  
    from prod in ctx.Products  
    where prod.Color == "Red"  
    select prod.Name;
```

What is useful is that the preceding query is part of the C# programming language, demonstrating the language-integrated part of LINQ. The preceding query shows how LINQ introduces *functional* programming by making a statement of *what* you want to accomplish. The functional style is different from the *imperative* style of programming you might use today, which states *how* a task should be completed. An additional benefit of this functional style is that the LINQ provider has flexibility in how it translates the query into a command that the data source understands. Because a common syntax is used across all data sources, you now have one API to learn, instead of many. For example, you can use the preceding query on multiple data sources by just changing *ctx.Products* to the syntax of the different data source—and maybe you won't have to change that, depending on the data source and how you set it up. As you read through this book, you'll find differences between providers, but they all still use the same LINQ query syntax, making it much easier for you to work with data.

The chapters in this book explain the new features of C# 3.0, all of the LINQ providers that ship with .NET, and even show you how to create your own LINQ provider. Additionally, an architecture chapter helps you design applications and understand how LINQ fits in. The following section provides an overview of the chapters in this book.

Chapter Overview

LINQ Programming has three parts, 12 chapters, and an appendix.

[Part I: LINQ Essentials](#)

This first part includes introductory material for new C# 3.0 language features, essential LINQ to Objects coverage, and features of Visual Studio 2008 (VS 2008) that you'll need to know.

[Chapter 1: Introducing LINQ](#)

While a lot of the new features of C# 3.0 were added to support LINQ, they can be useful in other parts of your code. If you're already familiar with these features, you can probably skip over this chapter and move on. However, I did add material where I took a deeper dive into each feature, and some examples contain Intermediate Language (IL). That said, you don't need to understand IL or the deep-down mechanics of the language features, but I added it anyway if you're ever curious to know more. Who

knows when a little extra information might help you solve a hard or unique problem!

[Chapter 2: Using LINQ to Objects](#)

An important feature of LINQ is how it is integrated into the programming language. This chapter shows you all of the C# language clauses that you can use to build LINQ queries. It focuses on LINQ to Objects, where collections are the source of data. You'll want to pay close attention to this chapter because you'll see this syntax used throughout the rest of the book. Of course, if you see a clause in the book and don't understand what it does, then you can refer back to this chapter. While this chapter shows you how to use clauses that are part of the language, the LINQ API includes many more options in the form of standard query operators, which I cover in the appendix.

[Chapter 3: Handling LINQ to SQL with Visual Studio](#)

LINQ to SQL is the .NET LINQ provider that directly targets Microsoft SQL Server as a database. Using VS 2008, you'll learn how to set up a LINQ to SQL project and how to write LINQ queries with automatically generated code from LINQ to SQL. You'll also learn how to perform insert, update, and delete operations, which are essential tools for any data provider. There are lessons on how to call functions and stored procedures too. You'll also go on a deep dive into how LINQ to SQL works, discovering the potential for you to extend LINQ to SQL when you need the flexibility.

[Part II: LINQ to Any Data Source](#)

Besides LINQ to SQL, .NET ships with LINQ providers for DataSet, XML, and the ADO.NET Entity Framework, which are the subjects of this part.

[Chapter 4: Working with ADO.NET Through LINQ to DataSet](#)

If you're using DataSets in your code, this chapter will explain how you can use LINQ with that existing code.

[Chapter 5: Programming Objects with LINQ to Entities](#)

XML is ubiquitous in .NET, being used in configuration files, web service messages, data files, and more. This chapter will show you how to use LINQ to XML for processing XML data sources. Besides learning how to locate elements, you'll learn how to navigate and how to build new XML documents.

[Chapter 6: Programming Objects with LINQ to XML](#)

The ADO.NET Entity Framework includes support for multiple data sources through a rich set of objects called *entities*. This chapter will explain how you can use the LINQ to Entities provider to query and manipulate entities in the ADO.NET Entity Framework.

[Chapter 7: Automatically Generating LINQ to SQL Code with SqlMetal](#)

LINQ to SQL has good VS 2008 designer support, but a command-line tool called SqlMetal.exe also ships with the .NET Framework. SqlMetal.exe allows you to generate entity files or XML files that map to entities, allowing you to write LINQ to SQL queries that map to database objects. You'll learn what options are available, how to use the options, and more about the design choices you have.

Part III: Extending LINQ

This part of the book covers more advanced material. It takes you step-by-step through lambda expressions, expression trees, and extension methods. Then you'll be ready to learn how to create a custom LINQ provider. You'll delve into a chapter on architecture and also a chapter on parallel programming with LINQ.

Chapter 8: Digging Into Expression Trees and Lambdas

Lambdas are useful in everyday coding, but especially so with LINQ. This chapter expands upon what you learned about lambdas in [Chapter 1](#). To prepare you for more advanced material, you'll also learn about the special feature of lambdas that allows you to convert a lambda to an expression tree. You'll see how expression trees are data, allowing you to extract that data and make decisions, such as translation into a format that a data source understands. Learning about expression trees is an essential tool for building a custom LINQ provider.

Chapter 9: Constructing New Code with Extension Methods

To understand how LINQ providers work, you must understand extension methods. You'll also get a glimpse into how the C# query syntax translates to extension methods.

Chapter 10: Building a Custom LINQ Provider—Introducing LINQ to Twitter

If you have a custom data source and don't have an existing provider, you can build your own, providing yourself and other developers with the common way of accessing that data via LINQ. The example in this chapter demonstrates how to write a LINQ provider, LINQ to Twitter, for the micro-blogging service called Twitter. You'll see how to borrow existing code, pull together essential information from previous chapters, and add a few unique twists to see how to create a custom LINQ provider.

Chapter 11: Designing Applications with LINQ

Your code in the editor is where the rubber meets the road, and you'll need to know how to fit LINQ into your applications. Talking about architecture and design isn't easy because there are so many ways to do it. Nonetheless, this chapter explains some of the techniques that I and others have used successfully to build and ship applications to customers.

Chapter 12: Concurrent Programming with Parallel LINQ (PLINQ)

Most of the new computers being sold today have multiple cores on their processors, and multiple CPUs is an available option; the number of cores per CPU will increase. With power, size, and heat limitations on CPUs, the way to obtain speedups in future applications will be to parallelize your work. Microsoft is addressing this need with a LINQ provider named Parallel LINQ (PLINQ). You'll learn how to use PLINQ with queries and how to handle exceptions that could occur during processing. Additionally, there are a few dos and don'ts when using PLINQ that might surprise you.

Appendix: Standard Query Operator Reference

Most of the queries you've seen throughout the book use C# query syntax, but you also have an entire library of Standard Query Operators that match and extend the query functionality of LINQ. The appendix gives you a complete list of the Standard Query Operators with signatures, explanation of parameters, and

examples of how you would code them. The appendix is organized as a reference to help you look up the various query operators that you'll need while coding.

Summary

From essential information, provider coverage, and advanced topics, *LINQ Programming* can help you learn to use LINQ as an essential tool in your development. I hope you enjoy reading this book as much as I enjoyed writing it.

All the best,
Joe Mayo

PART I

LINQ Essentials

CHAPTER 1

Introducing LINQ

CHAPTER 2

Using LINQ to Objects

CHAPTER 3

Handling LINQ to SQL
with Visual Studio

CHAPTER 1

Introducing LINQ

Wouldn't the world be a nicer place if people were able to see things the same way, listen to each other, and communicate in a common language? Perhaps that's a lot to ask when it comes to people, but maybe not so much to ask of the technology we use. Just as people communicate and share information, programmers like you and I build systems that interoperate with data. Today, the idea of being able to work with data the same way, regardless of the source, isn't such a far-off dream as it used to be—now we have *Language Integrated Query* (LINQ).

I used the concept of data in a general sense because many types of data come from diverse sources. As developers, we've become accustomed to trying to access data with different tools. The sophistication of these tools has grown over time, but the market is still swamped with different data technologies, making our lives as developers less productive than they should be. LINQ is a step in the right direction to help us overcome the diversity in data technologies that takes up so much of our time.

As your journey through this book progresses, you'll learn how to use LINQ to access different data sources. However, there will always be a single constant, LINQ. LINQ is a set of technologies added to the .NET Framework and .NET Languages to make it easier for you to work with data. Both C# and VB.NET support a SQL-like syntax that most developers will be immediately familiar with.

For you to understand how LINQ works, it is useful to examine the new features added to the C# programming language, giving meaning to the "Language Integrated" part of the LINQ acronym. VB.NET has similar new features, plus some that C# doesn't support. Nonetheless, the overall idea is that we as developers have an easy way to work with data that is now built directly into our language. The new C# 3.0 programming language enhancements include anonymous types, auto-implemented properties, extension methods, implicitly typed local variables, lambda expressions, object (and collection) initializers, partial methods, and query expressions. The visual experience of these features is dramatic, changing the way we work with code as we move into the future. Some of these features might sound strange, but by the end of this chapter, you should have a solid understanding of their purpose and how you can use them to increase your own productivity.

Prior to looking at LINQ queries, I'll show you how the new features of C# 3.0 work. But first, the next section gives you essential information on the AdventureWorks database, which examples in this book will be based on.

Getting Started

Before jumping in, I want to share a couple items that might make it easier for you to follow along: where to find example data that examples are based on and some perspective on *Intermediate Language* (IL). This book uses C# to teach the contents, but I'll dig a little deeper on occasion.

Example Data

The examples in this chapter and throughout the rest of the book are based on the AdventureWorks database. To make this more accessible to everyone, including those who are using SQL Express 2005, I'm using the AdventureWorksLT database, which you can download from www.CodePlex.com.

Although LINQ to Objects doesn't use a database, I still base my objects on the tables in

AdventureWorksLT. Hopefully, you'll find the consistent use of objects easy to follow.

Viewing Assembly Contents and Intermediate Language

This book is very much about using C#, one of the .NET-managed languages for working with LINQ, but on occasion, I'll dig a little deeper into a subject to clarify a point. This could involve looking at Assembly contents or Intermediate Language (IL). The tool I use is IL Disassembler (ILDASM), which ships with the .NET Framework SDK and also comes with Visual Studio 2008 (VS 2008).

VS 2008 users can open ILDASM through the Visual Studio 2008 Command Prompt. Just select Start | All Programs | Microsoft Visual Studio 2008 | Visual Studio Tools | Visual Studio 2008 Command Prompt. You'll see the command line in the window that appears with a prompt for your current path. Type **ildasm** and press ENTER to open ILDASM. After that, you can select File | Open, and use the File Open dialog box to navigate to the Assembly with the code you wish to view. If you're using VS 2008 Express or just using the .NET Framework SDK with your own editor, you can navigate to C:\Program Files\Microsoft SDKs\Windows\v6.0A\bin and find ildasm.

You can open and close tree branches to navigate through the types. Double-clicking a type member brings up the IL for you to read.

Since this book is about LINQ programming, I won't go into any depth on how to read IL. Rather than going off on a tangent, I'll focus only on the parts of the Assembly or IL that have a direct connection with the subject I'm explaining. Digging deeper is mostly for those who are curious, and it can sometimes clarify the behavior of a feature.

Auto-Implemented Properties

The designers of C# recognized that most properties are primarily wrappers to a single backing store with no additional logic. So, they added a new feature named *auto-implemented properties* to make it easier for us to work with this very common scenario. The following sections show you what auto-implemented properties are, provide a few tips on using them, and then dig a little deeper in case you are curious.

Using Auto-Implemented Properties

For years, we've coded C# properties like this:

```
private int m_productID;
public int ProductID
{
    get { return m_productID; }
    set { m_productID = value; }
}
```

Loyal to the object-oriented principle of encapsulation, many of us wrapped all internal state this way—even if all we did was get and set a single backing store, *m_productID*. Microsoft realized how much time was spent on this practice and gave us a shortcut called auto-implemented properties. Here are a few examples showing the abbreviated syntax of the auto-implemented property:

```
class Product
{
    public int ProductID { get; set; }
    public string Name { get; set; }
    public virtual decimal ListPrice { get; protected set; }
```

```
public DateTime SellStartDate { get; set; }  
}
```

The first member inside of the preceding example, *ProductID*, is equivalent to the C# 2.0 property shown previously. There are a couple significant differences where the auto-implemented property doesn't have accessor bodies. Additionally, you don't see backing store variables in the *Product* class. The backing store still exists, but the C# compiler will create it in Intermediate Language (IL)—it is implicit. *Name*, *ListPrice*, and *SellStartDate* are also automatic properties of type string, *decimal*, and *DateTime*, respectively.

A Few More Tips

Here are a few more facts about the automatic property that you might be curious about:

- It must include both get and set accessor prototypes. Otherwise, use a regular property.
- Access modifiers of accessors can be different; that is, the preceding *ListPrice* set accessor is protected so that it can only be set within *Product* or derived classes.
- It can be virtual, and overrides can be either automatic or regular properties.
- It is either fully auto-implemented or not; that is, if you implement custom logic in a set accessor, then the get accessor must be implemented too—no prototypes. The same applies if you implement a get accessor; then the set accessor must be implemented too.

Auto-implemented properties are one of the few new C# 3.0 language features that weren't added to support LINQ. Rather, they are a productivity feature that makes it quick and easy to add properties to a class with less syntax. Visual Studio 2008 (VS 2008) snippets make this even faster—just type **prop**, press TAB, TAB, and poof! You have a snippet template for an auto-implemented property.

In Case You're Curious

Somewhere in the discussion of auto-implemented properties, you might have begun wondering where the property value (backing store) is actually stored. That's a reasonable reaction, since we've been explicitly defining backing stores since the birth of C#. The C# compiler will automatically create the backing store in IL, as shown in [Figure 1-1](#).

The *Product* class in [Figure 1-1](#) has four properties. The highlighted line is the backing store for the auto-implemented property named *ListPrice*. As you can see, the C# compiler uses the convention `<PropertyName>k__BackingField` to name the backing store, which is `<ListPrice>k__BackingField` in the case of the *ListPrice* property. The naming convention contains the term “BackingField” though I've been using the term “backing store” to refer to it. I've found the term backing store more accurate because, outside of an auto-implemented property, nothing dictates that the information a property works with must be a field. You can also see the backing stores for the *Name*, *ProductID*, and *SellStartDate* auto-implemented properties, which use the same naming convention as *ListPrice*.

As you might know, C# properties are really a set of accessor methods in IL, with the *get* accessor named as `get_PropertyName` and the *set* accessor named as `set_PropertyName`. In the

FIGURE 1-1 IL for automatic property backing store



case of the *ListPrice* property, the method names in [Figure 1-1](#) are *get_ListPrice* and *set_ListPrice*. You can see that the other properties follow the same convention. With normal properties, the IL will reflect your implementation of the *get* and *set* accessors, but the IL for auto-implemented properties will implement the *get* and *set* accessors for returning and assigning to the backing store for you. Here's the IL generated for the *get* accessor, *get_ListPrice*:

```
.method public hidebysig newslot specialname virtual
    instance valuetype [mscorlib]System.Decimal
    get_ListPrice() cil managed
{
    .custom instance void
    [mscorlib]System.Runtime.CompilerServices
```

```
.CompilerGeneratedAttribute::.ctor() = ( 01 00 00 00 )
// Code size      11 (0xb)
.maxstack 1
.locals init (valuetype [mscorlib]System.Decimal V_0)
IL_0000: ldarg.0
IL_0001: ldfld valuetype [mscorlib]System.Decimal Chapter_01.Produc
t::'<ListPrice>k__BackingField'
IL_0006: stloc.0
IL_0007: br.s IL_0009
IL_0009: ldloc.0
IL_000a: ret
} // end of method Product::get_ListPrice
```

The line for *IL_0001* in the preceding code demonstrates that the *get* accessor for the *ListPrice* auto-implemented property is working with the backing store variable *<ListPrice>k__BackingField*. Similarly, here's the IL implementation of the *set* accessor, *set_ListPrice*:

```
.method family hidebysig newslot specialname virtual
instance void set_ListPrice(valuetype
[mscorlib]System.Decimal 'value') cil managed
{
.custom instance void
[mscorlib]System.Runtime.CompilerServices
.CompilerGeneratedAttribute::.ctor() = ( 01 00 00 00 )
// Code size      8 (0x8)
.maxstack 8
IL_0000: ldarg.0
IL_0001: ldarg.1
IL_0002: stfld valuetype [mscorlib]System.Decimal Chapter_01.Produc
t::'<ListPrice>k__BackingField'
IL_0007: ret
} // end of method Product::set_ListPrice
```

In *IL_0002* of the preceding code, the IL for the *set* accessor is working with the backing store, *<ListPrice>k__BackingField*, also. So, the point being made here is that you have a simplified syntax with the auto-implemented property feature of C#, but the actual IL being written is equivalent to your having implemented a normal property that works with a private field as its backing store.

In case you either want to be clever or are just curious, the C# compiler won't let you reference the implicit backing store of an auto-implemented property in code. For example, the following won't compile:

```
decimal listPrice = <ListPrice>k__BackingField;
```

The error you receive is that C# doesn't recognize the *<* character. This comes from the fact that C# identifiers must start with an underscore, *@*, or an alphanumeric character. If you have a need to work with the backing store of a property, you should implement a normal property instead of an auto-implemented property. However, you might want to think twice about doing that anyway, because one of the big benefits of having properties is encapsulation, which is broken if you do something like this—even inside of the same class and even more so if you give the backing store more than private access.

Another new feature of C# 3.0, which is also a type member, is partial methods, discussed next.

Partial Methods

This is a tough subject to talk about this early in the book because sometimes it's hard to get the gist of

how a feature works and its intended purpose without seeing it in action. In this case, the *partial method* feature is an optimization for the C# compiler that gives you the option to implement a method in your own class that is defined in another class. This is a feature that is used extensively in LINQ to SQL, which you can learn more about in [Chapter 3](#). However, here I can show you the syntax and mechanics and explain them, and then you'll understand them more fully when you see partial methods used with LINQ to SQL.

How Are Partial Methods Used?

As a demonstration of how a partial method could be used, imagine you have an entity object (which represents some chunk of data you are working with) that you will be modifying. You might be interested in knowing what types of changes are being made to a property of that object. The following code represents a *Product* entity with its *SellEndDate* property instrumented to handle events before and after changing the backing store value:

```
partial class Product
{
    partial void OnSellEndDateChanging(DateTime value);
    partial void OnSellEndDateChanged();

    private DateTime m_sellEndDate;

    public DateTime SellEndDate
    {
        get
        {
            return m_sellEndDate;
        }
        set
        {
            OnSellEndDateChanging(value);
            m_sellEndDate = value;
            OnSellEndDateChanged();
        }
    }
}
```

Any type that contains partial methods must be declared as partial as the preceding *Product* class has been. Before examining the partial methods, look at the *set* accessor on *SellEndDate*. The *OnSellEndDateChanging* and *OnSellEndDateChanged* method calls surround value assignment to the backing store, *m_sellEndDate*. This is the typical way that partial methods are used, providing hooks for your code to be notified of changes to the backing store of a property.

Take a look at the partial methods, *OnSellEndDateChanging* and *OnSellEndDateChanged*, at the top of the *Product* class member list. They each have *partial* modifiers, which all partial methods have. Also, these partial methods don't have an implementation and are called the *defining* methods. Partial methods are implemented in a separate partial class and are called *implementing* methods. The following code shows an additional partial *Product* class that contains a partial method implementation:

```
partial class Product
{
    partial void OnSellEndDateChanging(DateTime value)
    {
        Console.WriteLine(
```

```
        "Changing SellEndDate to " + value);  
    }  
}
```

This *Product* class is partial also, confirming that partial methods must be members of partial types. The implementing method must have the *partial* modifier, as does *OnSellEndDateChanging*.

Earlier, I alluded to the fact that partial methods must be members of partial types, with *types* being the operative term. The *Products* class shows how to implement partial methods in a reference type, but structs, which are value types, have methods too and can implement partial methods. Here's an example:

```
partial struct ProductStruct  
{  
    partial void OnPropertyChanging();  
}
```

The same rules for partial methods in structs apply as they do for class types. The preceding example shows the syntax without all of the other code that uses the partial type, which would be the same as the *Product* class shown earlier.

A Few More Tips

Here are a few more facts for partial methods that you might be curious about:

- They're implicitly private. They must be called within the same class.
- Access modifiers and other modifiers such as *abstract*, *extern*, *new*, *override*, *sealed*, or *virtual* aren't allowed.
- There must be only one defining partial method and one implementing partial method.
- The defining and implementing partial methods can exist in the same partial type. However, this won't do much good because you could have implemented a normal method in that case. Besides, in the context of auto-generated code, as in the *.dbml for LINQ to SQL, you don't want to add code to the auto-generated file because it could easily be deleted by VS 2008. The benefit of partial methods is the fact that you can implement the method outside of the auto-generated code file.

In Case You're Curious

Digging a little deeper into partial methods, you might have noticed that I didn't add an implementing method for *OnSellEndDateChanged*. You aren't required to add implementing methods for defining methods, which is a benefit of the partial method approach. The C# team added partial methods to avoid the overhead associated with other techniques that could have been used to add extensibility to auto-generated code, such as the *.dbml files created for LINQ to SQL (see [Chapter 3](#)).

The overhead that partial methods help you avoid comes from the fact that when you're implementing entities for LINQ to SQL, or other implementations of LINQ, those classes must expose some public members for you to hook into. Among possible mechanisms to do this, delegates, events, and/or interfaces (observer pattern) come to mind. Let's assume that events were chosen for the implementation, and that your code that uses the entities could hook up to events, occurring before and after modification of those entities, to be notified of changes. This means that you would have an entity with two events for each property and then another event for the entity as a whole. To quantify this more fully and bring it home, think about how many LINQ to SQL entities exist in a database. It might not be a problem for a small database, but the larger the database schema becomes, the more events are defined to support this extensibility.

All of these events add overhead to the application, which is likely to be a problem in many scenarios. Again, I simply used events for demonstration purposes, but other implementations would have similar

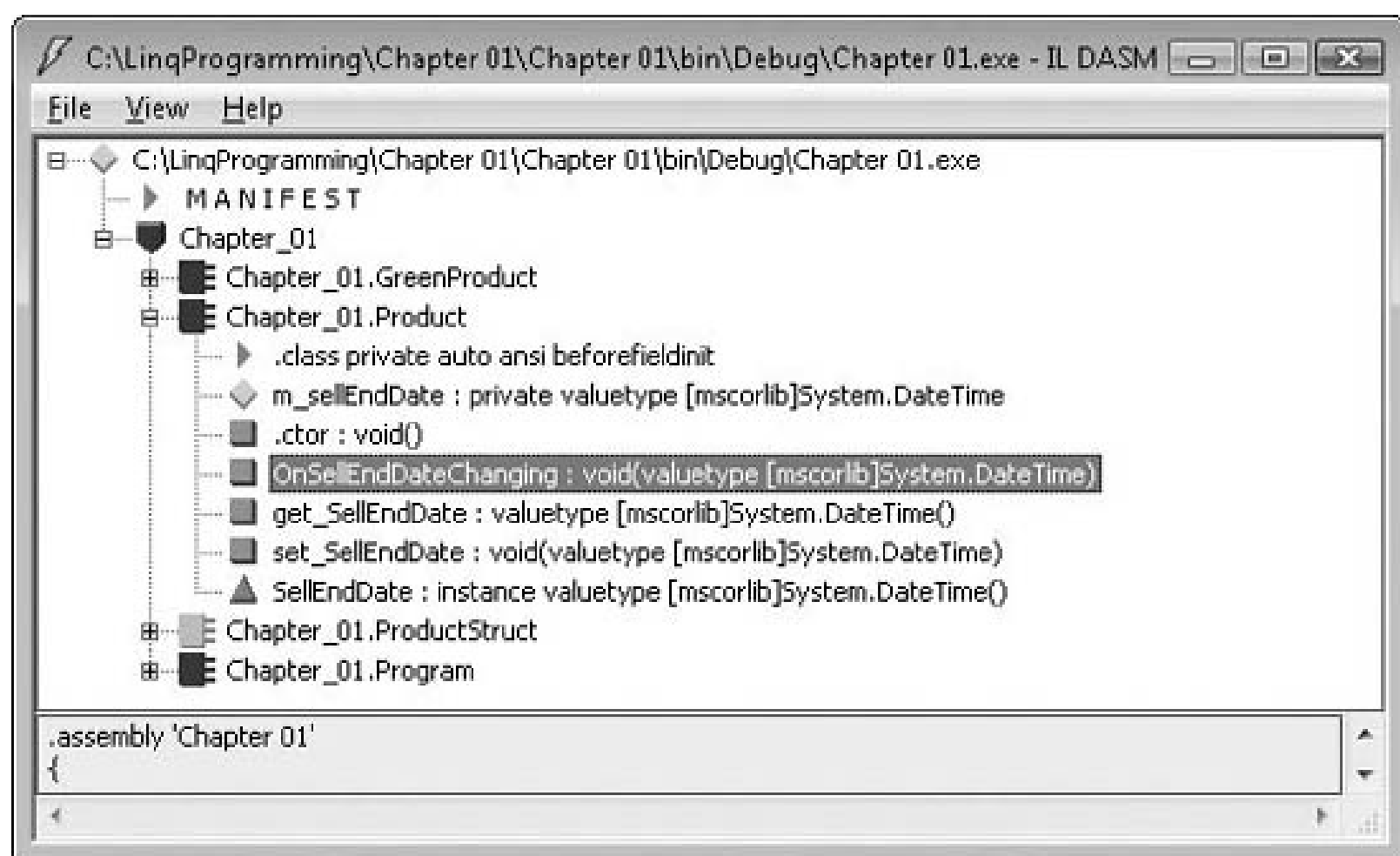
impact. Partial methods are an answer to the problem because the C# compiler does not produce IL for a defining method if there is not an implementing method. To see that this is true, you can open ILDASM and verify which members of the *Product* class appear, as shown in [Figure 1-2](#).

I've opened the *Product* class in ILDASM, shown in [Figure 1-2](#). You can see that the *OnSellEndDateChanging* method appears. This is because the second partial *Product* class contains the implementing method for *OnSellEndDateChanging*. However, an implementing method for *OnSellEndDateChanged* wasn't created in the first partial *Product* class. Regardless of whether the first partial *Product* class contains a defining method for *OnSellEndDateChanged*, the C# compiler will not generate IL for that method, because the implementing method of *OnSellEndDateChanged* doesn't exist. The IL is only generated when some partial class implements the method.

This feature of the C# compiler recognizes that hooking up to entity notifications is the exception, and not the rule. Therefore, you get an optimization of smaller object size by leaving out unimplemented partial methods. You'll learn more about entities and partial methods later in [Chapter 3](#) and [Chapter 5](#).

The next feature to discuss, extension methods, allows you to add members to an existing type.

FIGURE 1-2 Partial methods minimizing overhead in an Assembly



Extension Methods

The new C# 3.0 extension methods feature is at the heart of LINQ, allowing implementation of custom LINQ providers, yet retaining the same query syntax within the programming language. [Chapter 9](#) goes into greater detail about how extension methods support creation of custom LINQ providers. What you'll see here is the basic mechanics of extension methods and how you can use them to build libraries that

extend other types as you need.

Creating Extension Methods

In a nutshell, an *extension method* is a static method in a static class that refers to the type it extends, enabling you to call the extension method through an extended type instance. To show you how extension methods work, I’m going to extend the C# *decimal* type. As you know, *decimal* is a value type, and there is no normal way to derive from or extend *decimal* using features from versions of C# earlier than 3.0. The specific extension method is *GetDollars*, which assumes that the value represents a financial value, and the caller only wants the whole number portion of the value. Here is the code for the *GetDollars* extension method, followed by an in-depth explanation of what it means and how it works:

```
public static class LinqProgrammingExtensions
{
    public static decimal GetDollars(
        this decimal amount, // extended type
        bool shouldRound) // input parameter
    {
        if (shouldRound)
        {
            return (int)(amount + .5m);
        }
        else
        {
            return (int)amount;
        }
    }
}
```

Another requirement is that the extension method must have a first parameter that identifies the type being extended. In the preceding example, *GetDollars* extends the *decimal* type. You can tell this because the extended type is the first parameter, and its type is modified with *this*. The name of the extended type is *amount*, which the *GetDollars* extension method uses to access the instance that invoked it.

The second and subsequent parameters, if any, are passed by the calling code, but the extended type instance is passed implicitly as you will see shortly.

Both the class and the method preceding are static, which is required. Although the class is named *LinqProgrammingExtensions*, you won’t use that class name in most cases. Instead, you’ll use the extension method with an instance of the type defined by the first parameter, decorated with the *this* modifier. The next section explains how to use the extension method.

Using Extension Methods

To show you how the *GetDollars* extension method can be used, the following code invokes *GetDollars* and uses the value that it returns:

```
decimal amount = 55.55m;
Console.WriteLine(
    "Dollars: " + amount.GetDollars(true));
```

The *amount* variable is type *decimal*, containing both whole and fractional number parts. Notice how the *GetDollars* method is called on the *amount* instance, using the instance member (dot) operator. This causes *amount* to be passed implicitly as the first parameter to the *GetDollars* method. Also, notice the

true value being passed as the first argument. Since *amount* is the first parameter to *GetDollars*, the first argument becomes the second parameter to *GetDollars*, which is *shouldRound*. If there were subsequent parameters, they would follow *shouldRound*.

Extension Method Precedence

The next concern you might have about extension methods is about the precedence when a type method matches an extension method. The answer is that the type method has precedence. To demonstrate, here's an extension method that extends the string class:

```
public static class LinqProgrammingExtensions
{
    public static int CompareTo(
        this string origStr,
        string compareStr)
    {
        string orig = origStr.ToUpper();
        string comp = compareStr.ToUpper();

        return orig.CompareTo(comp);
    }
}
```

As in the previous example, both the class and the method are *static*, and the first parameter of *CompareTo* has a *this* modifier and the type being extended, which is *string*. As you know, the string class already has a *CompareTo* method. However, this *CompareTo* is different in that it is case insensitive, but the *CompareTo* method of the string class is case sensitive. The following code tests this scenario, demonstrating that *CompareTo* of string will be called and not the extension method:

```
string name = "joe";
Console.WriteLine(
    "joe == Joe: " +
    name.CompareTo("Joe"));
```

In the preceding example, *name* is lowercase "joe". The *Console.WriteLine* statement calls *CompareTo* on *name*, with a capitalized "Joe". The result of the call to *CompareTo* is -1 (the internal value of "true"), which is consistent with the implementation of *CompareTo* on the string class. If you wanted the behavior of the extension method, you can call it through its type, *LinqProgrammingExtensions*:

```
Console.WriteLine(
    "joe == Joe: " +
    LinqProgrammingExtensions
        .CompareTo(name, "Joe"));
```

This time, the result is 0, indicating that the case-insensitive comparison of the extension method was invoked. Notice that extension methods, when called through their type, allow you to explicitly pass an instance of the extended type as the first parameter. So, in the case of the *CompareTo* extension method, the *name* argument is passed as the *origStr* parameter and the "Joe" argument is passed as the *compareStr* parameter.

Invoked via the type, as shown earlier, extension methods behave like any other static method.

In Case You're Curious

If you're curious about how C# implements extension methods, it might surprise you that the resulting IL is simply a call to a static method. Going back to the *GetDollars* extension method, discussed earlier, I can show you what the C# compiler does when it encounters an extension method. Behind the scenes, the C# compiler resolves which method should be called and will execute the static extension method directly, as shown in the IL for the *Main* method that follows, where *GetDollars* was called:

```
.method private hidebysig static void Main(
    string[] args) cil managed
{
    .entrypoint
    // Code size      45 (0x2d)
    .maxstack 6
    .locals init ([0] valuetype
        [mscorlib]System.Decimal amount)
    IL_0000: nop
    IL_0001: ldc.i4      0x15b3
    IL_0006: ldc.i4.0
    IL_0007: ldc.i4.0
    IL_0008: ldc.i4.0
    IL_0009: ldc.i4.2
    IL_000a: newobj      instance void
        [mscorlib]System.Decimal::.ctor(int32,
                                        int32,
                                        int32,
                                        bool,
                                        uint8)
    IL_000f: stloc.0
    IL_0010: ldstr      "Dollars: "
    IL_0015: ldloc.0
    IL_0016: ldc.i4.1
    IL_0017: call       valuetype
        [mscorlib]System.Decimal
        Chapter_01.LinqProgrammingExtensions::GetDollars(
            valuetype [mscorlib]System.Decimal,
            bool)
    IL_001c: box        [mscorlib]System.Decimal
    IL_0021: call       string
        [mscorlib]
        System.String::Concat(object, object)
    IL_0026: call       void
        [mscorlib]System.Console::WriteLine(string)
    IL_002b: nop
    IL_002c: ret
} // end of method Program::Main
```

The important line in the preceding code is *IL_0017*. Notice how C# emits code that calls *GetDollars* on the *LinqProgrammingExtensions* type directly. This is all a C# programming language feature, and the CLR didn't need to change to accommodate it.

If you recall, C# extension methods specify the type to extend by modifying the first parameter with *this*. However, the same syntax doesn't appear in the generated IL, as shown in [Figure 1-3](#).

As shown in [Figure 1-3](#), the *GetDollars* method is just a normal static method. However, there is metadata that indicates that *GetDollars* is an extension method. The following IL for the *GetDollars* method shows that it will be decorated with an *ExtensionAttribute*:

```
.method public hidebysig static valuetype
```

```

[mscorlib]System.Decimal
GetDollars(
    valuetype [mscorlib]System.Decimal amount,
    bool shouldRound) cil managed
{
    .custom instance void
        [System.Core]
        System.Runtime.CompilerServices
.ExtensionAttribute::.ctor()
        = ( 01 00 00 00 )
    // Code size      56 (0x38)
    .maxstack 7
    .locals init ([0] valuetype
        [mscorlib]System.Decimal CS$1$0000,
        [1] bool CS$4$0001)
    IL_0000: nop
    IL_0001: ldarg.1
    IL_0002: ldc.i4.0
    IL_0003: ceq
    IL_0005: stloc.1
    IL_0006: ldloc.1
    IL_0007: brtrue.s    IL_0027
    IL_0009: nop
    IL_000a: ldarg.0
    IL_000b: ldc.i4.5
    IL_000c: ldc.i4.0
    IL_000d: ldc.i4.0
    IL_000e: ldc.i4.0
    IL_000f: ldc.i4.1
    IL_0010: newobj     instance void
        [mscorlib]System.Decimal::.ctor(int32,
                                        int32,
                                        int32,
                                        bool,
                                        uint8)
    IL_0015: call      valuetype
        [mscorlib]System.Decimal
        [mscorlib]System.Decimal::op_Addition(
            valuetype
[mscorlib]System.Decimal,
            valuetype [mscorlib]System.Decimal)
    IL_001a: call      int32
        [mscorlib]System.Decimal::op_Explicit(
            valuetype
[mscorlib]System.Decimal)
    IL_001f: call      valuetype
        [mscorlib]System.Decimal
        [mscorlib]
System.Decimal::op_Implicit(int32)
    IL_0024: stloc.0
    IL_0025: br.s      IL_0036
    IL_0027: nop
    IL_0028: ldarg.0
    IL_0029: call      int32
        [mscorlib]System.Decimal::op_Explicit(
            valuetype [mscorlib]System.Decimal)
    IL_002e: call      valuetype
        [mscorlib]System.Decimal
        [mscorlib]System.Decimal::op_Implicit(int32)
    IL_0033: stloc.0

```

```

IL_0034: br.s IL_0036
IL_0036: ldloc.0
IL_0037: ret
} // end of method LinqProgrammingExtensions::GetDollars

```

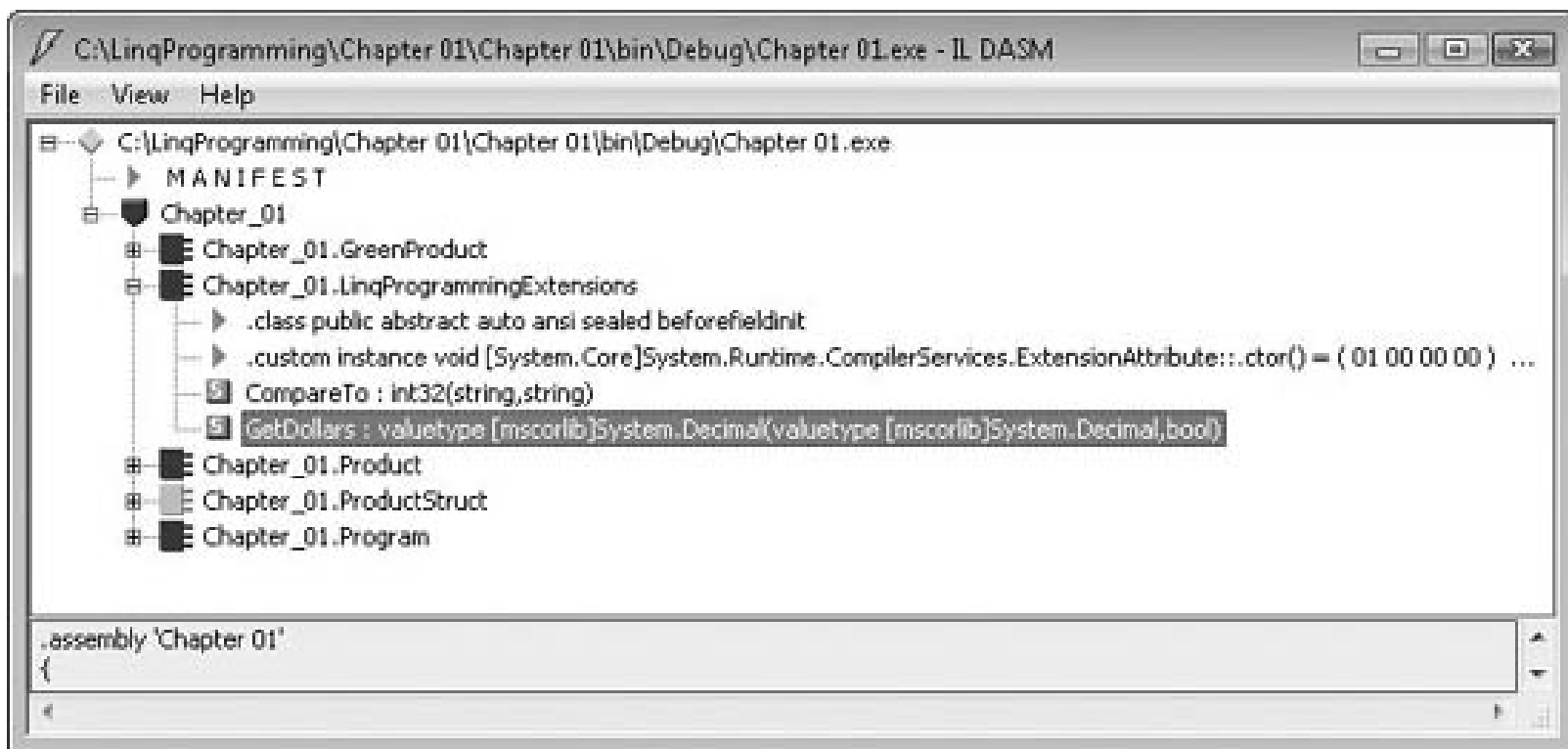


FIGURE 1-3 Extension method IL declaration

In the preceding code, you can see the *ExtensionAttribute* line between the opening curly brace and the line that reads `.maxstack 7`. One useful result of this could be that if you were using reflection, searching for extension methods, you could look for the *ExtensionAttribute* attribute decorating a static method in a static class.

Given this last tidbit of information, that an extension method in IL is decorated with the *ExtensionAttribute* attribute, you might want to be clever and try to use *ExtensionAttribute* instead of the *this* modifier, like so:

```

[System.Runtime.CompilerServices.Extension]
public static int CompareTo(
    string origStr,
    string compareStr)
{
    string orig = origStr.ToUpper();
    string comp = compareStr.ToUpper();

    return orig.CompareTo(comp);
}

```

However, your mission would be thwarted by the C# compiler, which would give you the error message “Do not use ‘System.Runtime.CompilerServices.ExtensionAttribute’. Use the ‘this’ keyword instead.”

Extension methods were added to C# 3.0 primarily to support LINQ. You’ll see this more clearly in [Chapter 9](#). Another use of extension methods is to build your own reusable libraries. You can find

extension method libraries at www.codeplex.com, to see what other people are doing in this area.

Another way to extend types in C# 3.0 is via constructors, using object and collection initializers, which you'll learn about in the next section.

Object Initializers

When designing a type, you typically consider how it should be initialized by calling code, especially if it is reusable or part of a third-party library. As your type takes on more properties, the complexity of initializing it increases. You must either create many constructors with permutations of allowable values, or provide what you predict are the most common, and then allow callers to instantiate and then set properties. With object initializers, you don't have to worry about these issues as much anymore. Additionally, object initializers make it easy for calling code to instantiate a type.

Using Object Initializers

You can use *object initializer* syntax to set the public properties (or public fields) of a type during instantiation. For example, the following code uses an object initializer to instantiate a *Product* class.

```
Product prodInit =  
    new Product  
    {  
        Name = "widget",  
        ProductID = 1,  
        SellStartDate = DateTime.Now  
    };
```

The appropriate use of whitespace can make code like the preceding much easier to read. You can see that *prodInit* is instantiated as a *Product* with the specified values. The curly braces surround a comma-separated list of name/value pairs, which is the object initializer syntax. The properties *Name*, *ProductID*, and *SellStartDate* are initialized to the specified values. Pressing CTRL-spacebar in VS 2008 shows a list of available properties. The preceding code is equivalent to the following in traditional object instantiation:

```
Product prodTraditional = new Product();  
  
prodTraditional.Name = "widget";  
prodTraditional.ProductID = 1;  
prodTraditional.SellStartDate = DateTime.Now;
```

Some would say that the object initializer syntax is easier to code and read.

In Case You're Curious

Of course, you don't have to take my word for it, the proof is in the IL. The *Main* method that follows shows the results of using object initializer syntax and then following it with the equivalent traditional object instantiation syntax:

```
.method private hidebysig static void Main(string[] args) cil managed  
{  
    .entrypoint  
    // Code size      80 (0x50)  
    .maxstack 2  
    .locals init ([0] class Chapter_01.Product prodInit,
```

```

[1] class Chapter_01.Product prodTraditional,
[2] class Chapter_01.Product '<>g__initLocal0')
IL_0000: nop
IL_0001: newobj     instance void Chapter_01.Product::.ctor()
IL_0006: stloc.2
IL_0007: ldloc.2
IL_0008: ldstr      “widget”
IL_000d: callvirt   instance void
    Chapter_01.Product::set_Name(string)
IL_0012: nop
IL_0013: ldloc.2
IL_0014: ldc.i4.1
IL_0015: callvirt   instance void
    Chapter_01.Product::set_ProductID(int32)
IL_001a: nop
IL_001b: ldloc.2
IL_001c: call       valuetype
    [mscorlib]System.DateTime
    [mscorlib]System.DateTime::get_Now()
IL_0021: callvirt instance void
    Chapter_01.Product::set_SellStartDate(
        valuetype [mscorlib]System.DateTime)
IL_0026: nop
IL_0027: ldloc.2
IL_0028: stloc.0
IL_0029: newobj     instance void
    Chapter_01.Product::.ctor()
IL_002e: stloc.1
IL_002f: ldloc.1
IL_0030: ldstr      “widget”
IL_0035: callvirt instance void
    Chapter_01.Product::set_Name(string)
IL_003a: nop
IL_003b: ldloc.1
IL_003c: ldc.i4.2
IL_003d: callvirt   instance void
    Chapter_01.Product::set_ProductID(int32)
IL_0042: nop
IL_0043: ldloc.1
IL_0044: call       valuetype
    [mscorlib]System.DateTime
    [mscorlib]System.DateTime::get_Now()
IL_0049: callvirt instance void
    Chapter_01.Product::set_SellStartDate(
        valuetype [mscorlib]System.DateTime)
IL_004e: nop
IL_004f: ret
} // end of method Program::Main

```

Looking at the preceding IL, you can see that the statements from *IL_0001* to *IL_0021*, which are for the object initializer, are nearly identical to the statements from *IL_0029* to *IL_0049*, which are for the traditional object instantiation.

You can leverage knowing that object initializers call the default constructor when they aren’t passed parameters to restrict the use of object initializers. For example, what if you needed to guarantee that specific values were passed when your type is instantiated? By default, object initializers allow callers to set any properties they want, even leaving some out if they want. This could lead to your type being initialized improperly without the proper values and operating incorrectly. One thing you can do is make

your default constructor private. For example, if the *Product* class had the following constructors:

```
private Product() {}

public Product(string name, decimal price)
{
    Name = name;
    ListPrice = price;
}
```

Since the default constructor is private, the earlier object initializer code for *Product* wouldn't compile. Callers would then be required to use the constructor overload with parameters, as shown next:

```
Product prodTraditional =
    new Product("widget", 55.55m);

prodTraditional.ProductID = 1;
prodTraditional.SellStartDate = DateTime.Now;
```

Of course, that doesn't completely preclude the use of object initializers because the exact same results can be achieved with the following object initializer implementation:

```
Product prodInit =
    new Product("widget", 55.55m)
    {
        ProductID = 1,
        SellStartDate = DateTime.Now
    };
```

Notice that the *Product* instantiation can use the overloaded constructor and still tack on object initializer syntax. This gives you the best of both worlds—the ability to ensure your type is initialized properly and the convenience of object initializer syntax.

Collection Initializers

Related to object initializers is a new feature called *collection initializers*, which make it easy to initialize a collection within a single statement.

Implementing Collection Initializers

Once I liked using array initialization syntax for adding parameters to *SqlCommand* objects, like this:

```
cmd.Parameters.AddRange(
    new SqlParameter[]
    {
        new SqlParameter("@Parm1", val1),
        new SqlParameter("@Parm2", val2),
        new SqlParameter("@Parm3", val3)
    }
);
```

Assuming that *cmd* is type *SqlCommand*, and *val1*, *val2*, and *val3* are parameters passed to my data access layer method, the preceding code would populate the *Parameters* collection for *cmd*. Of course, moving forward you'll want to use LINQ for such things, but this syntax has been very convenient and still remains useful today. C# 3.0 extends the array initializer of earlier C# versions and allows you to use it with collections in a feature known as collection initializers. Here's an example of a collection of

Products:

```
List<Product> products =  
    new List<Product>  
    {  
        new Product  
        {  
            Name = “wadget”,  
            ProductID = 1,  
        },  
        new Product  
        {  
            Name = “widget”,  
            ProductID = 2,  
        },  
        new Product  
        {  
            Name = “wodget”,  
            ProductID = 3,  
        }  
    };
```

Collection initialization syntax is very much like object initialization syntax, except collections initialize with instances of the same type, rather than setting properties. One of the things you might notice is that object initialization can become verbose, making it difficult to look for VS 2008 code folding so it doesn’t take up too much screen real-estate, but object initializers aren’t included in code folding. However, you can achieve a similar result by surrounding initialization code with *#region/#endregion* directives.

Collection Initializers for Dictionary Collections

You can use the same syntax in any *IList*- or *IList<T>*-derived collections. However, *IDictionary<T>* collections are different. Here’s an example of how you could use an object initializer for a *Dictionary* collection:

```
Dictionary<int, Product> keyedProducts =  
    new Dictionary<int, Product>  
    {  
        {  
            1,  
            new Product  
            {  
                ProductID = 1,  
                Name = “wadget”  
            }  
        },  
        {  
            2,  
            new Product  
            {  
                ProductID = 2,  
                Name = “widget”  
            }  
        },  
        {  
            3,  
            new Product
```

```

    {
        ProductID = 3,
        Name = "wodget"
    }
},
};

```

To understand how the collection initializer syntax for Dictionaries, and other *IDictionary*-derived types work, remember that they really contain *KeyValuePair<TKey, TValue>*, meaning that you must supply both the key and the value. In reality, the C# compiler will produce IL that calls the *Add* method of the Dictionary collection for each of the provided key/value pairs. This is an important point because syntax errors will result in a compiler error on the *Add* method, which is not part of the code you wrote, making the error message confusing if you don't know this.

Also, the extra set of curly braces surrounding key/value pairs is required.

I'll skip the IL demonstration in this section because the previous section on object initializers can be applied to collection initializers.

Lambda Expressions

A *lambda expression* (*lambda*) is a delegate without a name. That should sound familiar because C# 2.0 added a new feature called anonymous methods that did the same thing, allowing you to execute a block of code without having to create a method. There are two primary differences between anonymous methods and lambdas: a lambda has shorter syntax, and it can be treated as data as well as executable code. You will learn more about the concept of lambdas being used in the form of data in [Chapter 8](#), when we go over expression trees. For this chapter, I'll show you the syntax of a lambda and how it can be used anywhere that an anonymous method can be used.

How to Use a Lambda Expression

Lambdas excel at tasks that require a quick statement because of their terse syntax. For example, the List collection has a set of methods that take either *Action* or *Predicate* generic delegates. If you wanted to extract all of the *Products* from a list that met specific criteria, you could use the *Find* method. The following two examples show how to do this with both an anonymous method and a lambda.

Here's the anonymous method example:

```

Product widget = products.Find(
    delegate(Product product)
    {
        return product.Name == "widget";
    });

```

And here's the lambda example:

```

widget = products.Find(
    product => product.Name == "widget" );

```

Both the anonymous method and lambda are logically equivalent, but a glance tells you that the lambda has less syntax. The extra code associated with the delegate keyword, parenthesis, and parameter type aren't necessary with the preceding lambda example. Neither is the block defined by the curly braces. The statement itself doesn't need a *return* keyword; return of the results of a single statement is implied, as is

the end of the statement that doesn't need a semicolon. It's short and sweet—run this lambda on each item in the list. The `=>` operator, which is often referred to as “goes to,” “such that,” or “where,” separates the parameter list from the statement or optional block.

Replacing Anonymous Methods and Delegates

That was a typical implementation of the lambda. Another way to use a lambda is anywhere you could use a normal delegate. The next example shows how a lambda can be bloated to look almost like an anonymous method. It hooks up a handler to the Click event of a Button:

```
Button btnClickMe =
    new Button { Name = "My Button" };

btnClickMe.Click +=
    (object sender, EventArgs e) =>
    {
        Button btnFake = sender as Button;
        MessageBox.Show(
            "Fake click on " + btnFake.Name);
    };

btnClickMe.PerformClick();
```

This time, I set up a button and simulated a click on it. I also hooked up a lambda to the button's *Click* event. Notice that this lambda has *EventHandler* delegate parameters. The parameter types were optional, but I put them there so you could see them being used; they are available in case you might need to resolve any ambiguity. Here's what the lambda looks like without the parameter types, which is also valid:

```
btnClickMe.Click +=
    (sender, e) =>
    {
        Button btnFake = sender as Button;
        MessageBox.Show(
            "Fake click on " + btnFake.Name);
    };
```

Because this lambda has multiple statements, the block with curly braces and semicolons on each statement is required. If the lambda were to return a value, the return value would be necessary too—again because of multiple statements.

A Few Tips on Lambda Expressions

Here are a few more facts about lambdas:

- They can have an empty parameter list, as in `() => statement`.
- They can be assigned to any matching delegate if parameter and return types match.
- .NET has new *Func<>* generic delegate types that are used extensively with LINQ and lambdas.

You'll see many uses of lambdas throughout this book because they are an integral part of making LINQ work. When you get to [Chapter 8](#), where I show you how lambdas can become data in expression trees, you'll see this even more clearly.

Implicitly Typed Local Variables

The term *implicitly typed* means that you don't have to specify the type of a local variable because the C#

compiler can figure it out—when you use the *var* keyword. To see one of the benefits, here’s one way we’ve typically instantiated classes:

```
Product prod = new Product();
```

If you are bothered by the redundancy of specifying the type, *Product*, twice in the same statement, you aren’t alone. For cases like this, the new *var* keyword is helpful. Here’s how to rewrite the preceding statement with *var*:

```
var prod = new Product();
```

Just as with auto-implemented properties and other features, the immediate benefit is less typing. Before you get too nervous about the *prod* variable’s type, I’ll share a few factoids that might put you at ease. A *var* is not a variant like you see in languages like JavaScript or VBA—it is strongly typed. Once you instantiate a *var* variable, it has the type of the instance assigned to it. In the preceding example, *prod* is of type *Product*. It is also strongly typed because you can’t assign anything to it other than a reference of type *Product*. A *var*’s type is determined at compile time. It follows that a *var* must be declared and assigned/instantiated in the same statement. For example, the following statement is illegal:

```
var prod;
```

The preceding statement results in the compiler error “Implicitly-typed local variables must be initialized.” In addition to being strongly typed, implicitly typed local variables are local, meaning that they must be members of a method, operator, conversion operator, property accessor, indexer accessor, event accessor member, or a block (between curly braces). In other words, the compiler prevents a *var* from being declared in a nonlocal scope.

One of the main reasons that *var* was added to C# was to support LINQ. In the next section, you’ll finally get to see a little bit of query syntax and see why both implicitly typed local variables and anonymous types were added to the language.

Query Syntax

One of the C# language features you’ll see a lot in this book is query syntax. It is the prominent feature of C# that makes it easy to perform LINQ queries, is simple to code, and is easy to read. Since this book contains entire chapters dedicated to query syntax, I’ll just give you a quick look here so we can quickly finish up and move to the next chapter, which expands on query syntax much more.

A First Look at a Very Simple Query

Here is a LINQ query with C# query syntax:

```
IEnumerable<Product> prodSequence =  
    from currentProduct in products  
    where currentProduct.Name != “widget”  
    select currentProduct;
```

On first seeing the preceding code, you should get a sense that some query is returning a collection of *Product*. The query, defined by *from*, *where*, and *select* clauses is very similar to Structured Query Language (SQL). Depending on your level of comfort with SQL, you should also have a feeling that this is going to be very easy. For the most part, it is quite easy, and I’ll be showing you tricks throughout this book to help you get past any of the parts that might be more challenging. For all its simplicity, though,

LINQ is very powerful.

Consider the preceding statement. The *from* clause identifies the collection, *products*, which was defined earlier in the collection initializer demo. The *currentProduct* is a variable that holds each of the items as they are referenced in *products*. You can see how we use *currentProduct* in the *where* clause, which allows filtering the results. In this case, we don't want the product that is named "widget." The *select* clause will identify the item that will be added to the *prodSequence* collection.

Using Sequences from a Query

Once you have a collection, like *prodSequence*, you can do many things with it. You can return it from a method, bind it to controls, or perform further processing. Essentially, use it any way you would use a collection of objects that you obtain from any other data store. The following loop demonstrates how you can access each of the *Product* instances in *prodSequence*:

```
foreach (var item in prodSequence)
{
    Console.WriteLine(
        "ID: {0}, Name: {1}",
        item.ProductID,
        item.Name);
}
```

The objects in *prodSequence* are type *Product*, so they are used in the preceding loop to extract and print properties. So, the collection you get back from a LINQ query is a .NET collection that you can use just like any other collection. In later chapters, you'll learn that this collection isn't populated until it's used somewhere, such as in a *foreach* loop. This is an optimization called *deferred execution* that will be covered in more detail later.

In the next chapter, you'll learn all of the interesting details of how query syntax works as you learn about LINQ to Objects. Before leaving this chapter though, I would like to tie in the motivation for implicitly typed local variables and introduce you to the final new C# 3.0 feature, anonymous types.

Anonymous Types

In the previous example on query syntax, the *select* clause grabbed the whole object for each item in the collection. C# has a feature, anonymous types, that enables you to extract a subset of each object returned. You'll see a very simple anonymous type in this section, but they have other very important functions that you'll learn about in later chapters when I explain topics such as grouping and joining.

A Simple Anonymous Type

Here's an example of using anonymous types to grab only the *ProductID* and *Name* fields from the *products* collection:

```
var prodAnon =
    from currentProduct in products
    where currentProduct.Name != "widget"
    select
        new
        {
            Name = currentProduct.Name,
            ID = currentProduct.ProductID
        }
```

```
};
```

You need to know about two parts of the preceding query: the use of the implicitly typed local variable and the anonymous type in the *select* clause. Looking first at the *select* clause, you can see the *new* operator followed by a block with two property assignments. You might have noticed that *new* was not followed by an identifier, or type name. This is intentional because it creates a type that doesn't have a name, leading to the reason why this feature is called an anonymous type. The block identifies a set of properties that this new type will have, *Name* and *ID*. The values of *Name* and *ID* come from the *Name* and *ProductID*, respectively, of the current object being evaluated, *currentProduct*. This gives you a collection of anonymous type objects that have *Name* and *ID* properties.

Because the result of this query is a collection of anonymous types, it is impossible to specify a type for the result, *prodAnon*. Is it *IEnumerable<T>* or what? You don't know. Therefore, you must define *prodAnon* as an implicitly typed local variable. This is the primary reason why implicitly typed local variables were added to C#.

Anonymous Types Are Strongly Typed

As you learned earlier, variables declared with *var* are strongly typed, and the same holds true for anonymous types. For C#, the type of *prodAnon* is an *IEnumerable<T>* of a type that has a *Name* property of type string and an *ID* property of type int. Here's an example that demonstrates strong typing:

```
var anonList = prodAnon.ToList();

anonList.Add(
    new
    {
        Name = "wedget",
        ID = 4
    });
```

The *ToList* method is an extension method for the *IEnumerable<T>* that allows you to convert *IEnumerable<T>* collections to *List<T>* collections, which was necessary in this case to allow manipulation of collection members.

The call to *Add* works perfectly because the anonymous type has a property that is *Name* of type *string*, followed by a property *ID* of type *int*. Order matters, so the following example won't compile:

```
anonList.Add(
    new
    {
        ID = 4,
        Name = "wedget"
    });
```

With *ID* and *Name* positions switched, the preceding type is not the same as the members of *anonList*. More obvious is the following example where the type isn't even close:

```
anonList.Add(
    new
    {
        ListPrice = 13m
    });
```

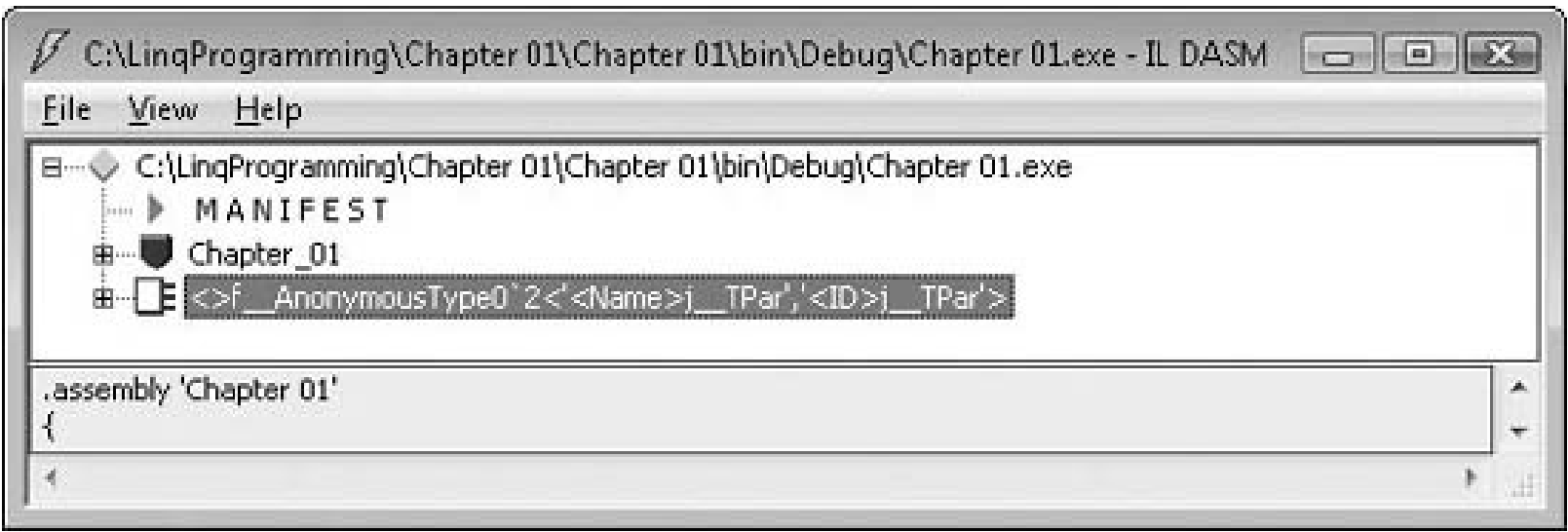
The preceding example won't compile because the property name, type, and number of properties is different.

In Case You're Curious

Although you'll never know the name of the anonymous type in C# code, you might be curious to know that the type really does have a name in IL. [Figure 1-4](#) shows what the name of the anonymous type is for the objects added to the *prodAnon* collection.

As shown in [Figure 1-4](#), the anonymous type name is `<>f__AnonymousType0`2<'<Name>j__TPar', '<ID>j__TPar'>`. That's pretty ugly and won't compile in C# code for the same reason that IL backing store names, as explained earlier, won't compile—it's an invalid C# identifier. The IL for the method that uses this is pretty long, so I'll spare you all of the irrelevant details and only show the part that uses the anonymous type.

FIGURE 1-4 Anonymous type name in IL



```
IL_011e: nop
IL_011f: ldloc.1
IL_0120: call     class
[mscorlib]System.Collections.Generic.List`1<!!0>
[System.Core]System.Linq.Enumerable::ToList<
  class '<>f__AnonymousType0`2'<string,int32>
>(<
  class
[System.Collections.Generic.IEnumerable`1<!!0>
)
IL_0125: stloc.3
IL_0126: ldloc.3
IL_0127: ldstr    "wedget"
IL_012c: ldc.i4.4
IL_012d: newobj   instance void class
  '<>f__AnonymousType0`2'<string,int32>::ctor(!0,
!1)
IL_0132: callvirt instance void class
[System.Collections.Generic.List`1<
  class '<>f__AnonymousType0`2'<string,int32>
>::Add(!0)
```

```
IL_0137: nop  
IL_0138: ret
```

At *IL_0120*, the code calls *ToList* on *prodAnon*, giving us *anonList*. You can see that it is creating a list of the anonymous type, whose mangled name you see in [Figure 1-4](#). Then on *IL_012d*, you can see a new instance of the anonymous type being added to *prodAnon*. If you recall from the C# preceding code, the only thing making this anonymous type the same as the type members of *prodAnon* is the property list. C# produced IL that recognizes that this is an identical property list and uses the same anonymous type.

While it's cool to look at IL and see what's happening, the real point here is to understand how anonymous types are strongly typed. Further, features such as implicitly typed local variables and anonymous types can make you look twice when you first learn about them, but it's important to know that you still have the protections of the C# compiler to keep you in the strongly typed world that has made C# programmers productive over the last several years.

Summary

Before jumping into LINQ, it's helpful to have a background in its foundations. In doing so, you learned about several features of C# 3.0 and how they contribute to LINQ. I went into depth on several of the features, even demonstrating some of their internal behavior by showing you the IL the C# compiler produces. Throughout the rest of this book, you'll see many more examples of each of these features that will enhance your understanding of how they're used, especially to support LINQ.

One of the features, query syntax, is one of the more prominent C# language features that you'll use with LINQ. In particular, you can use C# query syntax to query objects, which is called LINQ to Objects. The next chapter takes you deeper into LINQ to Objects and the many clauses you can use with query syntax.

CHAPTER 2

Using LINQ to Objects

The ability to query collections in memory, which is what LINQ to Objects allows you to do, can be very powerful. Anything that you could implement a loop on to read values is a potential candidate for a LINQ to Objects implementation. Collections are the obvious choice for queries, and you'll see many examples in this chapter, but any source that you can pull an *IEnumerable<T>* from is also eligible. I'll also show you how to query collections that aren't *IEnumerable<T>*, opening the door to more objects to query with LINQ.

All of the material you'll see in this chapter uses C# query syntax. In addition to query syntax, numerous extension methods perform the same task or add new capabilities for performing LINQ queries. In [Chapter 9](#), I do a deep dive on extension methods and show you how C# query syntax translates into a subset of LINQ extension methods. Additionally, the Appendix contains a complete reference with multiple examples of all available LINQ extension methods.

In addition to learning what you can query, you'll learn about all of the clauses supported by C# in query syntax. In the last chapter, you saw only a tiny sampling of querying a collection, but now you'll see how to control what information you see, which objects appear, the order of the objects, and a few query operators that you'll probably use frequently. The first section continues where the last chapter left off, with query syntax and selecting data.

LINQ to Objects Essentials

It's helpful to have an idea of what LINQ to Objects does for you, what you can query, and how to set up your development environment before jumping into code. The rest of the chapter is all about LINQ to Objects, so I'll use the shortened "LINQ" to refer to it. The following sections will help you get ready to learn LINQ.

LINQ to Objects Benefits

If you've read any of the latest news, LINQ is advertised as a great way to write code to work with data sources such as SQL Server or XML data. This is true, but an often understated benefit of LINQ is that you can query collections with it too. This is a powerful capability in your programming arsenal. Compare the ability to make a SQL-like query against an object with the ability to use loops; it's the difference between a declarative approach and an imperative approach.

To demonstrate this point, here's the same query syntax example from [Chapter 1](#). It obtains *Product* instances from the *products* collection whose *Name* property is not *widget*:

```
IEnumerable<Product> prodSequence =  
    from currentProduct in products  
    where currentProduct.Name != "widget"  
    select currentProduct;
```

The preceding example is easily understandable by anyone familiar with SQL. It is a single statement, which is simple. Query syntax is part of the C# programming language, giving you compile-time type safety. The approach is also *declarative*, meaning that you tell the code what to do, but not how to do it.

You can benefit from a declarative style of programming in the areas of simplicity, performance, and application reliability. If you tell the application to do something but not how to do it, you've potentially

relieved yourself of much of the complexity associated with the details of how the code will run. Modern database management systems (DBMSs) have done this for years—you write a SQL query, and the database run-time engine optimizes it based on environmental factors such as indexes, precompilation, available hardware resources, and more. Because you aren’t specifying details, the underlying LINQ implementation can take advantage of that flexibility to ensure that the code executes most efficiently. Admittedly, this is not necessarily the case today, but, just like SQL running on a DBMS, the potential is there for tomorrow. From a reliability perspective, the functional programming style tends to promote immutability in objects, which is a benefit in concurrent scenarios. Essentially, if a variable can’t change, then it is safer for multithreading. These ideas require a paradigm shift for programmers who have used imperative programming styles for years, whether procedural or object-oriented.

Considering the preceding LINQ example, here’s an equivalent example, written with a loop:

```
List<Product> prodSequence = new List<Product>();

for (int i = 0; i < products.Count; i++)
{
    if (products[i].Name == “widget”)
    {
        prodSequence.Add(products[i]);
    }
}
```

As you know, there are multiple ways of writing this loop, but the important aspect of this example is the imperative style of how this loop works. It explicitly instantiates a new *List<Product>*, *prodSequence*. The LINQ example created the instance, but you didn’t need to know how it was built. The *for* loop explicitly keeps track of where it is in the source list, *products*, but the act of visiting each item with the LINQ query was implied. The *if* statement that compares the name of the product isn’t too far off from the *where* clause of the LINQ query, but it is an entirely separate statement. With the loop, adding the product to the *prodSequence* collection means that you must index into the *products* array, and call the *Add* method of the *prodSequence* collection with a reference to the product. In contrast, the *select* statement of the LINQ query simply says what it wants as a result of each record that it sees.

This was a very simple example, but you’ll see many more complex examples throughout this book where LINQ makes it incredibly simple to perform tasks that take a lot more code with a traditional imperative approach. Once you begin using LINQ, you’ll begin to see the benefits and how it can make you more productive in your software engineering endeavors.

Before moving on, let’s review the collection that will be used for many of the queries in this chapter.

The Example Collections

Many of the examples in this chapter use a set of collections of *Product* objects and related *ProductCategory* objects as their data source. Instead of repeating this code, I’ll show it here, one time, for your convenience. Examples are based on the following *Product* and *ProductCategory* classes:

```
public class Product
{
    public int ProductID { get; set; }
    public string Name { get; set; }
    public virtual decimal ListPrice { get; set; }
    public DateTime SellStartDate { get; set; }
```

```

    public int ProductCategoryID { get; set; }
}

public class ProductCategory
{
    public int ProductCategoryID { get; set; }
    public string Name { get; set; }
}

```

Notice that both *Product* and *ProductCategory* have a *ProductCategoryID* property. This enables a relationship between the two objects where you can identify the category that a product belongs to. The *GetProducts* method, following, returns the collection of products used in this chapter:

```

public static List<Product> GetProducts()
{
    return
        new List<Product>
        {
            new Product()
            {
                Name = “wadget”,
                ProductID = 1,
                ListPrice = 7.53m,
                ProductCategory = 2,
                SellStartDate =
                    new DateTime(2009, 1, 1)
            },
            new Product
            {
                Name = “widget”,
                ProductID = 2,
                ListPrice = 13.29m,
                ProductCategory = 2,
                SellStartDate =
                    new DateTime(2009, 5, 1)
            },
            new Product
            {
                Name = “wodget”,
                ProductID = 3,
                ListPrice = 9.71m,
                ProductCategory = 4,
                SellStartDate =
                    new DateTime(2009, 9, 30)
            }
        };
}

```

The preceding example uses object and collection initialization syntax to create three *Product* objects that support the queries in this chapter. The collection is a *List* of *Product* objects. Similar to the *GetProducts* method, you’ll also see the following *GetProductCategories* method that returns a *List* of *ProductCategory* objects being used for examples in this chapter:

```

public static List<ProductCategory> GetProductCategories()
{
    return
        new List<ProductCategory>
        {

```

```

new ProductCategory
{
    ProductCategoryID = 1,
    Name = "Bikes"
},
new ProductCategory
{
    ProductCategoryID = 2,
    Name = "Components"
},
new ProductCategory
{
    ProductCategoryID = 4,
    Name = "Accessories"
}
};
}

```

As you can see in the preceding example, the *ProductCategoryID* in each *ProductCategory* has a value that is used by one or more *Product* objects returned by the *GetProducts* method. This relationship facilitates grouping and joining, which you'll learn how to use later in this chapter.

Now that you have an idea of what LINQ can do for you, it's time to show you. The next section begins this journey by showing you different ways to perform projections.

Implementing Projections

The *select* clause of a LINQ query creates a *projection*, which is the data of each object in the results of a query. You can use *select* on the entire object, a single field or property of an object, or on an anonymous type, each resulting in unique projections of the object type being queried. The following sections describe these projection types and their results.

Collection Object Projections

If you need to use the entire object from a collection, you can do a projection that returns each object. Here's a basic projection on a collection that returns the same object being queried:

```

var products2 =
    from product in GetProducts()
    select product;

ObjectDumper.Write(products2  );

```

In the preceding *from* clause, *product* is a *range* variable, holding each element of the collection that is specified after the *in* clause. The *range* variable is local to the query, so you can reuse the same *range* variable in subsequent queries in the same scope. Also, I could have assigned the results of *GetProducts* to a *local* variable and used the *local* variable with the *in* clause. The only requirement for the *in* clause is that it receive an *IEnumerable<T>*, which *List<T>*, as returned by *GetProducts*, derives from. The projection from the *select* clause is the *range* variable object, which is type *Product*. The result is a collection of *Product* objects, which I'll discuss in the next section. The *Write* method of the *ObjectDumper* dumper class produces a list of each object in the collection. Here are the results from the previous example:

```

ProductID=1  Name=wadget  ListPrice=7.53  SellStartDate=1/1/2009

```

ProductID=2 Name=widget ListPrice=13.29 SellStartDate=5/1/2009
ProductID=3 Name=wodget ListPrice=9.71 SellStartDate=9/30/2009

NOTE *ObjectDumper* is a free utility that ships as part of the Microsoft C# Samples for Visual Studio 2008 package. You can download it from the MSDN Code Gallery at <http://code.msdn.microsoft.com/csharpsamples>. The C# Samples for Visual Studio 2008 package contains hundreds of LINQ examples.

Next, let's discuss the results of the preceding LINQ query.

LINQ to Objects Queries Return `IEnumerable<T>`

The result of a LINQ to Objects query is `IEnumerable<T>`. You couldn't see that directly from the previous section because the results were assigned to an implicitly typed *local* variable. However, that variable could have been declared as `IEnumerable<Product>`, like this:

```
IEnumerable<Product> products3 =  
    from product in GetProducts()  
    select product;
```

The results of the preceding query are exactly the same as the example in the previous section, where *products2* was declared as an implicitly typed *local* variable.

The `IEnumerable<T>` interface is significant because extension methods that implement LINQ to Objects use `IEnumerable<T>` as their target instance. If you recall, extension methods do identify their target instance as their first parameter with a *this* modifier. Here's the equivalent of the preceding query, using the *Select* extension method:

```
IEnumerable<Product> products4 =  
    GetProducts().Select(product => product);
```

The preceding example, with the *Select* extension method, is equivalent to the previous LINQ query that used query syntax. You can see the *lambda* method as the *Select* parameter, selecting the same object that it reads from the `List<Product>` returned by the *GetProducts* method. In fact, the C# compiler translates query syntax into extension methods, such as the preceding *Select*. As you read through this book, you'll develop a clearer picture of how LINQ works, behind the scenes, with a comprehensive look at building a custom LINQ provider in [Chapter 10](#).

Continuing our examination of projections, the next section shows you how to select a single field or property from an object.

Selecting a Single Field or Property

When performing queries, you might only want a single piece of data from each object. This data can be either a field or property. In practice, I rarely ever expose a public field and prefer properties for encapsulation. So all of the examples you see in this book will be based on properties. Here's an example that gets the price of each product:

```
IEnumerable<decimal> products5 =  
    from product in GetProducts()  
    select product.ListPrice;
```

```
ObjectDumper.Write(products5);
```

The preceding projection is for the *ListPrice* property of each product instance. Notice the return value

is now type *IEnumerable<decimal>* to reflect the projection type. Here's the ObjectDumper output from the preceding query:

```
7.53
13.29
9.71
```

Next, you'll see how to perform a projection with an anonymous type.

Projecting into Anonymous Types

You've seen how to perform projections on entire types and on a single property of a type. However, sometimes you might need to perform projections that only use parts of those types. One way to do this is with anonymous types. Here's a projection that uses an anonymous type to access the *ProductID* and *Name* of each *Product*:

```
var products6 =
    from product in GetProducts()
    select
        new
        {
            product.ProductID,
            product.ListPrice
        };
```

```
ObjectDumper.Write(products6);
```

The anonymous type has a comma-separated list of properties, indicating which items will be extracted from each product. Here's the output:

```
ProductID=1    ListPrice=7.53
ProductID=2    ListPrice=13.29
ProductID=3    ListPrice=9.71
```

As you can see, the output indicates that the property names of the anonymous type match the property names from the object being read. You can change these defaults by specifying the property names when declaring the anonymous type. Here's an example that changes the property names in the anonymous type:

```
var products7 =
    from product in GetProducts()
    select
        new
        {
            ID = product.ProductID,
            Price = product.ListPrice
        };
```

```
ObjectDumper.Write(products7);
```

Now, the anonymous type property names are *ID* and *Price*. Notice how the return value of the projection is now an implicitly typed *local* variable. In fact, this is the primary use case for implicitly typed *local* variables. You see, there is no way to declare the return type as *IEnumerable<T>*, because you don't know what *T* is—it's an anonymous type with a type name that you don't know. If you try to be clever and look at the type name in IL, using the techniques I describe in [Chapter 1](#), you still can't declare it in C#. Here's the output from the preceding example, showing that the property names of the anonymous

type are *ID* and *Price*:

```
ID=1    Price=7.53
ID=2    Price=13.29
ID=3    Price=9.71
```

The fact that anonymous type property names can be specified is an important point. In more sophisticated LINQ queries where you need to test equality between objects, the anonymous type property names of objects must match, but of course we know that the property names of separate object types aren't always the same. You'll see examples of this later in the book, but I wanted to point it out as an essential tip for later.

One of the problems with anonymous type projections is that you can't return collections based on anonymous types from a method in a reliable manner. How can callers access anonymous type members when they don't know the type? It's possible to use reflection, as some APIs such as ASP.NET MVC do, but for average software development this is more work than necessary. For an example of how to use reflection to read an anonymous type, you can read the *ObjectDumper* code, which I described earlier. Instead of using anonymous types, you can project into a custom type, as discussed in the next section.

Projecting into Custom Types

When you need a set of data that differs from the object collection being queried and you need to pass that data to a calling method, you can build custom types. You can then project into the public fields or properties of the custom type. The data you need might be a subset or superset of data being queried. In a later section of this chapter, I'll show you different ways to perform *group* and *join* operations that yield a data set that you want to project into a custom type (or anonymous type). In this section, I'll show you the subset use case.

A common feature of applications is to display information in various types of list controls, such as a *ListBox* or *DropDownList*. The requirements for binding data to these controls are that you have value and text, where the text is the name of the item, and the value holds the object index that you can use in a subsequent query when the user selects an item. Along these lines, a collection of *Product* class instances could be displayed in a list where the user wanted to create a master/detail relationship. When they select the product, the program can grab the product key and then do a query on product features. Here's the custom type for holding the required information for a *Product* list:

```
public class ProductInfo
{
    public int ProductID { get; set; }
    public string ProductName { get; set; }
}
```

In the *ProductInfo* class, the *ProductID* becomes the control value, and *ProductName* is the text for the control. If you like coding conventions, here's one that I use: adding an *Info* suffix to types that I use for custom projections. In this case, *ProductInfo* is a custom projection type for the *Product* class. Here's the query that performs the projection into *ProductInfo*:

```
var products8 =
    from product in GetProducts()
    select
        new ProductInfo
        {
```

```

        ProductID = product.ProductID,
        ProductName = product.Name
    };

```

```
ObjectDumper.Write(products8);
```

Here, you can see that *select* uses “new ProductInfo” for the projection, resulting in a collection of *ProductInfo*. Here’s the output:

```

ProductID=1   ProductName=wadget
ProductID=2   ProductName=widget
ProductID=3   ProductName=wodget

```

You can see that the output shows the property names from *ProductInfo*. You can now bind the results to controls in your user interface that work with lists.

Another feature of using custom types for projections is that you can modify their values after the custom type is instantiated. However, you don’t have the ability to change the value of an anonymous type, which is immutable.

In addition to specifying the members of the object to return, you need the ability to limit which objects are returned. The next section shows how to do this, by filtering results with the *where* clause.

Filtering Data

In the previous section on projections, you learned how to pull data from a collection, but the query returned every record in the collection. With limited sets of data, such as a list of product types, this isn’t a problem. However, many of the operations you need to accomplish only require a subset of the records. This section will show you how to filter results by using the *where* clause. Here’s a query that returns *Products* with a *ListPrice* greater than 8.00:

```

var products =
    from product in GetProducts()
    where product.ListPrice > 8.00m
    select product;

```

```
ObjectDumper.Write(products);
```

The preceding *where* clause contains the condition to use in filtering each record in the collection. If the *where* clause expression is *true*, the object is included. Here’s the output:

```

ProductID=2  Name=widget  ListPrice=13.29  SellStartDate=5/1/2009
ProductID=3  Name=wodget  ListPrice=9.71   SellStartDate=9/30/2009

```

As I said, the *where* clause takes an expression that evaluates to *true*. You can build sophisticated conditions using && (and) and || (or) logical operators. Here’s an example of using the && operator:

```

var products2 =
    from product in GetProducts()
    where product.ListPrice > 8.00m &&
           product.Name.StartsWith(“wi”)
    select product;

```

```
ObjectDumper.Write(products2);
```

In the preceding example, the query calls a C# string method, *StartsWith*, to produce a Boolean

condition. Here's the output:

```
ProductID=2  Name=widget  ListPrice=13.29  SellStartDate=5/1/2009
```

With LINQ to Objects, the *where* clause can accept a method call in your own code. Here's a method that gets recent products, within the last 10 days:

```
private static bool Recent(DateTime dateTime)
{
    TimeSpan dateDiff = DateTime.Now - dateTime;
    return Math.Abs(dateDiff.Days) <= 10;
}
```

And here's a query that uses it:

```
var products3 =
    from product in GetProducts()
    where Recent(product.SellStartDate)
    select product;
```

You might need to adjust the dates in the *List<Product>* in *GetProducts* to get the results of the call to *Recent* to filter properly. Although LINQ to Objects allows you to call methods like the preceding example in the *where* clause, the situation changes with other LINQ providers; that is, LINQ to SQL requires that the *where* clause be convertible to SQL, which limits what you can use.

A common query is to search for a specific object, where you already know what its ID is. Here's a query that takes a *ProductID* and returns that specific object:

```
var products4 =
    from product in GetProducts()
    where product.ProductID == 2
    select product;
```

This example used a single ID, but you can also use the techniques discussed earlier to combine conditions for multiple keys.

Sometimes a *where* clause can become long and difficult to read. You also have situations where the same value must be specified in multiple clauses. The next section shows how you can simplify these situations with the *let* clause.

Calculating Intermediate Values

With the *let* clause, you can define an intermediate *range* variable for each object that is processed. Then you can use that variable in the rest of the query, for the object it is defined for. A couple of immediately useful applications of the *let* clause are to simplify *where* clauses and to reduce redundancy. The following example shows how a *where* clause can become difficult to read and redundant when calculating whether a product sale start date is between five and ten days old:

```
var products =
    from product in GetProducts()
    where (DateTime.Now -
        product.SellStartDate)
        .TotalDays > 5 &&
        (DateTime.Now -
        product.SellStartDate)
        .TotalDays <= 10
```

```
select product;
```

In the preceding example, the same calculation is performed for each condition of the *where* clause expression. With a more sophisticated set of conditions, such a query can become increasingly complex. The *let* clause helps manage complexity by calculating that value one time. Here’s a rewrite of the preceding example that shows how it can be simplified with the *let* clause:

```
var products2 =  
    from product in GetProducts()  
    let dayCount =  
        (DateTime.Now - product.SellStartDate)  
        .TotalDays  
    where dayCount > 5 && dayCount <= 10  
    select product;
```

In this example, the calculation for the number of days is performed one time only, in the *let* clause. The benefit of this is that the *where* clause is much more readable because it uses the value of *dayCount* and avoids the redundancy of the “total number of days since starting sales” calculation.

The results so far have appeared in the order that they appear in the data source. However, you’ll often need to order query results in different ways, which the next section covers.

Sorting Query Results

You’ll often need to order results from a query in different ways. Typically, you’ll select one or more columns and the direction of the sort. This section will describe how to perform sorting operations in LINQ with the *orderby* clause. You’ll see sorting by a single property, multiple properties, and how to change sort direction.

Sorting a Single Property

You can sort by a single property, using the *orderby* clause. Here’s an example that sorts products by *ListPrice*:

```
var products =  
    from product in GetProducts()  
    orderby product.ListPrice  
    select product;
```

```
ObjectDumper.Write(products);
```

As previously shown, you add an *orderby* clause, just before the *select* statement to sort the results. The property being sorted, *ListPrice*, follows the *orderby* keyword. If you recall from the definition of the *GetProducts* method earlier in this chapter, the *ListPrice* of the *Products* in the *List* that is returned is not in order. Here’s the output from the example:

```
ProductID=1  Name=wadget          ListPrice=7.53  
             SellStartDate=1/1/2009 ProductCategoryID=2  
ProductID=3  Name=wodget          ListPrice=9.71  
             SellStartDate=9/30/2009 ProductCategoryID=4  
ProductID=2  Name=widget          ListPrice=13.29  
             SellStartDate=5/1/2009 ProductCategoryID=2
```

You can see that the *Product* objects are sorted by *ListPrice*; I’ve formatted the output to fit the book page. Now let’s see how to sort by multiple properties.

Sorting Multiple Properties

In addition to sorting by a single property, you'll need to sort by multiple properties. The solution is to add the new sort properties in a comma-separated list. The properties will be sorted from left to right. Here's an example that sorts products by *ProductCategoryID* and then by *Name*:

```
var products2 =
    from product in GetProducts()
    orderby product.ProductCategoryID, product.Name
    select product;

ObjectDumper.Write(products2);
```

The preceding example shows the comma-separated list of *product.ProductCategoryID* and *product.Name* in the *orderby* clause. The results are that the *Product* objects are sorted by *ProductCategoryID*, and within the *ProductCategoryID* sort, they are sorted by *Name*, shown following:

```
ProductID=1 Name=wadget ListPrice=7.53 SellStartDate=1/1/2009
ProductCategoryID=2
ProductID=2 Name=widget ListPrice=13.29 SellStartDate=5/1/2009
ProductCategoryID=2
ProductID=3 Name=wodget ListPrice=9.71 SellStartDate=9/30/2009
ProductCategoryID=4
```

Next, you'll see how to change the direction of the sort.

Managing Sort Direction

When sorting, the default ordering is *ascending*, which is why all of the query results have appeared in lowest to highest order. However, you can reverse the ordering by specifying *descending* in the *orderby* clause. Additionally, you can explicitly specify that a sort occurs in *ascending* order, though it isn't required. Here's an example that specifies the sort direction of each element of an *orderby* clause:

```
var products3 =
    from product in GetProducts()
    orderby
        product.ProductCategoryID descending,
        product.Name ascending
    select product;
```

In the preceding example, *ProductCategoryID* has a *descending* keyword, indicating that the sort direction is from highest to lowest. The *ascending* keyword for the *Name* property explicitly sorts in ascending order, but it isn't required. Here are the results:

```
ProductID=3 Name=wodget      ListPrice=9.71
    SellStartDate=9/30/2009 ProductCategoryID=4
ProductID=1 Name=wadget      ListPrice=7.53
    SellStartDate=1/1/2009 ProductCategoryID=2
ProductID=2 Name=widget      ListPrice=13.29
    SellStartDate=5/1/2009 ProductCategoryID=2
```

Here, you can see that *ProductID* number 3 is in *ProductCategoryID* of 4. Being the highest numbered category, it is first. Of the *Product* objects with *ProductCategory* of 2, the *Name* properties are sorted.

So far, all of the data has been flat, where all properties are included in the same object. The next section shows you how to get hierarchical data with grouping.

Grouping Sets

Grouping is the ability to organize data into categories. With LINQ, you can do grouping and organize objects into a hierarchy. In the following sections, I'll show you how to group by a single property or multiple properties.

Grouping by a Single Property

This first section will show you how to group by a single key. This will let you learn about the *group by* clause with the minimal amount of syntax. The *Product* object that we've been using throughout this chapter has a *ProductCategoryID* that will be used to show you how grouping works. Here's an example of grouping by using the *ProductCategoryID* from a list of *Product* objects:

```
var products =  
    from product in GetProducts()  
    group product by product.ProductCategoryID  
    into productCategories  
    select  
        new  
        {  
            CategoryID = productCategories.Key,  
            Products = productCategories  
        };  
  
ObjectDumper.Write(products, 2);
```

Several parts of the preceding example make grouping work: the *group by* clause, key selector, query continuation, and projection. The syntax of the *group by* clause is *group object by property*. In this example, object is *product*. Later in this chapter, you'll see how to build more complex queries where the choice of object to group on can vary. The property (key selector), *product.ProductCategoryID*, in the *group by* clause will establish a key for the grouping. The *into* clause creates a query continuation, a *range* variable called *productCategories* that is used in subsequent clauses of the same query. When specifying the key selector property, all objects that match that key will be grouped into the query continuation, *productCategories*, which is a collection of objects.

The *select* clause is the subsequent clause in the preceding example that uses the query continuation, *productCategories*. As you know, you could project into a custom type, but I've used an anonymous type to streamline the example. The important part of the *select* clause in this example is the fact that we grab the *Key* property from the query continuation, *productCategories*, and assign it to the *CategoryID* property of the projection type. The continuation itself is assigned to the *Products* property. This creates an object with a hierarchical relationship where the object contains both the grouping property (*Key*) and a collection property for all of the *Product* objects belonging to that *group*. Here's the output:

```
CategoryID=2 Products=...  
  Products: ProductID=1      Name=wadget  
            ListPrice=7.53   SellStartDate=1/1/2009  
            ProductCategoryID=2  
  Products: ProductID=2      Name=widget  
            ListPrice=13.29   SellStartDate=5/1/2009  
            ProductCategoryID=2  
CategoryID=4 Products=...  
  Products: ProductID=3      Name=wodget  
            ListPrice=9.71    SellStartDate=9/30/2009
```

The output reflects the hierarchy of the objects returned by the grouping query in the preceding example. By default, *ObjectDumper.Write* emits the top level of an object hierarchy, but this time we have two levels. This is why the second parameter to *ObjectDumper.Write* is set to 2, allowing two levels of output.

Next, you'll see how to group by multiple properties.

Grouping by Multiple Properties

In addition to grouping by a single property, you can perform groupings on multiple properties. An extra bonus in this example is that you'll see a practical application of anonymous types. The following example groups by both *ProductCategoryID* and *Name*:

```
var products2 =
    from product in GetProducts()
    group product by
        new
        {
            product.ProductCategoryID,
            product.Name
        }
    into productCategories
    select
        new
        {
            CategoryID =
                productCategories.Key.ProductCategoryID,
            Name = productCategories.Key.Name,
            Products = productCategories
        };

ObjectDumper.Write(products2, 2);
```

The syntax of the *group by* clause is very similar to the previous example for a single property, except that it now uses an anonymous type for a composite key, as the key selector. Before, moving on to the *select* clause, I want to point out the role of the anonymous type in this example. In a layered architecture, it's possible to consider anonymous types as code smells, a subject I'll discuss in [Chapter 11](#). However, in the context of composite keys, as used in the *group by* preceding example, anonymous types are essential.

Notice how the *select* clause projection drills into the query continuation, *productCategories*, *Key* selector to extract the value of each part of the composite key. Here's the output:

```
CategoryID=2 Name=wadget   Products=...
  Products: ProductID=1   Name=wadget
           ListPrice=7.53 SellStartDate=1/1/2009
           ProductCategoryID=2
CategoryID=2 Name=widget   Products=...
  Products: ProductID=2   Name=widget
           ListPrice=13.29 SellStartDate=5/1/2009
           ProductCategoryID=2
CategoryID=4 Name=wodget   Products=...
  Products: ProductID=3   Name=wodget
           ListPrice=9.71 SellStartDate=9/30/2009
```

In the output, you can see that the grouping occurs on the composite key of *CategoryID* and *Name*. Previous grouping examples used the *ObjectDumper.Write* method to emit output, but it isn't obvious how to access members of a grouping. Therefore, I cover accessing grouped objects next.

Accessing Grouped Objects

The preceding examples created a hierarchy where groups have a collection of products. Based on the query in the last section, the following example shows you how to access the members of grouped objects:

```
foreach (var group in products2)
{
    Console.WriteLine(
        "\nCategory ID: {0}, Name: {1}",
        group.CategoryID,
        group.Name);

    foreach (var product in group.Products)
    {
        Console.WriteLine(
            "\tList Price: {0}, Sell Start: {1}",
            product.ListPrice,
            product.SellStartDate.ToShortDateString());
    }
}
```

The outer *foreach* loop prints the values of properties belonging to the composite key. Then it prints all product records for the current group being processed. The types returned by grouping are *IEnumerable<T>*, where *T* is the type of the group. In this case, *T* is the anonymous type defined by the composite key. The continuation clause, which was assigned to *Products*, is type *IGrouping<T, U>*. You already know that *T* is the composite key, but *U* is the anonymous type projected into. In summary, the outer *foreach* looped on *IEnumerable<T>*, and the nested *foreach* looped on a collection of *IGrouping<T, U>*. Here's the output:

```
Category ID: 2, Name: wadget
    List Price: 7.53, Sell Start: 1/1/2009

Category ID: 2, Name: widget
    List Price: 13.29, Sell Start: 5/1/2009

Category ID: 4, Name: wodget
    List Price: 9.71, Sell Start: 9/30/2009
```

The preceding examples show the relationship between the composite grouping key and the contents of each *IGrouping*. In this example, each group only has a single member, but you would have multiple members in each group for any *Product* objects with the same *ProductCategoryID* and *Name*.

Previous examples showed how to extract information from a single object. However, in practice you'll need to combine the results of multiple objects. The next section shows you how to combine object results with joins.

Joining Objects

When working with LINQ to Objects, you'll often find that the object you're querying contains the information you need, but on occasion you will need to combine the values of multiple objects. The capability to join objects is especially important in LINQ to SQL or other scenarios where you have normalized data. The following sections show you how to perform joins with objects.

Joining Objects

In the beginning of this chapter, I showed you a class called *ProductCategory*, along with the *Product* class. The relationship between these objects was defined by a *ProductCategoryID*. In practice, you would normally have a *ProductCategory* object with a property that is a collection of references to *Product* objects. However, you need to learn how to perform *join* operations, which are essential to working with normalized data, so I created an example that lets you see how the *join* works. The following example joins the *Product* and *ProductCategory* classes:

```
var products =
    from product in GetProducts()
    join category in GetProductCategories()
    on product.ProductCategoryID equals
        category.ProductCategoryID
    select
        new
        {
            Category = category.Name,
            ProductName = product.Name
        };

```

```
ObjectDumper.Write(products);
```

The parts of the preceding example that make it work are the *join* clause, key comparison, and projection. The *join* clause defines a *range* variable for the collection returned by *GetProductCategories*. A LINQ *join* is based on equality between keys, and you don't have the option for other comparison operators. Therefore, the *equals* keyword is required in the comparison. Using the *=* operator results in a compiler error. The order of comparison is important too. You must specify a previous *range* variable on the left side of *equals* and the *join* range selector on the right side of *equals*. Once you've joined objects, you can project on the object members through their respective *range* variables. Here's the output:

```
Category=Components ProductName=wadget
Category=Components ProductName=widget
Category=Accessories ProductName=wodget

```

The output is more meaningful than previous examples where we printed the *ProductCategoryID*. Through the *join* clause, you can use the *Name* property of the *ProductCategory*, rather than the matching *ProductCategoryID*.

This was an example of an inner *join*, where the keys must match exactly for objects to be included in the results. However, you sometimes need to perform left joins, to include objects that don't have matching keys.

Performing Left Joins

With inner joins, all of the keys in both collections must match. All of the objects from the call to

GetProductCategories weren't included in the results for the previous examples. More specifically, the *ProductCategory* with a *ProductCategoryID* of 1 and *Name* of *Bikes* wasn't included. Sometimes you need to include all of the items in a collection, regardless of the fact that there aren't any matches in the other collection—a left *join*. The following example shows you how to perform a left *join*, which will ensure that all *ProductCategory* objects are included in the results:

```
var products2 =
    from category in GetProductCategories()
    join product in GetProducts()
    on category.ProductCategoryID equals
        product.ProductCategoryID
    into allProdCats
    from product in allProdCats.DefaultIfEmpty()
    select
        new
        {
            Category = category.Name,
            ProductName =
                product == null ?
                    "(empty)" :
                    product.Name
        };

ObjectDumper.Write(products2);
```

This time, I made the *from* clause *GetProductCategories* and then joined products; this put the left *join* on the category list to ensure all of them are included. Also, this *join* has a query continuation, *allProdCats*, that I reassign to the *join range* variable, *product*, in a subsequent *from* clause. The *DefaultIfEmpty* operator ensures the left *join* because it will return *null* for the object even if it doesn't match the *join* condition. Without running *DefaultIfEmpty* on the query continuation, the results would have been a normal *join*. *DefaultIfEmpty* is one of many extension methods you can use for queries. The Appendix provides explanations and examples of all of the available extension methods. Notice the *null* check when assigning to *ProductName* in the projection. This lets me print some default text for categories that don't have matching products. Here's the output:

```
Category=Bikes      ProductName=(empty)
Category=Components ProductName=wadget
Category=Components ProductName=widget
Category=Accessories ProductName=wadget
```

As you see, every *ProductCategory* is included, regardless of whether the joined *Product* collection contains matching items. You can also do a *join* that produces a hierarchical relationship between objects, discussed next.

Performing Group Joins

When performing grouping operations earlier, you learned how to build a hierarchy with the grouping key and all of the records included under that group. You can do something similar with joins, which is called a *group join*. Here's how you can do a group *join* with *Product* objects grouped by *ProductCategory* objects:

```
var products3 =
    from category in GetProductCategories()
    join product in GetProducts()
```



```

on category.ProductCategoryID equals
    product.ProductCategoryID
into allProdCats
from product in allProdCats.DefaultIfEmpty()
select
    new
    {
        Category = category.Name,
        Products = product
    };

ObjectDumper.Write(products3, 2);

```

The preceding example is the same as the previous section, except for the projection. Notice how this example assigns the query continuation to a property. This creates a hierarchy where *Category* contains the name of the category and *Products* contains the list of products in that category.

```

Category=Bikes           Products={ }
Category=Components      Products={ }
  Products: ProductID=1   Name=wadget
      ListPrice=7.53 SellStartDate=1/1/2009
      ProductCategoryID=2
Category=Components      Products={ }
  Products: ProductID=2   Name=widget
      ListPrice=13.29 SellStartDate=5/1/2009
      ProductCategoryID=2
Category=Accessories      Products={ }
  Products: ProductID=3   Name=wodget
      ListPrice=9.71 SellStartDate=9/30/2009
      ProductCategoryID=4

```

All of the categories have products, except for the *Bikes* category. This gave us a left *join* with grouping.

Previous joins used a single property, but you can learn how to use multiple properties for the *join* condition in the next section.

Joining with Composite Keys

Especially with relational data LINQ providers such as LINQ to SQL, you'll need to join tables with multiple keys—a *composite key*. The example I'll use has *ProductModel* and *ProductDescription* objects that have a many-to-many relationship. There is another object named *ProductModelProductDescription* that is used to define the relationship between these objects. The following series of code snippets formulates a small application that demonstrates how to join two tables with composite keys. One object will have *ProductModel* information, another *ProductDescription* object will have culture-specific descriptions, and another object will join *ProductModel* and *ProductDescription* objects with a composite key. The final result will be to use joins, based upon composite keys, to get a culture-specific description for a selected product. Here's the definition of *ProductModel* and *ProductDescription*:

```

public class ProductModel
{
    public int ProductModelID { get; set; }
    public string Name { get; set; }
}

```

```
public class ProductDescription
{
    public int ProductDescriptionID { get; set; }
    public string Description { get; set; }
}
```

Both *ProductModel* and *ProductDescription* have ID properties, *ProductModelID* and *ProductDescriptionID*, respectively. The many-to-many relationship between these two objects is managed by the *ProductModelProductDescription* class, shown here:

```
public class ProductModelProductDescription
{
    public int ProductModelID { get; set; }
    public int ProductDescriptionID { get; set; }
    public string Culture { get; set; }
}
```

Notice that *ProductModelProductDescription* has a *Culture* property. The example query for demonstrating a *join* with a composite key will take advantage of the culture to obtain a localized description for a particular product. Before creating the query, you'll need to see the data to understand the contents of each object and how they'll be affected by the query. The *GetProductModels* method following returns a *List* of *ProductModel*:

```
public static List<ProductModel> GetProductModels()
{
    return
        new List<ProductModel>
        {
            new ProductModel
            {
                ProductModelID = 1,
                Name = "Classic Vest"
            },
            new ProductModel
            {
                ProductModelID = 2,
                Name = "Cycling Cap"
            }
        };
}
```

Here is the *GetProductDescriptions* method, which returns a *List* of *ProductDescription* objects:

```
public static List<ProductDescription> GetProductDescriptions()
{
    return
        new List<ProductDescription>
        {
            new ProductDescription
            {
                ProductDescriptionID = 1712,
                Description = "เสื้อกันลม สีน้ำเงิน ขนาดพิเศษสำหรับฤดูร้อน"
            },
            new ProductDescription
            {
                ProductDescriptionID = 1721,
```

```
    },
};
}
```

Given the two collections of *ProductModel* and *ProductDescription* from the previous methods, here's a method that returns a collection of *ProductModelProductDescription* objects that defines the relationship between these objects:

```
public static List<ProductModelProductDescription>
    GetModelsAndDescriptions()
{
    return
        new List<ProductModelProductDescription>
        {
            new ProductModelProductDescription
            {
                ProductModelID = 1,
                ProductDescriptionID = 1712,
                Culture = "th"
            },
            new ProductModelProductDescription
            {
                ProductModelID = 2,
                ProductDescriptionID = 1721,
                Culture = "th"
            },
            new ProductModelProductDescription
            {
                ProductModelID = 3,
                ProductDescriptionID = 0000,
                Culture = "xx"
            }
        };
}
```

As you can see, the *ProductModelProductDescription* class has *ProductModelID* and *ProductDescriptionID* keys, some of which match objects returned by *GetProductModels* and *GetProductDescriptions* collections methods shown previously. The *GetModelsAndDescriptions* collection is also populated with cultures, where the *Culture* property of each *ProductModelProductDescription* instance is initialized with a culture. In the example, I included records for Thai language descriptions, *th*, and some other language that won't appear in the query; Replace *xx* with the language you love to hate.

Now you're ready to see the query that uses these collections. The goal of this query is to return the Thai language description for a cycling cap. We know the language, *Thai*, and the model, *Cycling Cap*. The following code shows how to do this:

```
int modelID = 2;
string culture = "th";

var products4 =
    from model in GetProductModels()
    where model.ProductModelID == modelID
    join modelDesc in GetModelsAndDescriptions()
```

```

on
  new
  {
    ID = modelID,
    Culture = culture
  }
equals
  new
  {
    ID = modelDesc.ProductModelID,
    Culture = modelDesc.Culture
  }
join description in GetProductDescriptions()
on modelDesc.ProductDescriptionID
equals description.ProductDescriptionID
select description.Description;

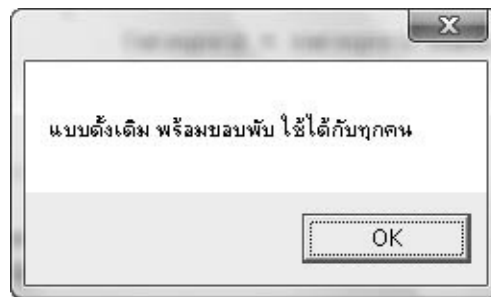
```

```

MessageBox.Show(products4.Single());
ObjectDumper.Write(products4);

```

FIGURE 2-1 Thai language description for Cycling Cap model



The most important part of this query is the first *join* that has two anonymous types defining the composite keys to be compared. Pay particular attention to the fact that both anonymous types have *ID* and *Culture* properties. This makes them the same type, which is a requirement for *join* keys. For example, if I had left the *ID* out and used the variables *modelID* and *modelDesc.ProductModelID*, the example wouldn't have compiled because the first property of each anonymous type would have been *modelID* and *ProductModelID*, respectively, which makes the two anonymous types different types.

A couple parts of the example are essential to making it work: variables and filtering. The *modelID* and *culture* variables are explicitly defined here, but you can imagine that they could have been method input parameters also. The *where* clause restricts the results so that we only work with a single object.

This example also demonstrates how to build more complex queries with multiple joins, a skill necessary for reports and summaries. You might have noticed the second *join* to the *ProductDescription* collection returned by *GetProductDescriptions*. This helps us get a reference to the proper description object, which was the ultimate goal of the query.

Because the results of Thai on an English language console appear as question marks, I added the call to *MessageBox.Show* so you can see the results in [Figure 2-1](#).

Another query type, related to joins, is *select many*, which is discussed next.

Performing Select Many Joins

There are a couple scenarios for *select many* clauses: flattening object hierarchies and performing cross-

joins. The following sections will show you how to handle each of these scenarios with *select many*.

Flattening Object Hierarchies

To understand how a *select many* query flattens a hierarchy, you'll need to see what the hierarchy of objects looks like. The following two objects, *SalesOrderDetail* and *SalesOrderHeader*, are related in that *SalesOrderHeader* contains a collection of *SalesOrderDetail*. Here are their definitions:

```
public class SalesOrderDetail
{
    public int ProductID { get; set; }
    public int OrderQty { get; set; }
    public decimal UnitPrice { get; set; }
}
public class SalesOrderHeader
{
    public DateTime OrderDate { get; set; }
    public List<SalesOrderDetail> Details { get; set; }
}
```

As you can see, *SalesOrderHeader* has a *Details* property whose type is *List* of *SalesOrderDetail*. The following method returns a collection of *SalesOrderHeader* objects that are necessary for demonstrating a *select many* query.

```
public static List<SalesOrderHeader> GetSalesOrders()
{
    return
        new List<SalesOrderHeader>
        {
            new SalesOrderHeader
            {
                OrderDate =
                    new DateTime(2009, 5, 1),
                Details =
                    new List<SalesOrderDetail>
                    {
                        new SalesOrderDetail
                        {
                            ProductID = 1,
                            OrderQty = 1,
                            UnitPrice = 1.00m
                        },
                        new SalesOrderDetail
                        {
                            ProductID = 2,
                            OrderQty = 2,
                            UnitPrice = 2.00m
                        },
                        new SalesOrderDetail
                        {
                            ProductID = 3,
                            OrderQty = 3,
                            UnitPrice = 3.00m
                        }
                    }
            }
        }
};
}
```

Given the collection of *SalesOrderHeader* objects returned from the preceding *GetSalesOrders* method, you can flatten the hierarchy with the following *select many* query:

```
var orders =
    from order in GetSalesOrders()
    from detail in order.Details
    select
        new
        {
            OrderDate = order.OrderDate,
            Product =
                from product in GetProducts()
                where product.ProductID == detail.ProductID
                select product,
            Quantity = detail.OrderQty,
            Price = detail.UnitPrice * detail.OrderQty
        };

ObjectDumper.Write(orders);
```

The operative clauses in the preceding query are the two *from* clauses, which is what makes this a *select many* query. The data source for the second *from* clause comes from the *Details* property of the first. This gives you *range* variables for both the *SalesOrderHeader*, *order*, and *SalesOrderDetail*, *detail*, which are used in the *select* clause. The *select* clause operates on the closest *from*, which is *detail*, allowing projections on every *detail* in each *order*.

You might have noticed that I am progressively adding more complexity to the queries so you can see more examples. In this case, you can see how I use a subquery to extract the *Product*, based on the *detail.ProductID*. Then I calculate the total price by multiplying the *detail.UnitPrice* and *detail.OrderQty*. Here are the results of the preceding *select many* query:

```
OrderDate=5/1/2009 Product=... Quantity=1 Price=1.00
OrderDate=5/1/2009 Product=... Quantity=2 Price=4.00
OrderDate=5/1/2009 Product=... Quantity=3 Price=9.00
OrderDate=1/1/2009 Product=... Quantity=22 Price=528.00
OrderDate=1/1/2009 Product=... Quantity=34 Price=1292.00
```

The other type of *select many* query can give you the effects of a cross-join, which is discussed next.

Performing Cross-Joins

A cross-join produces the Cartesian product of two sets. To see how this could be used, perhaps you need to localize the descriptions of all merchandise in your store. You could grab all of the products and begin writing a description in each language one by one, but that would be laborious and subject to error. Wouldn't it be nice if you could just generate a list? A good tool for the job in LINQ is a *select many* query. To do this, we'll use two methods that already exist, *GetProducts*, which returns a collection of *Product* objects, and *GetModelsAndDescriptions*, which returns a collection of *ProductModelProductDescription* objects that have a *Culture* property. The *GetProduct* method is defined at the beginning of this chapter, and the *GetModelsAndDescriptions* method is defined in the previous section on joins. Here's the query:

```
var productsAndCultures =
    from product in GetProducts()
    from culture in
```

```

        (from model in GetModelsAndDescriptions()
         select model.Culture)
        .Distinct()
select
new
{
    Product = product.Name,
    Culture = culture
};

ObjectDumper.Write(productsAndCultures);

```

As you saw earlier, a common implementation of the *select many* query is to flatten a hierarchy. However, the *from* clauses in the earlier *select many* clause read from data sources that don't have the hierarchical relationship shown in the previous section, resulting in a cross-join. The collection returned by *GetModelsAndDescriptions* contains multiple entries where the culture is Thai, and I needed a single instance of each language to avoid duplication. So, instead of providing *GetModelsAndDescriptions* as the data source, the second clause uses a subquery as its data source. Since a subquery returns *IEnumerable<T>*, it will work fine. The subquery needs to be enclosed in parentheses so we can use the *Distinct* operator, which ensures that only one instance of a culture is used in the cross-join. The following output demonstrates how easy it was to achieve the goal of producing a comprehensive list that includes all products and cultures:

```

Product=wadget Culture=th
Product=wadget Culture=xx
Product=widget Culture=th
Product=widget Culture=xx
Product=wodget Culture=th
Product=wodget Culture=xx

```

This completes our discussion of joins and *select many* clauses. The next section shows you an important skill—how to use LINQ with non-generic collections.

Querying Non-*IEnumerable<T>* Collections

Most of the data sources you access with LINQ to Objects will be *IEnumerable<T>* or *IEnumerable<T>*-derived type. However, there could be times when you encounter a non-generic collection, which is simply *IEnumerable*, but would like to access it with LINQ. This could be confusing and you certainly don't want to rewrite code or copy elements into a generic collection just to do a query. The following example shows you how to query non-*IEnumerable<T>* collections.

```

public static ArrayList GetProductsArrayList()
{
    return
        new ArrayList
        {
            new Product()
            {
                Name = "wadget",
                ProductID = 1,
                ListPrice = 7.53m,
                SellStartDate =
                    new DateTime(2009, 1, 1),
                ProductCategoryID = 2
            },

```

```

new Product
{
    Name = "widget",
    ProductID = 2,
    ListPrice = 13.29m,
    SellStartDate =
        new DateTime(2009, 5, 1),
    ProductCategoryID = 2
},
new Product
{
    Name = "wodget",
    ProductID = 3,
    ListPrice = 9.71m,
    SellStartDate =
        new DateTime(2009, 9, 30),
    ProductCategoryID = 4
}
};
}

```

The *ArrayList* returned by the preceding *GetProductsArrayList* method contains the same objects as the generic *List* of *Product* returned by the *GetProducts* method, defined at the beginning of this chapter. Here's an example of what *not* to do:

```

var products =
    from product in GetProductsArrayList()
    select product;

```

The preceding query results in the following C# compiler error:

```

Could not find an implementation of the query pattern for source type
'System.Collections.ArrayList'. 'Select' not found. Consider explicitly
specifying the type of the range variable 'product'.

```

While an *ArrayList* is *IEnumerable*, it still won't work, because the query can't figure out its type. Here's how to specify the query type, allowing you to query the non-generic *ArrayList* collection:

```

var products =
    from Product product in GetProductsArrayList()
    select product;

```

```

ObjectDumper.Write(products);

```

This example works because the *Product* type is specified for the *range* variable, *product*. The following output demonstrates how this works:

```

ProductID=1 Name=wadget      ListPrice=7.53
    SellStartDate=1/1/2009 ProductCategoryID=2
ProductID=2 Name=widget      ListPrice=13.29
    SellStartDate=5/1/2009 ProductCategoryID=2
ProductID=3 Name=wodget      ListPrice=9.71
    SellStartDate=9/30/2009 ProductCategoryID=4

```

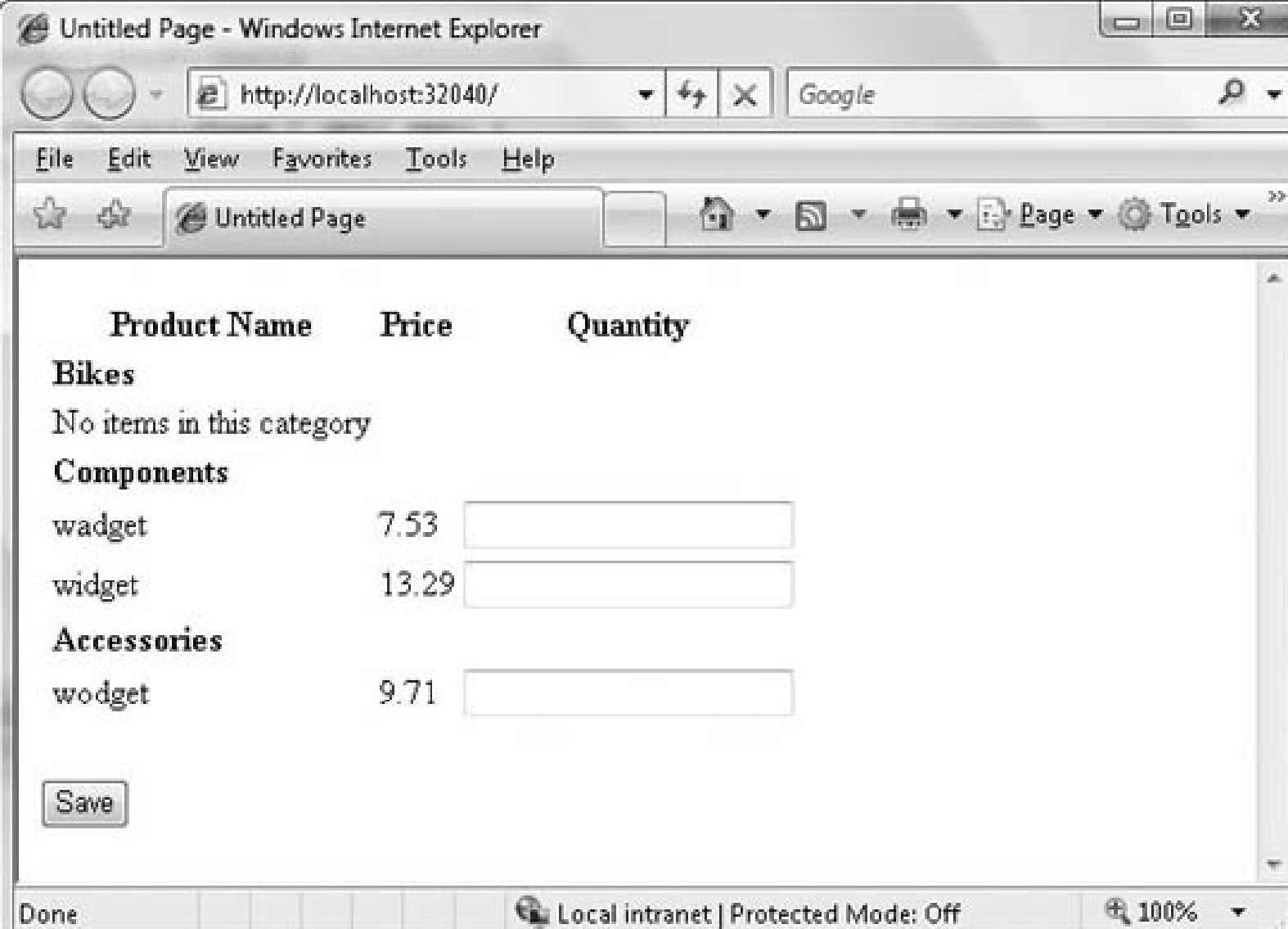
This example demonstrates that by adding the type to the *range* variable, you can query any *IEnumerable* data source. This concludes the examples that demonstrate the features of query syntax. Next, you'll see a more involved example that shows how LINQ to Objects can be used in practice.

You should only define a type on a *range* variable for *IEnumerable* collections. Defining the *range* variable on *IEnumerable<T>* collections causes the C# to generate additional code that is unnecessary. In the first release of LINQ, prior to Service Pack 1 (SP1), there was a bug with conversions where the array type was *floating point*, but the *range* variable was *int*. Instead of truncating the floating point value (a floor), as you would expect, the LINQ query rounded up (a ceiling). This is fixed in the .NET Framework 3.5 SP1.

A Practical Example of LINQ to Objects

LINQ to Objects can be applied anywhere you need to query objects. Collections are readily available throughout the .NET Framework Class Library, such as *Process.GetProcesses* or *ServiceController.GetServices*. LINQ to Objects is particularly useful for extracting data from UI controls.

The example in this section will show you how to use LINQ to Objects to extract data from an editable nested *ListView* control. The application simulates having an order screen, shown in [Figure 2-2](#), where products are grouped into categories.



The screenshot shows a web browser window titled "Untitled Page - Windows Internet Explorer". The address bar shows "http://localhost:32040/" and the search bar contains "Google". The browser's menu bar includes "File", "Edit", "View", "Favorites", "Tools", and "Help". The toolbar shows "Untitled Page" and various navigation and tool icons. The main content area displays an order form with the following structure:

Product Name	Price	Quantity
Bikes		
No items in this category		
Components		
widget	7.53	<input type="text"/>
widget	13.29	<input type="text"/>
Accessories		
widget	9.71	<input type="text"/>

At the bottom left of the form is a "Save" button. The browser's status bar at the bottom shows "Done", "Local intranet | Protected Mode: Off", and "100%".

FIGURE 2-2 Order form read via LINQ to Objects

In [Figure 2-2](#), you can change the quantities of the items and click a Save button to transform the

products into orders. To see how this works, you’ll see the HTML for the nested *ListView*, the definition of an object that handles business logic, and the LINQ to Objects implementation that reads results from the form. [Listing 2-1](#) starts off with the HTML for the nested *ListView* controls on an ASP.NET page.

Listing 2-1 Nested ListView Form

```
<%@ Page Language="C#" AutoEventWireup="true"
    CodeBehind="Default.aspx.cs"
    Inherits="DemoWebApp._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>
<body>
    <form id="form1" runat="server">
        <asp:ListView ID="lvCategories" runat="server"
            DataSourceID="ProductsDataSource">
            <LayoutTemplate>
                <table runat="server">
                    <tr runat="server">
                        <td runat="server">
                            <table id="itemPlaceholderContainer"
                                runat="server" border="0" style="">
                                <tr runat="server" style="">
                                    <th runat="server">
                                        Product Name
                                    </th>
                                    <th runat="server">
                                        Price
                                    </th>
                                    <th runat="server">
                                        Quantity
                                    </th>
                                </tr>
                                <tr id="itemPlaceholder" runat="server">
                                </tr>
                            </table>
                        </td>
                    </tr>
                    <tr runat="server">
                        <td runat="server" style="">
                        </td>
                    </tr>
                </table>
            </LayoutTemplate>
            <ItemTemplate>
                <tr style="text-align: left;">
                    <th>
                        <asp:Label ID="lblCategory" runat="server"
                            Text='<%# Eval("CategoryName") %>' />
                    </th>
                    <td></td>
                    <td></td>
                </tr>
            </ItemTemplate>
        </asp:ListView ID="lvProducts" runat="server">
    </form>
</body>
</html>
```

```

DataSource='<%# Eval("Products") %>'
DataKeyNames="ProductID">
<LayoutTemplate>
    <tr id="itemPlaceholder" runat="server" />
</LayoutTemplate>
<EmptyDataTemplate>
    <tr style="">
        <td>
            No items in this category
        </td>
        <td></td>
        <td></td>
    </tr>
</EmptyDataTemplate>
<ItemTemplate>
    <tr style="">
        <td>
            <asp:Label ID="lblProductName"
                runat="server"
                Text='<%# Eval("Name") %>' />
        </td>
        <td>
            <asp:Label ID="lblListPrice"
                runat="server"
                Text='<%# Eval("ListPrice") %>' />
        </td>
        <td>
            <asp:TextBox ID="txtQuantity"
                runat="server"></asp:TextBox>
        </td>
    </tr>
</ItemTemplate>
</asp:ListView>
</ItemTemplate>
</asp:ListView>
<asp:ObjectDataSource ID="ProductsDataSource" runat="server"
    OldValuesParameterFormatString="original_{0}"
    SelectMethod="GetCategoriesAndProducts"
    TypeName="DemoWebApp.ProductManager">
</asp:ObjectDataSource>
<br />
<asp:Button ID="btnSave" runat="server"
    onclick="btnSave_Click" Text="Save" />
</form>
</body>
</html>

```

[Listing 2-1](#) shows two *ListView* controls, where *lvProducts* is nested inside of *lvCategories*. The *DataSource* for *lvProducts* binds to the *Products* property of the *lvCategories* data source, which is defined in the *ProductsDataSource ObjectDataSource*. The *ProductsDataSource* reads its data from the *GetCategoriesAndProducts* method of the *ProductManager* class, shown in [Listing 2-2](#).

Listing 2-2 The Product Manager Class

```

using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Linq;

```

```

namespace DemoWebApp
{
    [DataContract]
    public class ProductManager
    {
        [DataObjectMethod(DataObjectMethodType.Select)]
        public List<CategoryAndProducts> GetCategoriesAndProducts()
        {
            var catsAndProds =
                from cats in GetProductCategories()
                join prods in GetProducts()
                on cats.ProductCategoryID equals
                    prods.ProductCategoryID
                into prods
                select
                    new CategoryAndProducts
                    {
                        CategoryName = cats.Name,
                        Products = prods.ToList()
                    };

            return catsAndProds.ToList();
        }

        public static List<Product> GetProducts()
        {
            // return products
        }

        public static List<ProductCategory> GetProductCategories()
        {
            // return product categories
        }
        internal void SaveOrder(SalesOrderHeader order)
        {
            // save the order
        }
    }
}

```

The *GetCategoriesAndProducts* method from [Listing 2-2](#) shows another example of performing a group *join*, which is covered earlier in this chapter. The query obtains its data from calls to *GetProducts* and *GetProductCategories*, which are the same methods defined in the beginning of this chapter. It also demonstrates a projection on a custom type, *CategoryAndProducts*, shown in [Listing 2-3](#).

Listing 2-3 The Category AndProducts Custom Type Used for Projection

```

using System.Collections.Generic;

namespace DemoWebApp
{
    public class CategoryAndProducts
    {
        public string CategoryName { get; set; }
        public List<Product> Products { get; set; }
    }
}

```

As stated earlier in this chapter, when discussing projections, the *GetCategoriesAndProducts* method in [Listing 2-2](#) could be called by other code, meaning that the object types returned must be strongly typed. The relationship between the properties in *CategoryAndProduct* is hierarchical, where *CategoryName* holds the name of the category and *Products* is a *List* of *Product*. The *Product* property is the same property that the nested *ListView*, *lvProducts*, in [Listing 2-1](#) binds to with its *DataSource* property.

Finally, you get to see the query that reads from the *ListView* controls, extracting data to create an order. [Listing 2-4](#) contains the event handler from the Save button that reads data and demonstrates how to read *ListView* data with LINQ to SQL.

Listing 2-4 Reading a ListView Control with LINQ to SQL

```
using System;
using System.Linq;
using System.Web.UI.WebControls;
namespace DemoWebApp
{
    public partial class _Default : System.Web.UI.Page
    {
        protected void btnSave_Click(object sender, EventArgs e)
        {
            var orderDetails =
                from cat in lvCategories.Items
                let lvProducts =
                    cat.FindControl("lvProducts") as ListView
                from prod in lvProducts.Items
                let txtQuantity =
                    prod.FindControl("txtQuantity") as TextBox
                let lblPrice =
                    prod.FindControl("lblListPrice") as Label
                select
                    new SalesOrderDetail
                    {
                        ProductID = int.Parse(
                            lvProducts
                                .DataKeys[prod.DataItemIndex]
                                .Value.ToString()),
                        OrderQty = int.Parse(txtQuantity.Text),
                        UnitPrice = decimal.Parse(lblPrice.Text)
                    };
            var order =
                new SalesOrderHeader
                {
                    OrderDate = DateTime.UtcNow,
                    Details = orderDetails.ToList()
                };
            new ProductManager().SaveOrder(order);
        }
    }
}
```

[Listing 2-4](#) is the code-behind file for the ASP.NET page in [Listing 2-1](#). The Save button handler contains a *select many* query that reads data from each of the *lvProducts ListView* instances nested inside of *lvCategory* items. The example uses *let* clauses to access *ListView* items with calls to *FindControl*. The *SalesOrderHeader* instance contains a *List* of *SalesOrderDetail*. Since the result of the query is

IEnumerable, the code executes to *ToList* operator on *orderDetails* to assign the collection as a *List* of *SalesOrderDetail* to the identically typed *Details* property.

The example in this section demonstrated how easy it is to use LINQ to Objects in everyday coding. The example is easy to read because it is a declarative syntax that explains what the code is doing. This relieves you of needing to code imperative instructions for every step that needs to be performed to accomplish the same task. Hopefully, you now see that it is possible to use LINQ to Objects in virtually all parts of your code and that it has potential to simplify your tasks and help you become more productive.

Summary

LINQ to Objects includes an entire syntax for querying data sources. You learned different ways to perform projections and to filter, order, and group code. Joins allow you to combine objects into a single source, and you learned how to perform left joins and group joins. The sections on grouping and joins showed you another way to use anonymous types, to create composite keys. A related topic was *select many* queries, which allow you to flatten object hierarchies or perform cross-*join* operations.

The final example of the chapter demonstrated how you can use LINQ to Objects to query data sources. In this case, the data source was a *ListView* control, but you also learned that you can query any *IEnumerable* or *IEnumerable<T>* data source. One aspect of the example was that it all ran in memory and didn't work with persistent data. For that, you need other LINQ providers for data sources that reside in a persistent medium. You'll learn about one of those providers in the next chapter, which covers LINQ to SQL.

CHAPTER 3

Handling LINQ to SQL with Visual Studio

Many applications work with data that is stored in a database. Therefore, it shouldn't be a surprise that Microsoft has created a LINQ provider for SQL Server: LINQ to SQL. Coding with LINQ to SQL is easier than coding to SQL with earlier technologies.

Before LINQ to SQL, a continuing stream of technologies such as ODBC, OleDb, and ADO.NET were introduced; each with incremental improvements in productivity and features over the other. But an overabundance of options for working with data created additional choices and confusion for developers. LINQ to SQL solves this problem by allowing you to work with SQL Server using the same query syntax that you use for all other LINQ providers.

Throughout this chapter, you'll see how LINQ to SQL builds upon the commonality of LINQ query syntax. You'll also learn about how LINQ to SQL works, having a *DataContext* object that manages its operations. Further, you'll understand how to carry forward your existing investments in stored procedures and functions with LINQ to SQL.

LINQ to SQL Capabilities

LINQ to SQL works exclusively with SQL Server. Because of this, it can take advantage of efficiencies and subtle nuances between versions of SQL Server. Through query syntax, direct execution, and support of stored procedures and functions, you can accomplish all of the tasks that other data access technology has provided in the past, plus more.

As you would expect, you can perform create, read, update, and delete (CRUD) operations with LINQ to SQL. Additionally, you can populate an entire database with LINQ to SQL schema. You can also code strings of custom SQL and execute them directly.

Visual Studio 2008 (VS 2008) includes native support for LINQ to SQL project items. It sports a drag-and-drop design surface where you can build *entities*, which are .NET objects that map to SQL Server objects. The choice of whether to work from database objects or to create your own .NET objects first is up to you; tools exist for you to have it either way.

To get you started, the next section leads you into Visual Studio, guides you through the steps of creating a *DataContext* (the environment that manages LINQ to SQL services), and helps you understand the role of the *DataContext* and what it can do for you.

Introducing the Example Database

The examples in this book are based on the Microsoft AdventureWorksLT database. AdventureWorks ships with SQL Server and is available via the SQL Server installer as a sample database. If you don't have SQL Server available, another option is to download the free SQL Server 2005 Express database and then download the AdventureWorksLT database through pages at MSDN. You can find these pages by visiting <http://msdn.microsoft.com> and doing a search for SQL Server 2005 Express. All of the examples in this book were run with SQL Server 2005, but they should also work with SQL Server 2008.

If you install SQL Server Express, make sure you get the SQL Server Management Studio Express also, which is part of the SQL Server Express Toolkit, to help you work with the database. In particular, if you download the AdventureWorks database, it will be set to Read-Only, meaning that you can't write to it—

yet. Before doing anything in VS 2008, you'll need to connect to the database by right-clicking on Data Connections in Server Explorer, selecting Add Connection, and following the dialog box instructions to connect to your database. If you're using SQL Server Management Studio Express, you'll need to attach the database. To enable read/write access to the database, open SQL Server Management Studio, open Databases in Object Explorer, right-click the AdventureWorksLT_Data database, select Properties, and you'll see the Database Properties window, shown in [Figure 3-1](#).

To turn off Read-Only in the Database Properties window, select the Options page, locate Database Read-Only in the State category of the properties grid, and change the value to False. Now, you'll be able to create a connection in Visual Studio and write to the database with LINQ to SQL.

Before digging into examples, you might want to familiarize yourself with the database schema for AdventureWorksLT, shown in [Figure 3-2](#).

While you're perusing the AdventureWorksLT database, take a look at the functions, stored procedures, and views—all of which are accessible via LINQ to SQL. In fact, you can map these files to objects in your application via a LINQ to SQL DataContext object, which is discussed next.

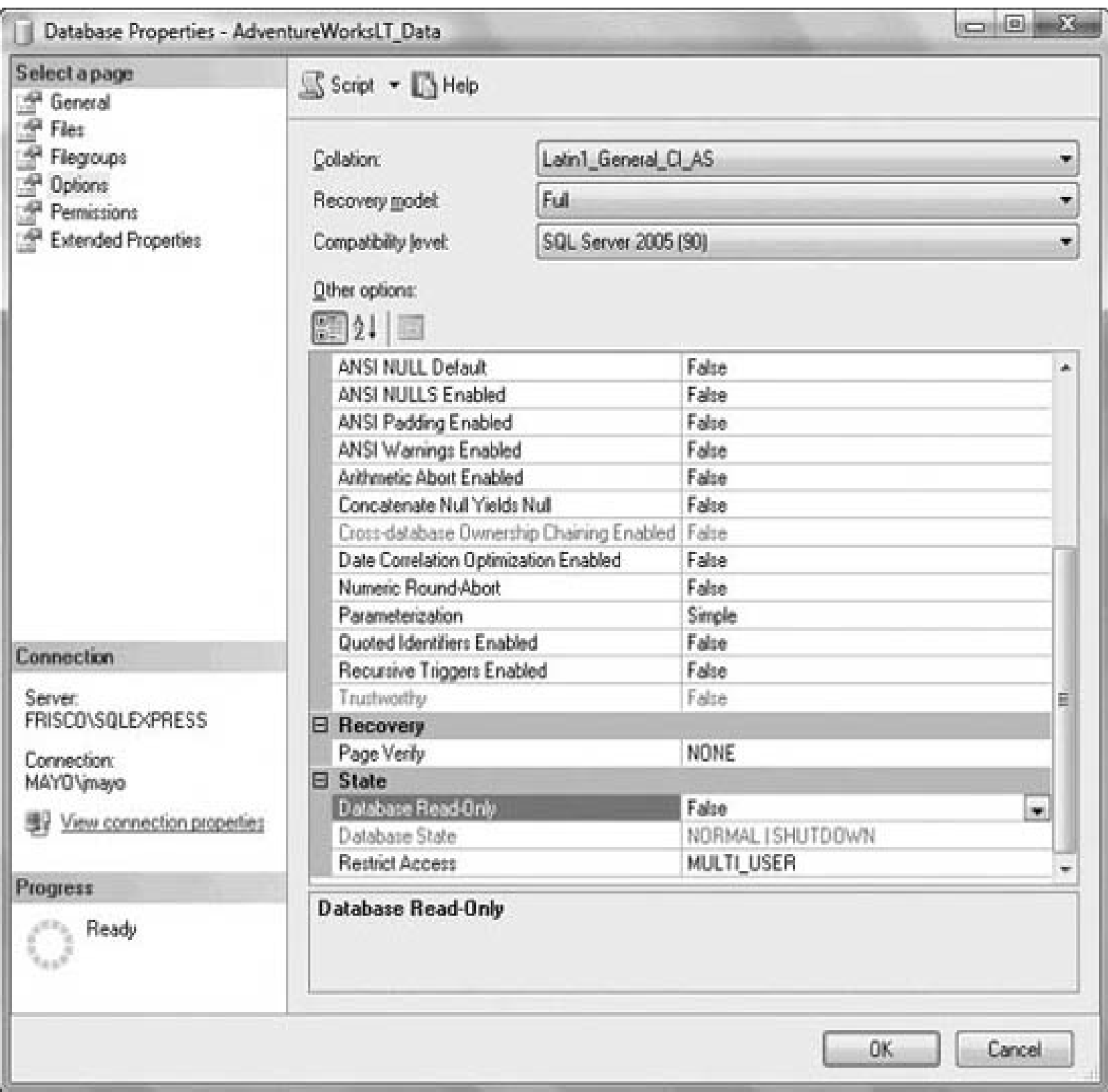


FIGURE 3-1 Turning off Read-Only in the AdventureWorksLT_Data database

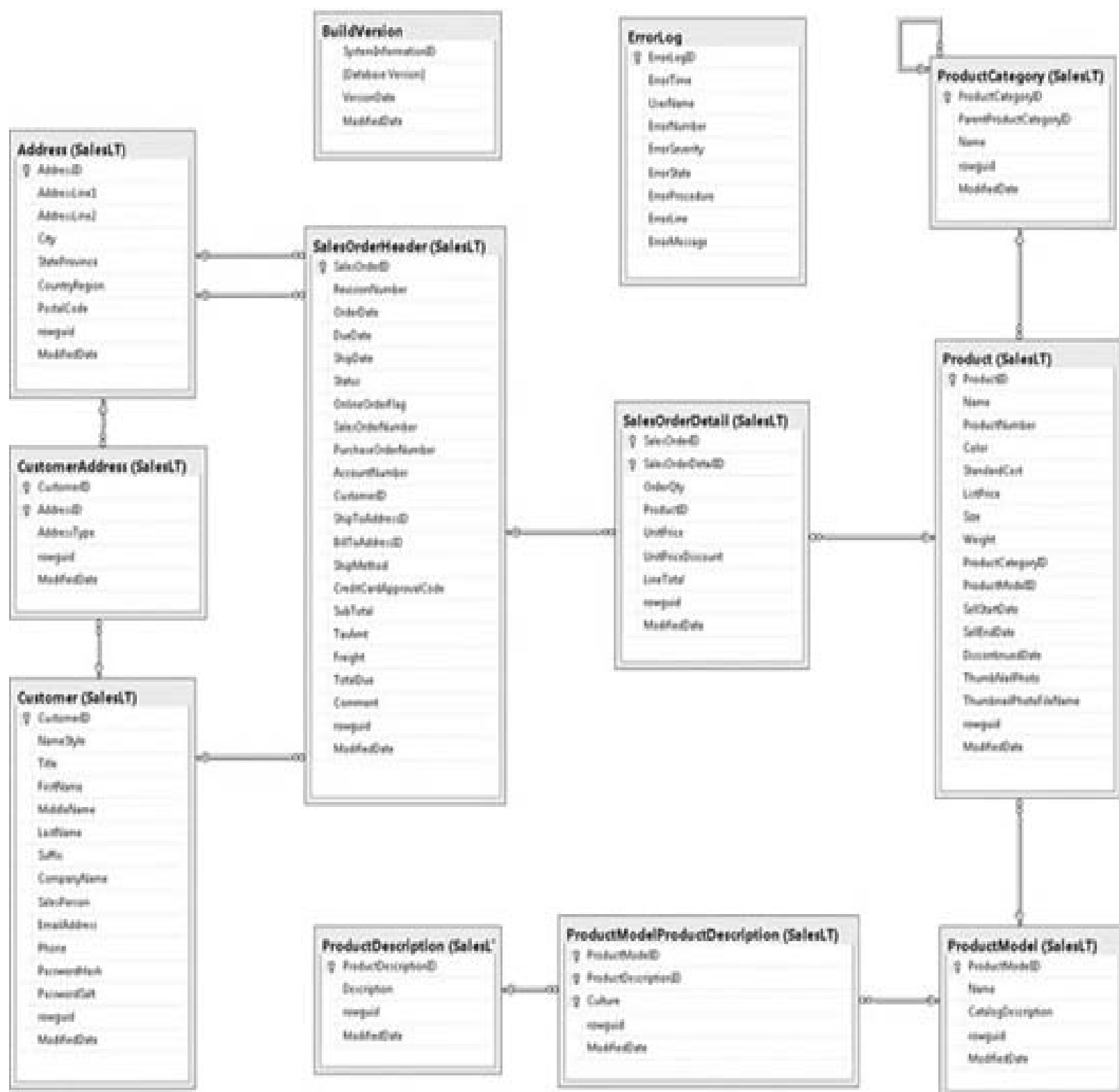


FIGURE 3-2 The AdventureWorksLT database schema

Understanding the DataContext

As its name implies, the DataContext object manages the context for your LINQ to SQL implementation. This context includes a set of services, including object mapping, change tracking, identity management, query execution, and database connection management. Before discussing too many of the details of the DataContext, the next section sets up the rest of the discussion by showing you how to create a DataContext using the LINQ to SQL designer in VS 2008.

Creating a DataContext with the VS 2008 LINQ to SQL Designer

VS 2008 has a visual designer that makes it easy to work with LINQ to SQL. You can create a DataContext with a Project Item wizard and then create entities, which are object representations of database tables. To create a new DataContext, open a new Console Application project, right-click on your project in Solution Explorer, select Add | New Item | LINQ To SQL Classes, name the file **AdventureWorks.dbml**, and click the Add button. [Figure 3-3](#) shows the LINQ to SQL designer created from the preceding action.

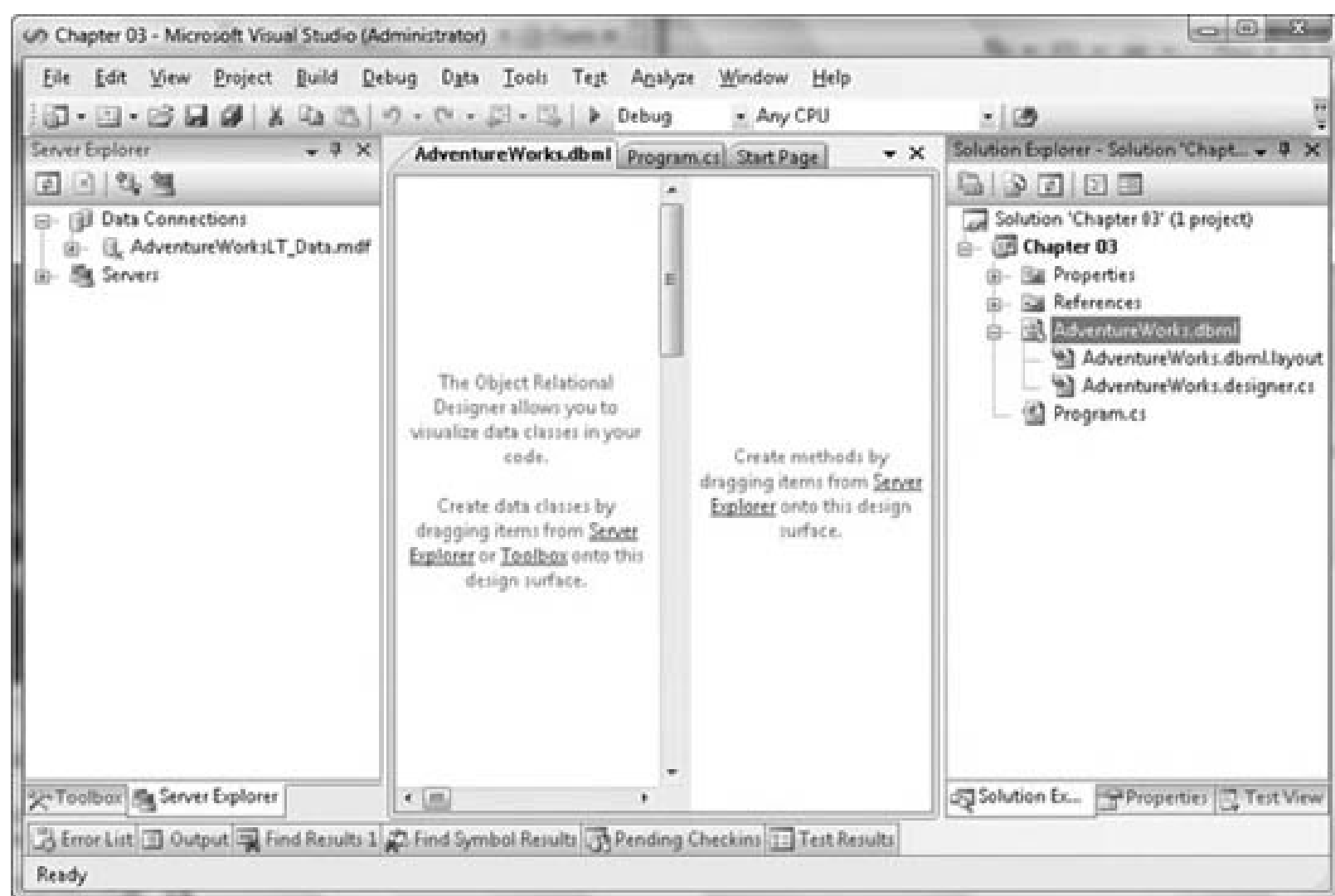


FIGURE 3-3 The LINQ to SQL designer

The LINQ to SQL designer, in the middle of the screen, hosts two blank surfaces. The surface on the left hosts entities, which represent database tables, and the surface on the right hosts functions and stored procedures. In the Solution Explorer, you can see the designer, which is named AdventureWorks.dbml. The AdventureWorks.dbml.layout file holds information for positioning of entities in the designer, and the AdventureWorks.designer.cs file holds auto-generated C# code, reflecting the changes made on the design surface, along with a DataContext derived class. You won't be able to see this code, or changes to the code, until you save any changes made in the LINQ to SQL designer, so make sure you click the Save button. Now, you can open AdventureWorks.designer.cs to see the DataContext, which you can see in [Listing 3-1](#).

Listing 3-1 The LINQ to SQL DataContext

```
#pragma warning disable 1591
```

```
//-----  
// <auto-generated>  
//   This code was generated by a tool.  
//   Runtime Version:2.0.50727.1434  
//  
//   Changes to this file may cause incorrect behavior and will be lost  
//   if the code is regenerated.  
// </auto-generated>  
//-----
```

```
namespace Chapter_03  
{  
    using System.Data.Linq;  
    using System.Data.Linq.Mapping;  
    using System.Data;  
    using System.Collections.Generic;  
    using System.Reflection;  
    using System.Linq;  
    using System.Linq.Expressions;  
    using System.ComponentModel;  
    using System;  
  
    public partial class AdventureWorksDataContext :  
        System.Data.Linq.DataContext  
    {  
  
        private static System.Data.Linq.Mapping.MappingSource  
            mappingSource = new AttributeMappingSource();  
  
        #region Extensibility Method Definitions  
        partial void OnCreated();  
        #endregion  
  
        public AdventureWorksDataContext(string connection) :  
            base(connection, mappingSource)  
        {  
            OnCreated();  
        }  
  
        public AdventureWorksDataContext(System.Data.IDbConnection  
connection) :  
            base(connection, mappingSource)  
        {  
            OnCreated();  
        }  
  
        public AdventureWorksDataContext(  
            string connection,  
            System.Data.Linq.Mapping.MappingSource mappingSource) :  
            base(connection, mappingSource)  
        {  
            OnCreated();  
        }  
  
        public AdventureWorksDataContext(  
            System.Data.IDbConnection connection,  
            System.Data.Linq.Mapping.MappingSource mappingSource) :  
            base(connection, mappingSource)  
        {
```

```

        OnCreated();
    }
}
}
#pragma warning restore 1591

```

I formatted the code a little to make it fit in the book better, but other than that, [Listing 3-1](#) shows the exact code that VS 2008 generated. As the listing comments indicate, this is auto-generated code, and you shouldn't make any changes to it. Instead, you should use the visual designer tools, partial classes, and partial methods to accomplish what you need.

A few important parts of the code to point out include derivation from *DataContext*, partial method extensibility, and database connection management. Notice that the name of the class is *AdventureWorksDataContext*. The name came from the filename you gave the VS 2008 wizard, with *DataContext* appended to the name, indicating that you have control over the *DataContext* name. Another point to remember is that *AdventureWorksDataContext* derives from the *DataContext* class, which you'll learn more about in sections to follow.

The *DataContext* derived type, *AdventureWorksDataContext*, is a partial class, meaning that you can create your own partial class to add functionality. Additionally, notice the partial method, *OnCreated*. The *OnCreated* method is invoked inside of each of the *Adventure WorksDataContext* constructors, giving you the ability to add your own logic to initialize the state of a partial class. [Chapter 1](#) introduces new C# features, including partial methods, if you're not sure about how partial methods work.

The *MappingSource* field, *mappingSource*, allows you to instantiate your *DataContext* with an XML file that defines the mapping between database objects and .NET objects. [Chapter 7](#) describes how to create and use a mapping file. Essentially, the mapping file enables you to define mapping between database objects and .NET objects outside of your code. This chapter shows you how to use VS 2008 instead of a mapping file.

To let the *DataContext* know which mapping file to use, you would pass it via one of the constructors that accepts *mappingSource*. Each of the constructors also accepts database connection instructions so that the *DataContext* will know which database it should communicate with. The next section begins a discussion of *DataContext* services, including database connection management, which begins with the constructors just introduced.

Database Connection Management

The *DataContext* provides database connection management services. When you perform a query, the *DataContext* opens the database connection, runs the query, and closes the database connection some time after receiving query results. You can dynamically set the database connection in code or read it from a configuration file, which you'll learn about in the following sections. You'll also want to know how to explicitly close the connection when you're done with it.

Dynamically Specifying the Database Connection

[Listing 3-1](#) has four constructors, demonstrating two ways to define the database connection: connection string or *IDbConnection*. The connection string is the same as what you would use for ADO.NET. If you wanted to, you could take an existing *SqlConnection* object and pass it to the *DataContext*. Remember to add a *using* declaration for the *System.Data.SqlClient* namespace if you intend to use *SqlConnection*.

The following code shows how to create a *DataContext* using either a connection string or *SqlConnection*:

```
var connStr =
    @"Data Source=.sqlexpress;
    AttachDbFilename=""C:\Program Files\Microsoft
    SQL Server\MSSQL.1\MSSQL\Data\AdventureWorksLT_Data.mdf"";
    Initial Catalog=AdventureWorksLT_Data;
    Integrated Security=True";

// with connection string
var awDataCtxStr = new AdventureWorksDataContext(connStr);

var conn = new SqlConnection(connStr);

// with SqlConnection
var awDataCtxConn = new AdventureWorksDataContext(conn);
```

In the preceding example, *connStr* is a connection string, and it's used to instantiate a new *AdventureWorksDataContext*. Similarly, the code instantiates a new *SqlConnection*, based on *connStr*, and then instantiates another *AdventureWorksDataContext*, only with the *SqlConnection*.

Configuring the Database Connection

You can set the *DataContext* to get its database connection information from a configuration file, which would be *web.config* for web applications or *app.config* (which compiles to *<assembly name>.exe.config*) for executable applications. There are two ways to do this: via the Properties window, or by adding database objects to the LINQ to SQL design surface.

To configure via the Properties window, open the LINQ to SQL designer by double-clicking the *AdventureWorks.dbml* file in Solution Explorer, and open the Properties window. You'll see a property named Connection String that you can use to either type in your own connection string or to select an item in the drop-down list. The Connection String drop-down list populates with databases configured in the Server Explorer. To get a database to show up in this list, open Server Explorer, right-click Data Connections, select Add Connection, and configure the database through the Add Connection window.

Alternatively, you can drag and drop a database object from Server Explorer onto the LINQ to SQL design surface. Visual C# Express will show you a dialog box asking if it's okay to add a data file to the project; click the OK button. Visual Studio won't show you this dialog box. The LINQ to SQL designer will recognize where the object came from and populate the Connection String property in the Property window. [Figure 3-4](#) shows the results of dragging and dropping the Product table from the AdventureWorksLT database onto the design surface.

From [Figure 3-4](#), you can see the Connection String property configured in the Properties window. This only occurs the first time you add a database object to the designer. Thereafter, you can open this Properties window and configure the connection as you need. The Visual Studio Properties window is context sensitive, which means that it will only show properties for items that are selected in the designer. Here, we want to be able to modify the connection string for the entire data context, so we must click on the blank

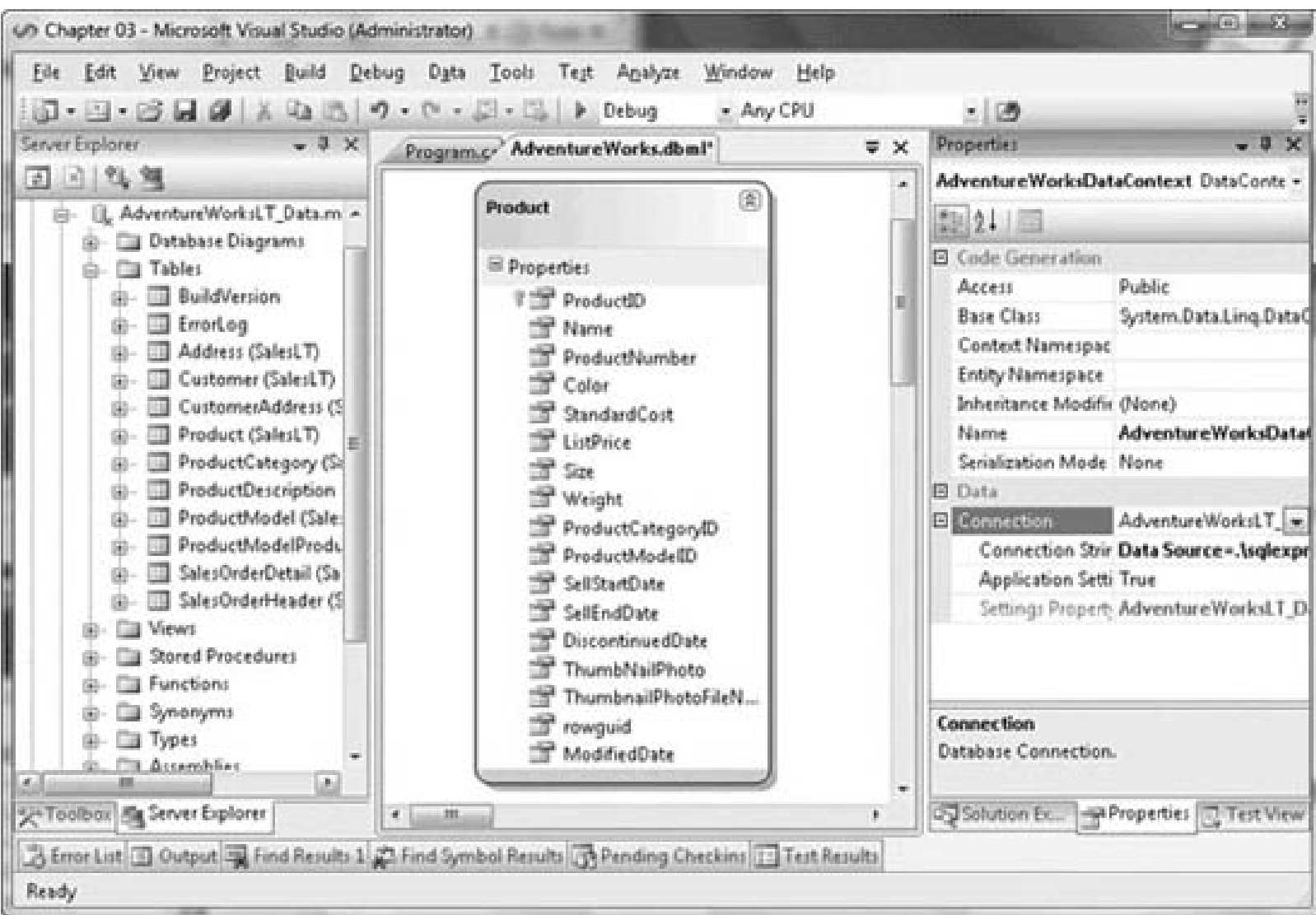


FIGURE 3-4 Adding the first database object to a LINQ to SQL designer configures a connection.

design surface to see the properties for the DataContext. Don't click on any of the entities because that will only show properties for the selected entity.

After the LINQ to SQL designer configures the Connection String, you can see the results in the configuration file and an updated *AdventureWorksDataContext*. The following XML shows the contents of a new app.config file added to the project:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <configSections>
  </configSections>
  <connectionStrings>
    <add name="Chapter_03.Properties.Settings.AdventureWorksLT_DataConnectionString"
      connectionString="Data Source=.\sqlexpress;
      AttachDbFilename=|DataDirectory|\AdventureWorksLT_Data.mdf;
      Initial Catalog=AdventureWorksLT_Data;Integrated Security=True"
      providerName="System.Data.SqlClient" />
  </connectionStrings>
</configuration>
```

This shows that you have another place besides the Properties window to modify the Connection

String. After you save the AdventureWorks.dbml file, you'll also see a new constructor, shown next, in the AdventureWorks.designer.cs file:

```
public AdventureWorksDataContext() : base(
    global::Chapter_03.Properties.Settings.Default
    .AdventureWorksLT_DataConnectionString, mappingSource)
{
    OnCreated();
}
```

As you can see, the new constructor is a default constructor that calls the base class constructor with the VS 2008 Property Settings, which maps to the connection string in app.config that you saw previously.

Closing Database Connections Through the DataContext

As with all prior database technologies, you should be cognizant of the fact that database connections are scarce resources to manage, and that doesn't change with LINQ to SQL. If you open a DataContext, as shown in the previous section, and don't close it, the CLR garbage collector is likely to eventually close it for you. With a single-user desktop application that has its own database, you might be satisfied with that solution, but that is insufficient for multiuser enterprise applications. Especially with web applications where scalability is an issue, you don't want to leave it to chance that the CLR garbage collector will clean up database connections sufficiently. Therefore, you'll want to consider the following form whenever using a DataContext:

```
using (var awDataCtxUsing = new AdventureWorksDataContext())
{
    // queries go here
}
```

Since DataContext is *IDisposable*, it's quick and easy to instantiate in a *using* statement, guaranteeing that the database connection will be closed when you're done with it. This has other minor implications with deferred loading and object tracking, but the benefits in scalability can outweigh any workarounds, depending on your situation. I'll discuss deferred loading in [Chapter 11](#) and object tracking a little later in this chapter. In terms of performance, the DataContext is designed as a lightweight object so that it can be opened and closed as necessary, minimizing performance overhead and offering greater scalability.

You're probably anxious to see how to perform LINQ to SQL queries, and we'll get there. However, you need to have objects to query, which occurs through another service of the DataContext, called *Object Mapping*.

Object Mapping

In the previous section, you learned that dragging and dropping a database object onto the LINQ to SQL design surface resulted in several changes, including database connection setup and a new DataContext constructor. Another change is that code is added to AdventureWorksDataContext.designer.cs for the *Products* table that is added. [Listing 3-2](#) shows the changes to support the new *Products* entity.

Listing 3-2 DataContext Entity Representation

```
// namespaces removed
```

```
[System.Data.Linq.Mapping.DatabaseAttribute(
```



```

    Name="AdventureWorksLT_Data")]
public partial class AdventureWorksDataContext :
    System.Data.Linq.DataContext
{
// mapping removed

#region Extensibility Method Definitions
partial void OnCreated();
partial void InsertProduct(Product instance);
partial void UpdateProduct(Product instance);
partial void DeleteProduct(Product instance);
#endregion

// constructors removed

    public System.Data.Linq.Table<Product> Products
    {
        get
        {
            return this.GetTable<Product>();
        }
    }
}

[Table(Name="SalesLT.Product")]
public partial class Product :
    INotifyPropertyChanging, INotifyPropertyChanged
{
    private static PropertyChangingEventArgs emptyChangingEventArgs
        = new PropertyChangingEventArgs(String.Empty);
    private int _ProductID;
    private string _Name;

// fields removed

#region Extensibility Method Definitions
partial void OnLoaded();
partial void OnValidate(System.Data.Linq.ChangeAction action);
partial void OnCreated();
partial void OnProductIDChanging(int value);
partial void OnProductIDChanged();
partial void OnNameChanging(string value);
partial void OnNameChanged();

// partial methods removed

#endregion

    public Product()
    {
        OnCreated();
    }

    [Column(Storage="_ProductID", AutoSync=AutoSync.OnInsert,
        DbType="Int NOT NULL IDENTITY", IsPrimaryKey=true,
        IsDbGenerated=true)]
    public int ProductID
    {
        get
        {

```

```

        return this._ProductID;
    }
    set
    {
        if ((this._ProductID != value))
        {
            this.OnProductIDChanging(value);
            this.SendPropertyChanging();
            this._ProductID = value;
            this.SendPropertyChanged("ProductID");
            this.OnProductIDChanged();
        }
    }
}

[Column(Storage="_Name", DbType="NVarChar(50) NOT NULL",
    CanBeNull=false)]
public string Name
{
    get
    {
        return this._Name;
    }
    set
    {
        if ((this._Name != value))
        {
            this.OnNameChanging(value);
            this.SendPropertyChanging();
            this._Name = value;
            this.SendPropertyChanged("Name");
            this.OnNameChanged();
        }
    }
}
}

```

// properties removed

```

public event PropertyChangingEventHandler PropertyChanging;

public event PropertyChangedEventHandler PropertyChanged;

protected virtual void SendPropertyChanging()
{
    if ((this.PropertyChanging != null))
    {
        this.PropertyChanging(this, emptyChangingEventArgs);
    }
}

protected virtual void SendPropertyChanged(String propertyName)
{
    if ((this.PropertyChanged != null))
    {
        this.PropertyChanged(this, new PropertyChangedEventArgs(
        Args(propertyName)));
    }
}
}
}

```

There are four important items in [Listing 3-2](#) that I want to highlight: the *Products* entity collection, the *Product* entity, *Product* properties, and attributes. The following sections explain each of these in more detail. Paying attention to pluralization can help here, where *Products*, a collection, holds multiple *Product* instances.

Entity Collections

The *Products* property repeated next, added to *AdventureWorksDataContext* allows access to a collection of *Product* entities.

```
public System.Data.Linq.Table<Product> Products
{
    get
    {
        return this.GetTable<Product>();
    }
}
```

The significance of the *Products* property is that it is the collection that you will query for *Product* entities. It returns a *Table<T>* collection, which derives from *IQueryable<T>*. If you recall, LINQ to Objects queries *IEnumerable<T>* collections, but LINQ to SQL queries *IQueryable<T>* collections. The primary difference is that *IQueryable<T>* collections have additional functionality for allowing you to access external data sources, rather than in-memory objects that reside in an *IEnumerable<T>* derived type. As you can see, the call to *GetTable<Product>* returns an *IQueryable<T>* collection you can query with LINQ. Next, you'll learn about the contents of this collection, the *Product* entity.

Entity Declarations

Entities are often direct representations of database tables, but can also be views or specializations of tables, discussed later in this chapter. The *Product* class, in [Listing 3-2](#), is an entity that represents the *Product* class in the AdventureWorksLT database. Here's the *Product* class, along with a few members, from [Listing 3-2](#):

```
public partial class Product :
    INotifyPropertyChanging, INotifyPropertyChanged
{
    partial void OnLoaded();
    partial void OnValidate(System.Data.Linq.ChangeAction action);
    partial void OnCreated();
    partial void OnProductIDChanging(int value);
    partial void OnProductIDChanged();
    partial void OnNameChanging(string value);
    partial void OnNameChanged();

    public Product()
    {
        OnCreated();
    }
}
```

The *Property* entity, implemented as a class, demonstrates the typical implementation of any LINQ to SQL entity. The *INotifyPropertyChanging* and *INotifyPropertyChanged* interfaces that the *Property* class

implements support object tracking, which I'll discuss in a later section. The partial methods with the *On* prefix open extensibility, where you can define your own partial method to execute custom code when these methods are called. There is an *OnPropertyChanging* and *OnPropertyChanged* for each property in the class, where *Property* in the method identifier is replaced with the name of the property. That is, the *ProductID* property would have *OnProductIDChanging* and *OnProductIDChanged*. A later section on properties will show you how these partial methods are used.

Three of the partial methods, *OnLoaded*, *OnValidate*, and *OnCreated*, support the containing object, rather than just a property. You can write your own partial methods to implement custom code whenever these events occur. Whenever the *DataContext* loads an object, it will call *OnLoaded*. You can implement *OnValidate* to validate the custom state of a partial class you've defined. Notice the call to *OnCreated* in the *Product* constructor, suggesting that you can initialize the state of your custom partial whenever the *Product* class is instantiated.

Next, you can learn about the partial methods that I mentioned here, used as part of property implementations.

Entity Properties

The *Product* class contains properties for each of the fields of the *Product* table. Most of the properties were removed in [Listing 3-2](#), to save space, but they all have similar implementations. For example, the *Name* property, shown next, is a typical entity property:

```
public string Name
{
    get
    {
        return this._Name;
    }
    set
    {
        if ((this._Name != value))
        {
            this.OnNameChanging(value);
            this.SendPropertyChanging();
            this._Name = value;
            this.SendPropertyChanged("Name");
            this.OnNameChanged();
        }
    }
}
```

The *get* accessor just shown simply returns a backing store value, but the *set* accessor has several more statements. The *set* accessor assigns value to the backing store, *_Name*, but is surrounded by a set of partial methods and events.

As is consistent with other aspects of .NET software development, events and methods with *ing* suffixes occur before a change, and events and methods with *ed* suffixes occur after a change. Each entity property has *OnPropertyChanging* and *OnPropertyChanged*, which are partial methods that you can implement. In the case of the preceding *Name* property, replace *Property* in the method identifier with *Name* for *OnNameChanging* and *OnNameChanged*.

In addition to partial methods, an entity *set* accessor has *SendPropertyChanging* and *Send*

PropertyChanged(“*Property*”), where the property string is replaced with the name of the property. In the *Name* property, this would be *SendPropertyChanged*(“*Name*”). You could attach *SendPropertyChanging* and *SendPropertyChanged* events, but their real purpose is to notify the *DataContext* that the property has changed; you’ll learn about object tracking in a later section of this chapter.

LINQ to SQL Attributes

The code in [Listing 3-2](#) contains different attributes that are essential to LINQ to SQL support: *Database*, *Table*, and *Column*. The *Database* attribute decorates the *DataContext*, as shown next:

```
[System.Data.Linq.Mapping.DatabaseAttribute(Name="AdventureWorksLT_Data")]
public partial class AdventureWorksDataContext : System.Data.Linq.DataContext
```

As shown earlier, the *DatabaseAttribute* attribute identifies which database the *DataContext* uses. The *Name* property of the *DatabaseAttribute* matches the database name.

The *Table* attribute matches an entity with its table or view as shown next for the *Product* entity:

```
[Table(Name="SalesLT.Product")]
public partial class Product : INotifyPropertyChanging,
INotifyPropertyChanged
```

The *Name* property holds the fully qualified table name. Just as the *Table* attribute matches a database table with a class, the *Column* attribute matches a field with a property. Here’s an example of a *Column* attribute:

```
[Column(
    Storage="_ProductID",
    AutoSync=AutoSync.OnInsert,
    DbType="Int NOT NULL IDENTITY",
    IsPrimaryKey=true,
    IsDbGenerated=true)]
public int ProductID
```

The preceding *ProductID* property has a *Column* attribute with several properties to specify the field definition. *Storage* means that the *DataContext* can optimize the object identity by setting the backing store directly, rather than going through the property setter and all of its event and partial method handling. *AutoSync.OnInsert* means that this property will be updated automatically anytime a new *Product* instance is added to the database. *DbType* specifies the field type and whether you can assign *null* to it. Notice that *DbType* also includes *IDENTITY* in the string. The *IsPrimaryKey* identifies this property as the primary key when set to *true*, which would be included on any properties that are part of the key for this entity. Setting *IsDbGenerated* to *true* means that this field has an auto-increment column, managed by the database.

That was how the primary key is specified, but normal data columns such as the *Name* property that follows don’t have all of the information for identity specification:

```
[Column(
    Storage="_Name",
    DbType="NVarChar(50) NOT NULL",
    CanBeNull=false)]
public string Name
```

From the previous description, you have an idea of what *Storage* and *DbType* are for, but *DbType* now holds *NVarChar(50)* as its type. While the purpose of the *CanBeNull* property is to support the *DataContext*, you still need to specify *NOT NULL* in the *DbType* column if you want to create a database with the data context.

Creating Databases

As suggested in the previous paragraph, you can use the *Database*, *Table*, and *Column* attributes along with the *DataContext* to create a new database. Instead of using the VS 2008 tools, you could write your own entities, as shown in this section, and then generate a database that matches your objects. Assuming that you created the objects in [Listing 3-2](#) by hand, [Listing 3-3](#) shows how you would go about creating a new database.

Listing 3-3 Creating a Database

```
var ctx = new AdventureWorksDataContext (@“C:\MyAWDB.mdf”);

if (!ctx.DatabaseExists())
{
    ctx.CreateDatabase();

    (from file in System.IO.Directory.GetFiles(@“C:\”)
     select file)
     .ToList()
     .ForEach(
         fileName => Console.WriteLine(fileName));
}

Console.WriteLine(“Pressing any key deletes new db.”);
Console.ReadKey();

ctx.DeleteDatabase();
```

[Listing 3-3](#) shows that you can use the *DataContext*, *AdventureWorksDataContext*, to perform database creation and deletion operations. The string *@“C:\MyAWDB.mdf”* is the connection string, passed to the *AdventureWorksDataContext* constructor, specifying the name and location of the database. This demonstrates the three methods—*DatabaseExists*, *CreateDatabase*, and *DeleteDatabase*—that you can call through a *DataContext*.

During the attributes discussion, you saw how one or more properties can specify the primary key of an entity with the *PrimaryKey* property of the *Column* attribute. The next section builds upon the significance of entity identity through the *DataContext* object tracking service.

Object Tracking

Every entity needs a primary key, which the *DataContext* uses to uniquely identify that object. Anytime you query an object, the *DataContext* will know whether it has that object in memory. If you subsequently perform another query that returns the same object on the same *DataContext* instance, you won’t get a second object in memory. It will be the same. However, the operative word here is on the *same* *DataContext*, because object tracking is per *DataContext* only. It follows that if you instantiate a second *DataContext* and perform the same query, you’ll have a second instance of the same object in memory; one instance for each *DataContext*.

Another feature of the object tracking service is change tracking. The `DataContext` keeps a copy of both the original and modified version of an object. Anytime you change an object that is being tracked by a `DataContext`, the `DataContext` will keep the original object and add the modified object. When submitting changes to the database, the object tracking service examines original and modified values of objects to figure out what needs to be written.

The next section builds upon some of the items in this section, looking at multiple entities and their relationships.

Getting CreateDatabase to Work

When you run [Listing 3-3](#), you could encounter an error message for a login or schema error during the call to `CreateDatabase`: `SecurityException`. The login error contains the text “Cannot open user default database,” and it means that the login doesn’t have a default database in SQL Server. To fix the problem on SQL Server 2005 (or on the Express version), you can log into the database with the following command:

```
sqlcmd -E -S .\sqlexpress -d master
```

In the preceding case, I’m using *sqlexpress*, which can be installed with VS 2008. Once you’re logged on, set the default database like this:

1. Type
`ALTER LOGIN [mayo\jmayo] WITH DEFAULT_DATABASE = master`
2. Press ENTER.
3. Type **go** and press ENTER.

Replace *[mayo\jmayo]* with your login or *[machine\Administrator]*. You can type **exit** to leave the *sqlcmd* prompt. On the schema error, “`SqlException`: The specified schema name ‘SalesLT’ either does not exist or you do not have permission to use it,” you can remove the schema from the *Table* attribute of the entity declaration. Here’s how:

```
//[Table(Name = "SalesLT.Product")]  
[Table(Name = "Product")]  
public partial class Product : INotifyPropertyChanging, INotifyPropertyChanged
```

Here, you only use the name of the table in the *Table* attribute.

Defining Entity Relationships

So far, you’ve only seen the specification and mapping attributes for a single entity. Because you have natural relationships between entities, you’ll need to be able to manage associations. This section will build upon the previous by adding new entities to a `DataContext`, demonstrating entity associations, and examining related attribute mappings.

Adding Multiple Entities to a DataContext

Adding entities for multiple database objects is as easy as adding the first *Product* entity in the previous section. You can open `AdventureWorks.dbml` from the Solution Explorer so you can see the design surface. Then open Server Explorer (Database Explorer in Visual C# Express) and navigate to the Tables

branch of the AdventureWorksLT database. Select all of the tables in Server Explorer, and drag and drop selected tables onto the AdventureWorks.dbml design surface. Depending on the size of your database, this process could take some time. An important point to remember is that you must save the *.dbml file after adding new entities because they won't generate code in *.designer.cs until you save. [Figure 3-5](#) shows the results, all of the new entities. The layout in your designer might look different because of screen size and positioning. Once the entities appear on your design surface, you are free to move them around as you prefer.

If you do this yourself, you might notice that now two identical entities correspond to the *Products* table, *Product* and *Product1*, which might occur if you left the original *Product* entity on the design surface. I removed the first *Product* entity before adding all of the entities you see in [Figure 3-5](#). If you forget to remove the previous entity and you end up with a second version after dragging and dropping all the entities from the database, the simplest solution is to remove both of the duplicate entities and re-add the table. That is, delete *Product* and *Product1* from the visual designer, and then drag and drop the *Product* table onto the visual designer again. This highlights a feature of the LINQ to SQL designer that accommodates using objects with the same identifiers. You'll also see this numbering scheme for a single entity with multiple associations to one other entity.

Demonstrating Entity Associations

[Figure 3-5](#) shows all of the associations between objects in the AdventureWorksLT database. You can see the associations between entities, defining one-to-many relationships where the open diamond is on the “one” side and the arrow points to the “many” side. To see what this means, look at the *Product* and *ProductCategory* tables in [Figure 3-6](#).

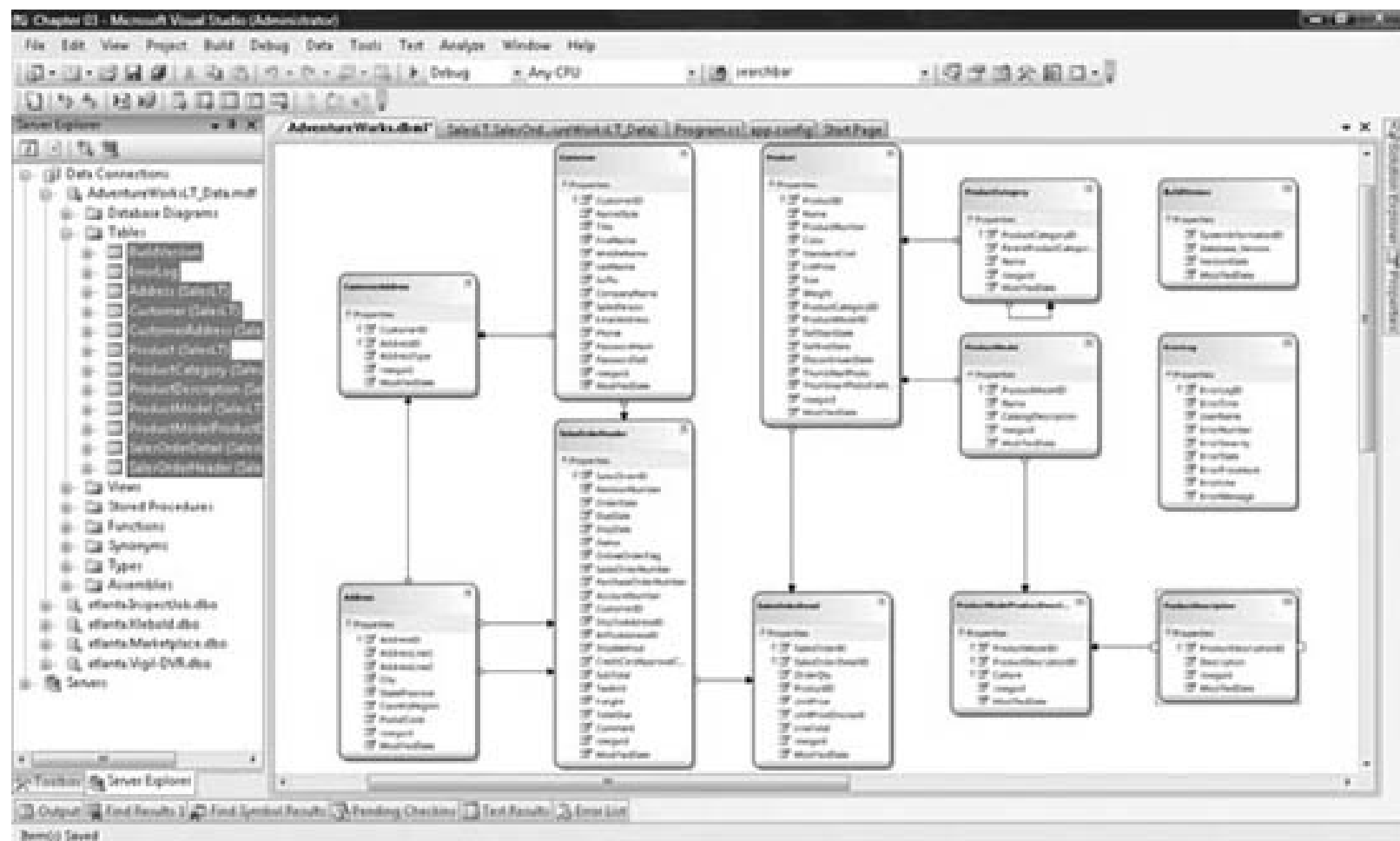


FIGURE 3-5 LINQ to SQL designer with full database

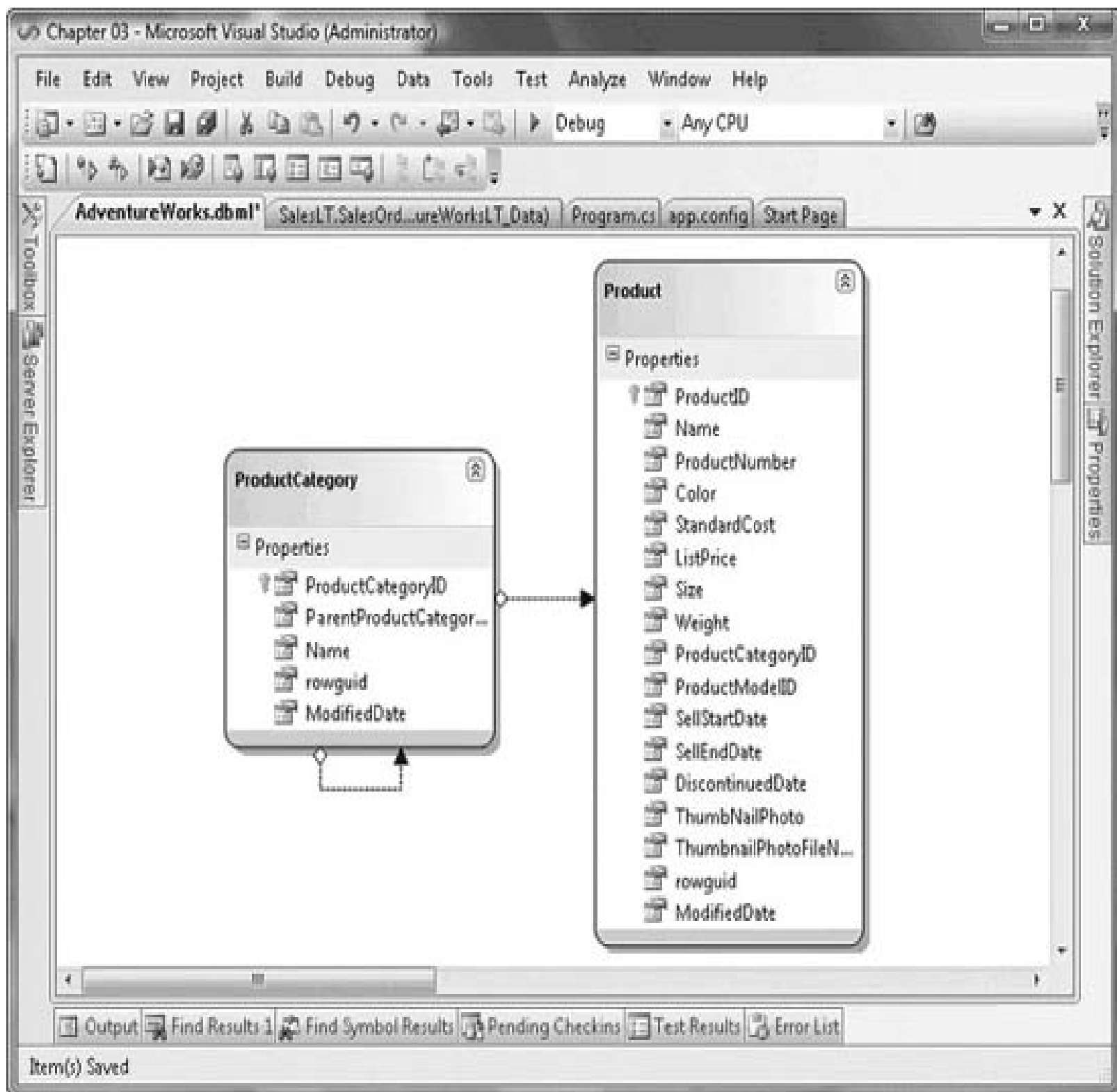


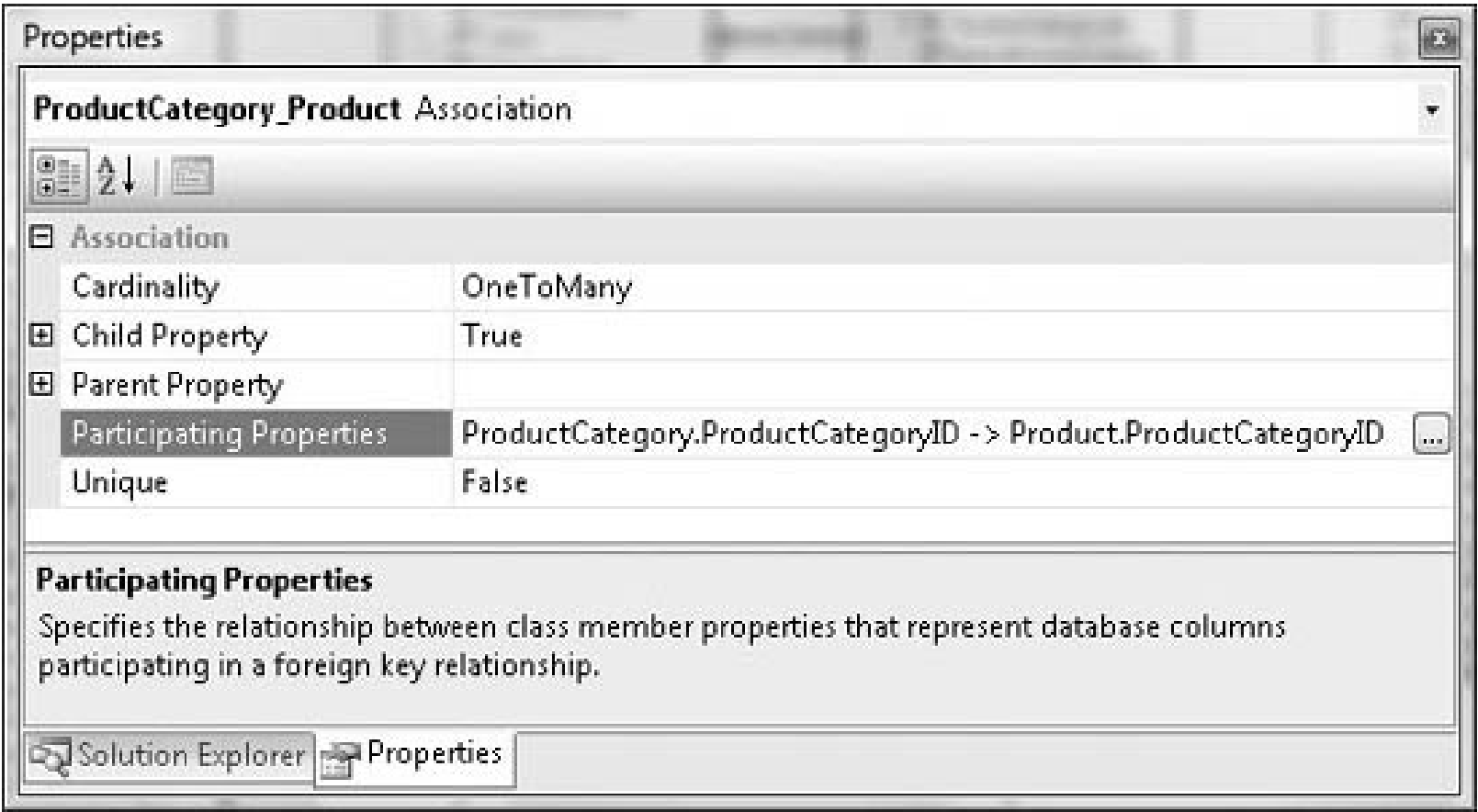
FIGURE 3-6 Entity associations

The *ProductCategory* entity has two associations, one on itself and another on the *Product* entity. The self-referencing entity supports a hierarchical relationship where a *ProductCategory* has 0 or more children that are each also a *ProductCategory* or subcategory. The association from *ProductCategory* to *Product* means that the *ProductCategory* can be applied to multiple *Products*. In other words, multiple *Products* belong to the same *ProductCategory*. If you click on the association line between *ProductCategory* and *Product*, you can see which properties establish the association, shown in [Figure](#)

3-7.

The *Participating Properties* property, shown in [Figure 3-7](#), is written in the direction of the arrow; *ProductCategory.ProductCategoryID* refers to *Product.ProductCategoryID*. From the database perspective, this might seem backwards because the foreign key relationship is from *ProductCategoryID* in the *Product* table to the *ProductCategoryID* in the *ProductCategory* table. Although this may be confusing, remember that the entities are objects that you code against, and the conceptual relationship is really that *ProductCategories* have *Products*, which is what you will write code for.

FIGURE 3-7 Participating Properties of a LINQ to SQL association



Handling Multiple Relationships Between Two Entities

If you recall from the previous section, I explained how you could have both *Product* and *Product1* entities in your DataContext because they refer to the same object and the *Product* table was added twice. You'll also see this situation on entity properties, but the reason is different. If one entity has two (or more) associations with one other entity, the associations will be named sequentially too. This situation happens in the AdventureWorksLT database between the *Address* and *SalesOrderHeader* tables, shown in [Figure 3-8](#), where the *SalesOrderHeader* has two foreign keys to the *Address* table, a *ShipToAddress* and a *BillToAddress*.

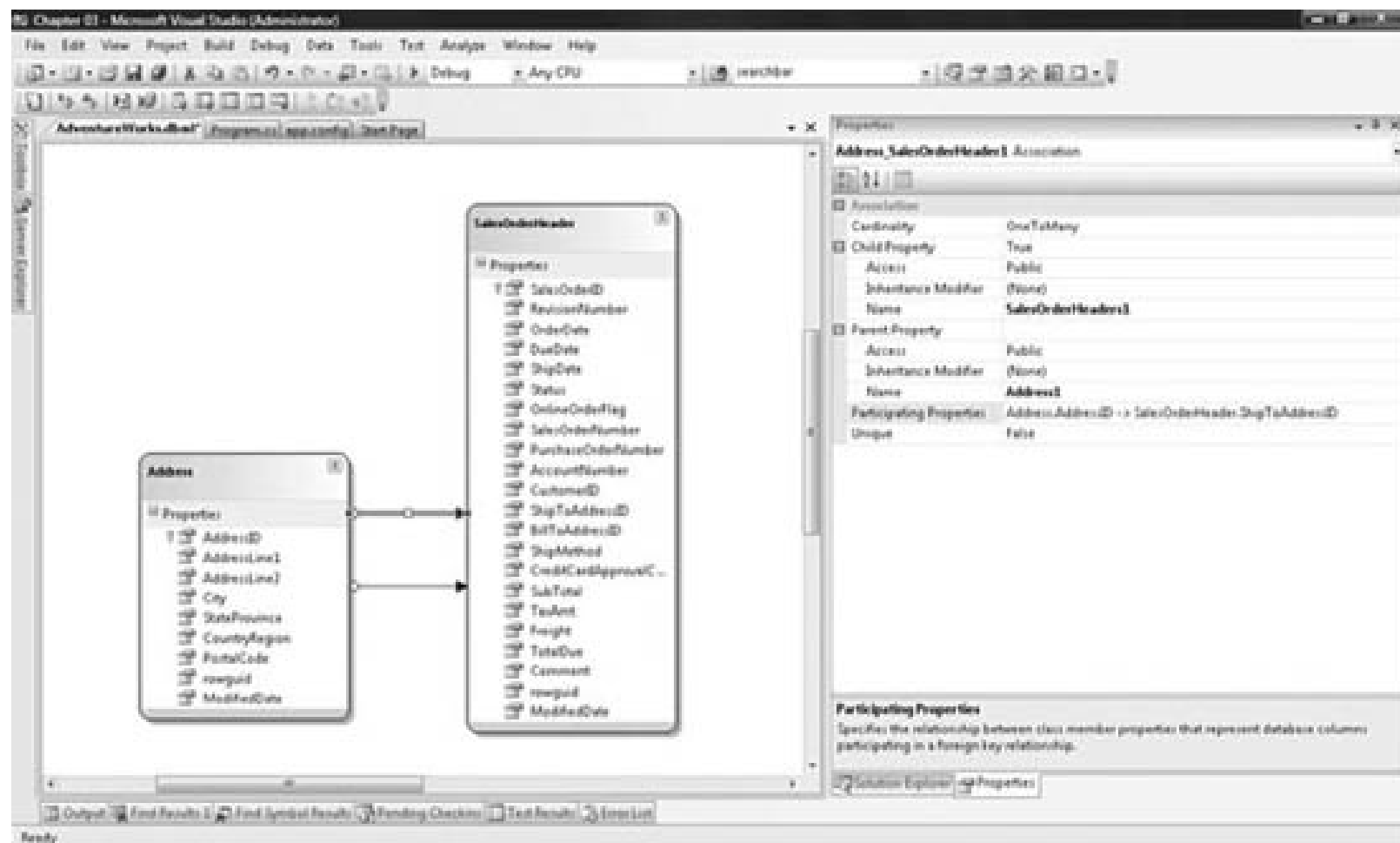


FIGURE 3-8 Multiple associations between two entities

As shown in [Figure 3-8](#), there are two associations between *Address* and *SalesOrderHeader*. The selected association, *Address_SalesOrderHeader1*, shows how the *AddressID* in *Address* refers to the *ShipToAddressID* in *SalesOrderHeader*. The problem is that the association appears as a property in the code, and it's confusing to tell what each property means. For example, the following code gets the first *SalesOrderHeader* from the database and reads the *Address1* and *Address*, which correspond to *ShipToAddress* and *BillToAddress*, respectively:

```
var ctx = new AdventureWorksDataContext();

var salesOrdHdr =
    ctx.SalesOrderHeaders.First();

var shipTo = salesOrdHdr.Address1;
var billTo = salesOrdHdr.Address;
```

The call to the *First* operator is equivalent to a SQL *Top 1* in a *select* clause, but that's not the purpose of the demo. Instead, observe the code that reads *Address1* and *Address* properties from the *salesOrdHdr* instance. The only way to know what these two properties mean is to go back to the *.dbml and to look at the properties for each entity, as shown in [Figure 3-8](#).

To fix this problem, modify the *Name* property underneath the Child Property and/or Parent Property. Since this property is from the perspective of the *Address* entity, indicated by the direction of the association arrow, Child Property is the *SalesOrderHeader*. Parent Property is from the perspective of the *SalesOrderHeader*, the child. Therefore, appropriate property names for the

Address_SalesOrderHeader1 association that you see in [Figure 3-8](#) might be *SalesOrdersShippedTo* for the Child Property and *ShipToAddress* for the Parent Property. The other association from *Address* to *SalesOrderHeader* is named *Address_SalesOrderHeader* (without a numeric suffix). Its Child Property could be named *SalesOrdersBilledTo*, and its Parent Property could be named *BillToAddress*.

One more note: When you're first designing a database, you might go through a couple iterations as you tweak your schema. If you delete and re-add objects, this destroys the customizations you've made on object properties. What I'll typically do is take advantage of VS 2008 task management and put a *// TODO:* comment in the code so that when the schema stabilizes, I can go back and update entities and referencing code. That's just a tip; the point is that you have to be cognizant of customizations and the fact that they are removed if you ever replace the entity or association in the *.dbml designer.

Now, let's dig a little deeper into attributes that make association mapping possible.

Examining Association Attributes

As you learned in a previous section, the mapping between database objects and entities occurs through attributes on DataContext objects, located in AdventureWorks.designer.cs for our examples. The specific attribute is *Association*, and participating LINQ to SQL types are *EntityRef* and *EntitySet*. The following code snippet demonstrates the attributes that define the association between *Product* and *ProductCategory*:

```
private EntityRef<ProductCategory> _ProductCategory;
[Association(
    Name="ProductCategory_Product",
    Storage="_ProductCategory",
    ThisKey="ProductCategoryID",
    IsForeignKey=true)]
public ProductCategory ProductCategory
```

Remember, you won't be able to see the entities in AdventureWorks.designer.cs until you save AdventureWorks.dbml. I've omitted a lot of code to focus on declaration of the property and its backing store. The backing store is type *EntityRef<ProductCategory>*, meaning that it is a reference to the *ProductCategory* object on the *one* side of the *one*-to-many relationship.

The *Association* attribute decorates the property that manages the relationship between entities. Effectively, the *Product* entity can hold a reference to a *ProductCategory* entity object. I discussed *Name* and *Storage* properties, which have the same purpose as the *Column* attribute described earlier. The *ThisKey* property specifies the key in this entity that holds the identity key to the *ProductCategory* being referred to. In addition to the property that defines the association, entities also have another property that actually holds the value of the key to the referred object, which is *ProductCategoryID* in this case. Having a property that holds a key is very convenient when interacting with the database because it means that all you need to keep track of in your code is the key, rather than the entire object being referenced. *IsForeignKey* states whether this property should be represented as a foreign key in the database. These properties help create foreign keys anytime you run the *CreateDatabase* command, described in an earlier section.

LINQ to SQL has attributes and code to define both sides of an association. So, as you might guess, the *ProductCategory* entity has code to define its association with the *Product* class, shown next:

```
private EntitySet<Product> _Products;
```

```
[Association(
    Name="ProductCategory_Product",
    Storage="_Products",
    OtherKey="ProductCategoryID")]
public EntitySet<Product> Products
```

In the preceding code, you can see that the *ProductCategory* entity, which references multiple *Product* entities, has a *_Products* backing store of type *EntitySet<Product>*. Similarly, the association on the *Products* property, which encapsulates the *_Products* field (backing store) has an *Association* attribute to define its relationship with the *Product* entity. *Name* and *Storage* are the same as defined for the *Column* property. *OtherKey* defines what key in *Products* will be used to refer to this entity. Notice also that the *Products* property is defined as an *EntitySet<Product>*, which can hold references to related *Product* entities.

Now that you know what services the DataContext offers, let's start using them to work with data, starting with LINQ to SQL queries in the next section.

A Closer Look at Queries

One of the more significant benefits of LINQ is a common syntax across all data sources. To see this, remember the query syntax you used in [Chapter 2](#): projections, filters, grouping, joins, and more. This SQL-like syntax is the same, regardless of the data source or provider. In fact, it is the same query syntax that you will use for LINQ to SQL.

This section won't rehash what you learned in [Chapter 2](#). Instead, it will highlight what is significant about using query syntax to work with LINQ to SQL. More specifically, you'll see how to use the DataContext to identify the entities to query. You'll also learn how to examine the materialization of your queries; that is, how LINQ to SQL translates your query syntax into a SQL statement that it will transmit to the database. Being able to examine the generated SQL can help you internalize what the query syntax means in terms of working with SQL Server. In a later chapter, when we discuss deferred execution and loading, examining query syntax will be an important tool.

Querying with the DataContext

As you learned in previous sections, the DataContext provides access to entities. This section will show you how to use the DataContext to access data. You'll find that it is similar to LINQ to Objects, except the object to query is provided via the DataContext. The following code demonstrates how to select *Product* records from the AdventureWorksLT database:

```
var ctx = new AdventureWorksDataContext();

IQueryable<Product> products =
    from product in ctx.Products
    where product.ListPrice < 100.00m
    select product;
```

After instantiating the DataContext, which you learned about in the previous section, the preceding code performs a query on the *Products* table. Notice how the return type is *IQueryable<T>* instead of *IEnumerable<T>*, which you learned about in [Chapter 2](#). All LINQ to SQL queries return *IQueryable<T>*, which has special functionality that enables queries to be translated into SQL and sent to SQL Server. In a later chapter, you'll see that *IQueryable<T>* is more general than just LINQ to SQL.

Now, notice that the data source for the query is *ctx.Products*. If you recall from the previous section, the *AdventureWorksDataContext* has a property named *Products* of type *Table<Product>*. This means you're querying through the property that gives you access to the underlying *Product* table in the SQL Server database.

If the preceding code doesn't run, it's possible that activity from the last section might have left code in *AdventureWorks.designer.cs* in an inconsistent state. It's easy to fix by opening the *AdventureWorks.dbml* file, deleting the *Product* entity, and re-adding the *Product* entity. Deleting and re-adding an entity is also the way you can refresh an entity if you've changed any of the schema at the database and need an update.

Because LINQ to SQL uses deferred execution, where a query doesn't actually happen until you request data, you won't see data for the products collection in the debugger until it is actually read by code. The following *foreach* loop causes the query to execute:

```
foreach (var product in products)
{
    Console.WriteLine(
        "Name: {0}, {1:C}",
        product.Name,
        product.ListPrice);
}
```

Once you access the first object, the *DataContext* will execute the query. You'll learn more about the deferred execution process in [Chapter 11](#). Deferred execution is a feature to enhance performance by not actually performing a query until it is ready.

Next, let's look at what is sent to the database for query execution.

Examining Generated SQL

If you had sufficient access to the database server, you could run a profiler to examine queries as they come across the wire. However, this is not always possible or convenient. Fortunately, LINQ to SQL gives you a logging capability that enables you to examine the SQL statements that LINQ to SQL generates. The following code shows how to use the *Log* property of the *DataContext* to examine SQL queries:

```
var ctx = new AdventureWorksDataContext();
ctx.Log = Console.Out;

var products =
    from product in ctx.Products
    where product.ListPrice < 100.00m
    select
        new
        {
            product.Name,
            product.ListPrice
        };
};
```

You can assign any *TextWriter* derived type to the *Log* property of the *DataContext* to access SQL queries. The preceding example assigns *Console.Out*, enabling output to the Console screen when the *foreach* loop executes. Notice how I decided to use an anonymous type to limit the number of parameters selected. I typically avoid projecting into anonymous types because my LINQ to SQL code executes in business objects, but it was helpful for this example to simplify the following logged output:


```

SELECT [t0].[Name], [t0].[ListPrice]
FROM [SalesLT].[Product] AS [t0]
WHERE [t0].[ListPrice] < @p0
-- @p0: Input Decimal (Size = 0; Prec = 31; Scale = 4) [100.00]
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build: 3.5.21022.8

```

There are some important points to be made of the preceding logged output in the areas of object specification, parameterization, and provider. Each object is surrounded by square brackets, ensuring there isn't any ambiguity in identifiers.

LINQ to SQL is inherently more secure than some hand-coded ADO.NET code because it parameterizes all input data. In the preceding example, *@p0* is the input passed to the *where* clause. You can see the parameter details under the SQL statement defining type and the input value *[100.00]*. Parameterizing user input helps avoid SQL Injection attacks that would be caused by string concatenation in ADO.NET. Of course, you can parameterize commands in ADO.NET, but the point to be made is that a developer who doesn't realize the hazards of SQL Injection can't accidentally make these mistakes with LINQ to SQL.

The *Context* defines the provider for interacting with the database. Currently, LINQ to SQL supports SQL Server 2000 and SQL Server 2005. The preceding example shows that, even though I'm using SQL Server 2005 Express, it uses the *Sql2005* provider. The *Model* parameter states that I'm using the *AttributedMetaModel*, meaning that the mapping between the database and my entities is done through .NET attributes, such as *Database*, *Table*, and *Column*, which you learned about in the previous section. Another model is *MappedMetaModel*, which means that you are using an external XML file to define the mapping between database and entities. You can learn how to use an external XML mapping file in [Chapter 7](#), where I also discuss the *SqlMetal* command-line tool.

This example displayed a simple *select* clause with a *where* statement, but the *Log* output gets much busier than this, which you'll learn in the next section.

Examining Query Results

Now that you know how to examine the DataContext output log, let's look at a few key queries that are essential to getting the most out of LINQ to SQL. In the following sections, you'll see grouping, joining, and *select many* statements, individually demonstrating what SQL will be sent to the database. In the following examples, you can assume that each of the queries has run through a *foreach* loop, causing the query to execute and produce the *Log* output that you'll see.

Examining SQL for Grouping

Given a hypothetical scenario where AdventureWorks has realized that providing a fair selection of colors is profitable, we're going to create a query that lets company managers see how they are doing in this area. The query will group products by color and then display the number of products for each color. The following query solves this problem for AdventureWorks managers:

```

var ctx = new AdventureWorksDataContext();
ctx.Log = Console.Out;

var colorDistribution =
    from product in ctx.Products
    group product by product.Color
    into productColors

```

```

select
    new
    {
        Color = productColors.Key,
        Count = productColors.Count()
    };

```

The preceding code is a *group by* statement that projects into an anonymous type with *Color* and *Count* properties. Notice the call to *Count* on the query continuation, *productColors*. This returns the count of all items in each group. *Count* is one of many LINQ to SQL operators that mirror equivalent SQL functions. For example, you can use *Average*, *Min*, *Max*, and more. You can see how this query is transformed into SQL after executing a *foreach* loop, shown next:

```

SELECT COUNT(*) AS [Count], [t0].[Color]
FROM [SalesLT].[Product] AS [t0]
GROUP BY [t0].[Color]
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build: 3.5.21022.8

```

Notice how the call to *Count* in the query is translated into an execution of the *Count* function in the SQL *select* clause. Next, you'll see what joins look like.

Examining SQL for Joins

You can perform both *inner* and *left outer* joins with query syntax joins. The *inner join* is based on identity equality between two entities. Here's an example that shows what categories a product belongs to:

```

var ctx = new AdventureWorksDataContext();
ctx.Log = Console.Out;

var productsWithCategories =
    from prod in ctx.Products
    where prod.ListPrice < 100.00m
    join prodCat in ctx.ProductCategories
    on prod.ProductCategoryID
    equals prodCat.ProductCategoryID
    select
        new
        {
            Product = prod.Name,
            Category = prodCat.Name
        };

```

The *join* in the preceding example is similar to what you've seen in the LINQ to Object examples in [Chapter 2](#), except that it uses the DataContext, *ctx*, to access the entities to query. The SQL shown next is transmitted to the database when the query executes:

```

SELECT [t0].[Name] AS [Product], [t1].[Name] AS [Category]
FROM [SalesLT].[Product] AS [t0]
INNER JOIN [SalesLT].[ProductCategory] AS [t1]
ON [t0].[ProductCategoryID] = ([t1].[ProductCategoryID])
WHERE [t0].[ListPrice] < @p0
-- @p0: Input Decimal (Size = 0; Prec = 31; Scale = 4) [100.00]
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build: 3.5.21022.8

```

This shows that a normal *join* in LINQ to SQL produces an *inner join* in SQL. The next example will

show you how to create a *left outer join*. The example considers the relationship between child and parent categories in the *ProductCategory* table where parent categories don't have parents; their *ParentProductCategoryID* is *null*. A normal *inner join* only returns 37 records, but 41 are in the *ProductCategory* table. The following query performs a *left outer join* so you can see all 41 records:

```
var ctx = new AdventureWorksDataContext();
ctx.Log = Console.Out;

var catsAndParents =
    from child in ctx.ProductCategories
    join parent in ctx.ProductCategories
      on child.ParentProductCategoryID
      equals parent.ProductCategoryID
    into catsAndParentsContinuation
    from childParent in catsAndParentsContinuation.DefaultIfEmpty()
    select
        new
        {
            Parent = childParent.Name,
            Child = child.Name
        };

```

As a refresher, you need to join into a *continuation* variable, *catsAndParentsContinuation* in this case. The key to the left *join* is calling the *DefaultIfEmpty* operator on the *continuation* variable, ensuring you get a record, even if it is *null*. The *childParent* variable holds parent records, even when a child doesn't have a parent. Here's the SQL generated from the preceding query:

```
SELECT [t1].[Name] AS [Parent], [t0].[Name] AS [Child]
FROM [SalesLT].[ProductCategory] AS [t0]
LEFT OUTER JOIN [SalesLT].[ProductCategory] AS [t1]
ON [t0].[ParentProductCategoryID] = ([t1].[ProductCategoryID])
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build: 3.5.21022.8

```

As you can see, this is a normal SQL *left outer join*. You can also create joins with *select many* queries, which is discussed next.

Examining SQL for Select Many

A *select many* clause can sometimes be easier to use than a *join* because it describes the hierarchical relationship of entities, allowing you to flatten the results into a single query. Used on a hierarchy, *select many* emits what is referred to as an old-style *join*; a *where* clause matches identifiers between tables when entities have an existing relationship. When *select many* is implemented between entities with no apparent relationship, you get a cross-*join*. Here's an example of a *select many* that produces an old-style *join* between *ProductCategory* and *Product* tables:

```
var ctx = new AdventureWorksDataContext();
ctx.Log = Console.Out;

var prodsAndCats =
    from cat in ctx.ProductCategories
    from prod in cat.Products
    select
        new
        {
            Category = cat.Name,

```

```
Product = prod.Name
};
```

Notice how the query follows the same relationship established by the association between *ProductCategory* and *Product*. The *from* for *prod* uses *cat.Products* as its source. Here’s the old-style *join* output:

```
FROM [SalesLT].[ProductCategory] AS [t0], [SalesLT].[Product] AS [t1]
WHERE [t1].[ProductCategoryID] = [t0].[ProductCategoryID]
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build: 3.5.21022.8
```

This shows that there is a *where* clause that uses the matching IDs for the *ProductCategory* and *Product* tables. You can also get a cross-*join* by combining *select many* with *join*. Here’s an example that uses the *ProductModel*, *ProductCategory*, and *Product* tables to produce a cross-*join*:

```
var ctx = new AdventureWorksDataContext();
ctx.Log = Console.Out;

var modelsAndColors =
    from model in ctx.ProductModels
    from color in ctx.Products
    join prod in ctx.Products
    on color.ProductID equals prod.ProductID
    select
        new
        {
            Model = model.Name,
            Product = prod.Name,
            Color = color.Color
        };
};
```

The combination of *select many* clauses and the *join* causes LINQ to SQL to generate a cross-*join*. As queries become more complex, you’ll need to pay attention to the results in generated SQL, especially if you didn’t expect a cross-*join*. Just a warning: if you run the preceding code, it will generate many results, so you might want to limit the number of records returned with the *Take* operator, shown next:

```
foreach (var prodColor in modelsAndColors.Take(25))
{
    Console.WriteLine(
        "Model: {0}, Product: {1}, Color: {2}",
        prodColor.Model,
        prodColor.Product,
        prodColor.Color);
}
```

The *Take* operator will return the *Top 25* records. The following SQL shows this along with the cross-*join* created by the query:

```
SELECT TOP (25) [t0].[Name] AS [Model], [t2].[Name] AS [Product], [t1].[Color]
FROM [SalesLT].[ProductModel] AS [t0]
CROSS JOIN [SalesLT].[Product] AS [t1]
INNER JOIN [SalesLT].[Product] AS [t2] ON [t1].[ProductID] = [t2].[ProductID]
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build: 3.5.21022.8
```

The preceding SQL will return the *Top 25* records from a cross-*join* of *ProductModel* and *Product* records. The next section demonstrates how to perform an *IN* query.

Implementing IN Queries

The LINQ query syntax doesn't include an *IN* operator that allows you to return *all* the records where a field matches an item in a list. However, this is easy to do by calling *Contains* on any collection. Here's an example that uses the *Contains* method to produce an *IN* query:

```
var ctx = new AdventureWorksDataContext();
ctx.Log = Console.Out;

var colorList =
    new List<string>
    {
        "Black",
        "Silver",
        "White"
    };

var blackSilverWhite =
    from prod in ctx.Products
    where colorList.Contains(prod.Color)
    select prod;
```

Notice the *colorList* in the preceding code and how it has three members: *Black*, *Silver*, and *White*. This defines the filter conditions to be used in the *where* clause of the query. The *where* clause checks whether the current product being evaluated, *prod*, matches items that *colorList* contains. Here's the SQL generated by this query:

```
SELECT [t0].[ProductID], [t0].[Name], [t0].[ProductNumber],
       [t0].[Color], [t0].[StandardCost], [t0].[ListPrice],
       [t0].[Size], [t0].[Weight], [t0].[ProductCategoryID],
       [t0].[ProductModelID], [t0].[SellStartDate], [t0].[SellEndDate],
       [t0].[DiscontinuedDate], [t0].[ThumbNailPhoto],
       [t0].[ThumbnailPhotoFileName], [t0].[rowguid], [t0].[ModifiedDate]
FROM [SalesLT].[Product] AS [t0]
WHERE [t0].[Color] IN (@p0, @p1, @p2)
-- @p0: Input NVarChar (Size = 5; Prec = 0; Scale = 0) [Black]
-- @p1: Input NVarChar (Size = 6; Prec = 0; Scale = 0) [Silver]
-- @p2: Input NVarChar (Size = 5; Prec = 0; Scale = 0) [White]
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build: 3.5.21022.8
```

I've formatted the output to make it more presentable, but the important part is the *where* clause, demonstrating that it uses *IN* to filter records. You can see that the definitions of the parameters match the members of the *colorList* from the LINQ to SQL query.

Inserting, Updating, and Deleting

In addition to reading data, you can also insert, update, and delete records. Because of deferred execution, changes aren't written to the database until you actually submit them, with a call to the *SubmitChanges* method on the *DataContext*. You'll see how to submit your changes to the database and to make modifications in the following sections.

Inserting New Data

If you've looked at the data in the AdventureWorks database, you'll notice that a lot of the products are related to biking. One noticeable omission is a category for MP3 players. The following code remedies

this by adding an “MP3 Players” subcategory under the *Accessories* category in the *ProductCategory* table:

```
var ctx = new AdventureWorksDataContext();

var mp3Cat =
    new ProductCategory
    {
        Name = “MP3 Players”,
        ParentProductCategoryID =
            (from cat in ctx.ProductCategories
             where cat.Name == “Accessories”
             select cat.ProductCategoryID)
            .Single(),
        ModifiedDate = DateTime.Now,
        rowguid = Guid.NewGuid()
    };

ctx.ProductCategories.InsertOnSubmit(mp3Cat);

// don't forget to persist changes
ctx.SubmitChanges();
```

When instantiating the new *ProductCategory* class just shown, you can see that I used a query to get the *ProductCategoryID* for the parent, *Accessories*, category, and I assign it to the *ParentProductCategoryID* of the new object. Alternatively, I could have performed the query before instantiating the object, assigned the *ProductCategoryID* to a variable, and then used the variable to set the *ParentProductCategoryID*. However, this demonstrates more ways that you can use queries to declaratively code what you want to happen. *Single* is an operator that will return an object when you only expect to receive one value.

Calling *InsertOnSubmit* lets the *DataContext* know that it has a new record to save, but doesn’t perform the action of saving that record. You could potentially have multiple items to add, and you don’t want every line of code to emit database changes. The name *InsertOnSubmit* reflects the nature of deferred execution. Once you call *SubmitChanges*, then the new record will be written to the database.

In addition to the *InsertOnSubmit*, you can also add an entire list of items to a database by calling *InsertAllOnSubmit*. Next, you’ll learn how to modify existing data.

Updating Existing Data

Maybe the term *MP3* for the new product category was too specific. Perhaps it would be more useful to generalize the category name and call it “Mobile Devices,” which would include smart phones and other multifunction devices. Here’s an example that changes the *ProductCategory* name, showing you how to update a record:

```
var ctx = new AdventureWorksDataContext();

var prodCat =
    (from cat in ctx.ProductCategories
     where cat.Name.StartsWith(“MP3”)
     select cat)
    .SingleOrDefault();

if (prodCat != null)
```

```
{
    prodCat.Name = "Mobile Devices";

    ctx.SubmitChanges();
}
```

The preceding example retrieves the *ProductCategory* instance from the database, modifies the retrieved *ProductCategory* object, and then saves the changes back to the database when *SubmitChanges* is invoked. The preceding example filtered the result with a call to *StartsWith*, but it's more common to filter on an ID that the application keeps track of. Since IDs change and there are so many options for filtering data, I'll show you different examples. Notice that I used *SingleOrDefault* to get an instance to one object. While I expect one record to be returned, it's possible that there won't be a match. Using *SingleOrDefault* will return the default value of the object, and since entities are reference types, the default will be *null*. Therefore, the code checks for *null* before going further to avoid a *NullReferenceException*. The call to *SingleOrDefault* will throw an *InvalidOperationException* if more than one record is returned, so you should consider an exception handling strategy appropriate for your situation.

Next, you'll learn how to delete a record from the database.

Deleting Existing Data

You've seen how to insert and update a database record. Now, you're going to learn how to remove a record from the database. In this example, we'll delete the Mobile Devices record that was modified in the previous section; management hasn't decided to go with the new product category yet. Here's how to perform a delete operation:

```
var ctx = new AdventureWorksDataContext();

var prodCat =
    (from cat in ctx.ProductCategories
     where cat.Name == "Mobile Devices"
     select cat)
     .SingleOrDefault();

if (prodCat != null)
{
    ctx.ProductCategories.DeleteOnSubmit(prodCat);

    ctx.SubmitChanges();
}
```

The preceding code retrieves the entity instance associated with the Mobile Devices product category. Then it calls *DeleteOnSubmit*. Again, the name reflects what is really happening because the delete action doesn't occur until you call *SubmitChanges*.

Another way to modify the database is by attaching objects, which is discussed next.

Attaching Objects

Sometimes, you might already have a reference to an existing object in memory or have all of the information associated with that object, including keys. That would enable you to use the *Attach* method rather than doing an update. Assuming that the following code belongs to a method where an existing

object is passed in as a parameter, you could attach a new object with that data:

```
ctx = new AdventureWorksDataContext();

var newProdCat =
    new ProductCategory
    {
        ProductCategoryID = oldProdCat.ProductCategoryID,
        ParentProductCategoryID =
            oldProdCat.ParentProductCategoryID,
        Name = "MP3 Players",
        ModifiedDate = oldProdCat.ModifiedDate,
        rowguid = oldProdCat.rowguid
    };

ctx.ProductCategories.Attach(newProdCat);

newProdCat.Name = "Music Devices";

ctx.SubmitChanges();
```

In the preceding code, *oldProdCat* holds the information for the existing objects. It's important that you have the key or *Attach* won't work. After creating the new *ProductCategory* instance, *newProdCat*, and attaching it, you can modify the new object. Remember, the object tracking service of the *DataContext* keeps track of changes. Therefore, the *DataContext* must first know that the object exists, via *Attach*, and then it can keep track of changes.

The *Attach* method has a couple more overloads that help in various situations. Here's an overload that takes a second parameter of type *bool*, enabling you to treat the whole object as modified at the time you attach it:

```
var newProdCat =
    new ProductCategory
    {
        ProductCategoryID = oldProdCat.ProductCategoryID,
        ParentProductCategoryID =
            oldProdCat.ParentProductCategoryID,
        Name = "Music Devices",
        ModifiedDate = DateTime.Now,
        rowguid = oldProdCat.rowguid
    };

ctx.ProductCategories.Attach(newProdCat, true);

ctx.SubmitChanges();
```

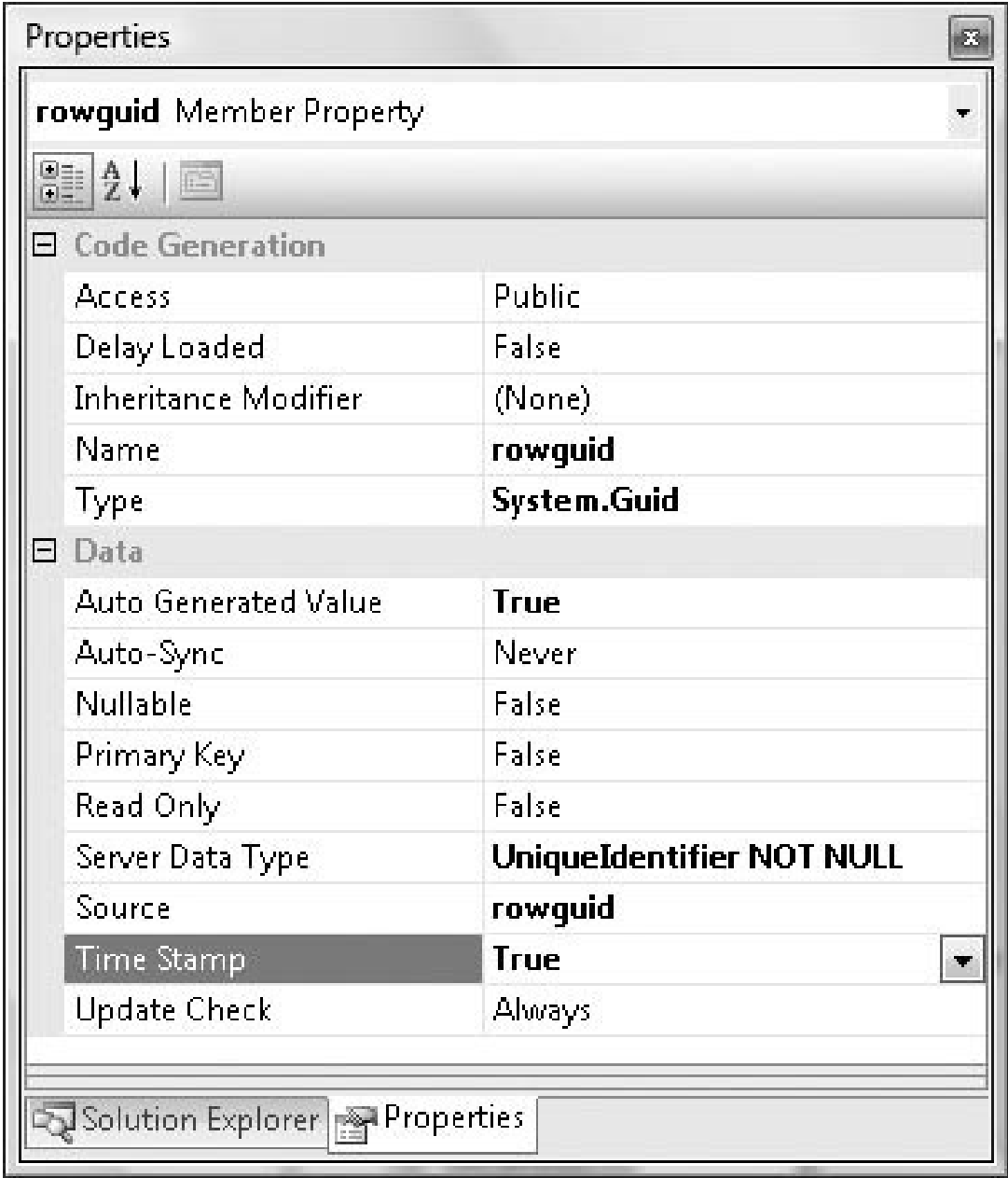
The first thing to notice is that the *newProdCat* instantiates with *Name* set to the modified value. In the previous *Attach* example, this was set to the original object value. Next, you can see that the overload of *Attach* has a second parameter, set to *true*, which treats the object as being modified. Also, there isn't any code to modify *newProdCat* because all of its non-key fields are used to update the object.

Although this example looked easy, using the *Attach* overload with the second *bool* parameter can be tricky when the underlying table has *Timestamp* or *Unique Identifier* columns. In the case of AdventureWorks tables, they have *Unique Identifier* columns named *rowguid*. When you create entities in the designer directly from the database and try to use *Attach* with a second *bool* parameter, you'll receive an exception message of "An entity can only be attached as modified without original state if it declares a

version member or does not have an update check policy.” To fix this problem, select the column, *rowguid* in this case, in *.dbml and set the *Time Stamp* property for the column to *True*, as shown in [Figure 3-9](#).

Remember that if you ever replace this entity with a new version from the database, you’ll need to select the column in the *.dbml and reset the *Time Stamp* property to *True* as shown in [Figure 3-9](#).

FIGURE 3-9 Setting Time Stamp column property to True for attaching modified entities



The next *Attach* overload takes a new version of an object and the old version. Assuming that you had a reference to the old version of the *ProductCatalog* entity, here’s an example that attaches the new entity to perform an update:

```
var newProdCat =
    new ProductCategory
    {
        ProductCategoryID = oldProdCat.ProductCategoryID,
        ParentProductCategoryID =
            oldProdCat.ParentProductCategoryID,
        Name = "Music Devices",
```

```
        ModifiedDate = DateTime.Now,  
        rowguid = oldProdCat.rowguid  
    };
```

```
ctx.ProductCategories.Attach(newProdCat, oldProdCat);
```

```
ctx.SubmitChanges();
```

Notice that *newProdCat* has its name set to *Music Devices*, which is the modification we need to make because we assume it was set to something else such as *MP3 Players*. Then you can pass the *newProdCat* and *oldProdCat* to *Attach* to make the changes. While the previous *Attach* updated all properties of the record, this *Attach* (with the *oldProdCat*) will only update the properties that were changed. Having both objects enables the DataContext object tracking service to figure out the differences between objects.

The examples in this section built queries that generate SQL statements that are sent to the database. Alternatively, you can use stored procedures, which is covered next.

Implementing Stored Procedures and Functions

In addition to LINQ query syntax, you can also use stored procedures and functions. The following sections show you how to add stored procedures and functions to your project and to call them through code.

Stored Procedures

To see how stored procedures work, you'll need to add the following *GetProducts* stored procedure to your database:

```
CREATE PROCEDURE dbo.GetProducts  
AS  
    select ProductID, Name  
    from SalesLT.Product
```

The purpose of this stored procedure is to return the *ProductID* and *Name* of each record in the *Product* database. Before you can use a stored procedure, you must add it to your DataContext. To do so, open the AdventureWorks.dbml file to see the designer, select the stored procedure in Server Explorer (Data Explorer in Express editions), and drag and drop the stored procedure onto the AdventureWorks.dbml design surface. Save the .dbml to update the DataContext. You'll see the stored procedure in the Methods pane in [Figure 3-10](#).

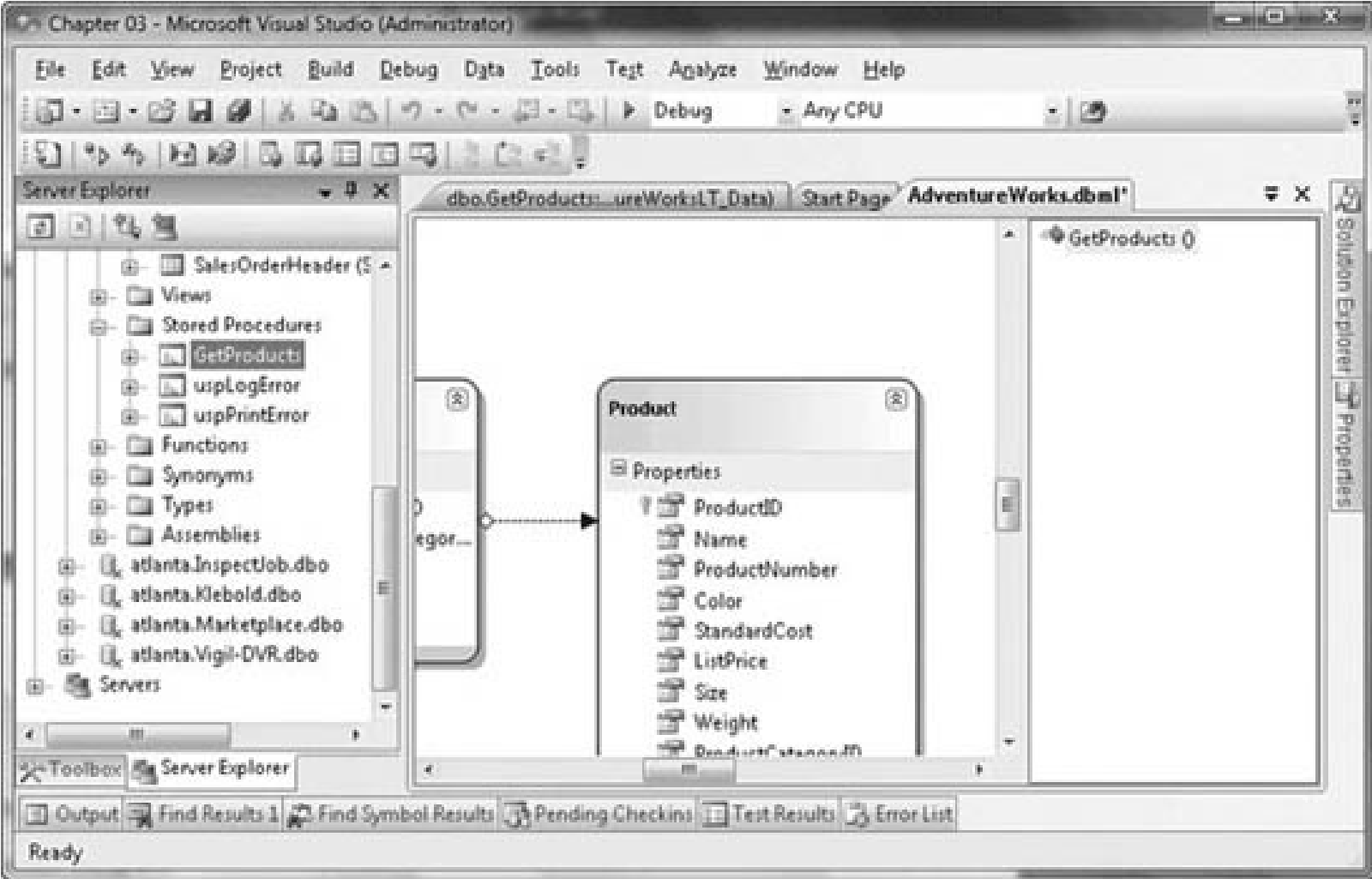


FIGURE 3-10 Adding a stored procedure to your DataContext

To use the stored procedure, all you need to do is call it like a normal method off the DataContext instance. It will appear in IntelliSense like any other method. Here's how you can use it:

```
var ctx = new AdventureWorksDataContext();

var products = ctx.GetProducts();

foreach (var product in products)
{
    Console.WriteLine(
        "ID: {0}, Name: {1}",
        product.ProductID,
        product.Name);
}
```

Other stored procedures such as inserts, updates, and deletes can be added to your project and used the same way. You have IntelliSense support in VS 2008 that allows you to find them and to see what parameters and return types are required.

Functions

You can use functions very similarly to how you use stored procedures. In the next example, you'll use the *ufnGetAllCategories* function to get a list of functions. Just as with the stored procedure in the previous section, open the AdventureWorks.dbml file, locate the *ufnGetAllCategories* function (under the

Functions branch of the AdventureWorksLT database in Server Explorer), and drag and drop it onto the AdventureWorks.dbml design surface. After that, you'll be able to see the function in the Methods pane. The following example demonstrates how to use it:

```
var ctx = new AdventureWorksDataContext();

var categories =
    from cat in ctx.ufnGetAllCategories()
    where cat.ProductCategoryName.Contains("bike")
    select cat;

foreach (var category in categories)
{
    Console.WriteLine(
        "ID: {0}, Category: {1}",
        category.ProductCategoryID,
        category.ProductCategoryName);
}
```

In the preceding example, you can see how I used the function as a data source in the *select* statement. No surprises—you can call functions just like methods.

On occasion, you might need to forgo query syntax and stored procedures and functions and run some raw SQL yourself. The next section shows you how.

Executing Raw SQL

Although query syntax, stored procedures, and functions enable a lot of productivity when interacting with SQL Server, you could occasionally need to write your own SQL instead. For example, you might be in a hurry and prefer not to translate a block of SQL from a migration of an older application—especially if it's complex; you might want to leave the translation for later. What if there are restrictions on what objects you can put in a database and you don't have permissions to add functions or stored procedures? What about the situation where you have a particularly complex query and are having trouble getting it to work the way you want in query syntax? I've been able to accomplish everything I've wanted with query syntax, but, admittedly, have been tempted on occasion to just write the SQL and get it over with. If you've found a case where you would prefer to use SQL, then you can do so with the *ExecuteQuery* method of the DataContext. Here's an example that retrieves a specific product with the *ExecuteQuery* command:

```
var ctx = new AdventureWorksDataContext();

string getProductString =
    @"select ProductID, Name
    from SalesLT.Product
    where ProductID = {0}";

var products = ctx.ExecuteQuery<Product>(getProductString, 707);
foreach (Product product in products)
{
    Console.WriteLine(
        "ID: {0}, Name: {1}",
        product.ProductID,
        product.Name);
}
```

In the *getProductString* variable, you can see that there is a placeholder parameter *{0}* for specifying the *ProductID* to search for. In the call to *ExecuteQuery*, the method call passes in the argument, *707*, which replaces the parameter placeholder.

Since the *ExecuteQuery* is parameterized, it's safer than concatenating strings for specifying parameters. However, it is still fraught with danger if you concatenate any type of input into the string itself.

As with *ExecuteQuery*, you can also use the *DataContext* object's *ExecuteCommand* for any other type of SQL statement. This also opens the possibility that you can do more than just SQL queries; you can also invoke other SQL Server commands, such as *CREATE*, *ALTER*, *DROP*, and so on. Here's an example of *ExecuteCommand*:

```
ctx.ExecuteCommand("Create table test ( MyColumn int );");
```

The preceding example creates a new table, named *test*, but you can also use *ExecuteCommand* for *insert*, *update*, and *delete* operations.

In addition to querying entities that mirror database objects, you also have the capability to create entities that derive from existing entities. The next section shows you how to do this.

Entity Inheritance

Database records are often categorized and classified in some way to differentiate between records of different types. For example, products have categories and models. The categorization is specified relationally and surfaces in LINQ to SQL via associations that require joins or *where* clauses to filter the records you need. However, you're coding in an object-oriented language, and sometimes it might be more comfortable to look at entities through their object relationships, rather than their relational relationships.

LINQ to SQL has a feature for specifying inheritance relationships between entities. It's based on a single table model where one column acts as a discriminator that specifies what the true object type should be. The example in this section will use the *Address* table and employ the *CountryRegion* column as the discriminator. *CountryRegion* has three distinct values: *Canada*, *United Kingdom*, and *United States*. Using *CountryRegion* as the discriminator, we'll create entities in the *AdventureWorksDataContext* for *Canada*, *United Kingdom*, and *United States*.

To add new entities that aren't part of the database, you'll need to use the Object Relational Designer, which appears on the Toolbox whenever you select the design surface of *AdventureWorks.dbml*. The following steps will walk you through the process of creating entities that derive from *Address*:

1. Open *AdventureWorks.dbml*, navigate to the *Address* entity, and open the Toolbox to reveal LINQ to SQL objects. [Figure 3-11](#) displays what your screen should look like.
2. Drag and drop a class onto the design surface below *Address*.

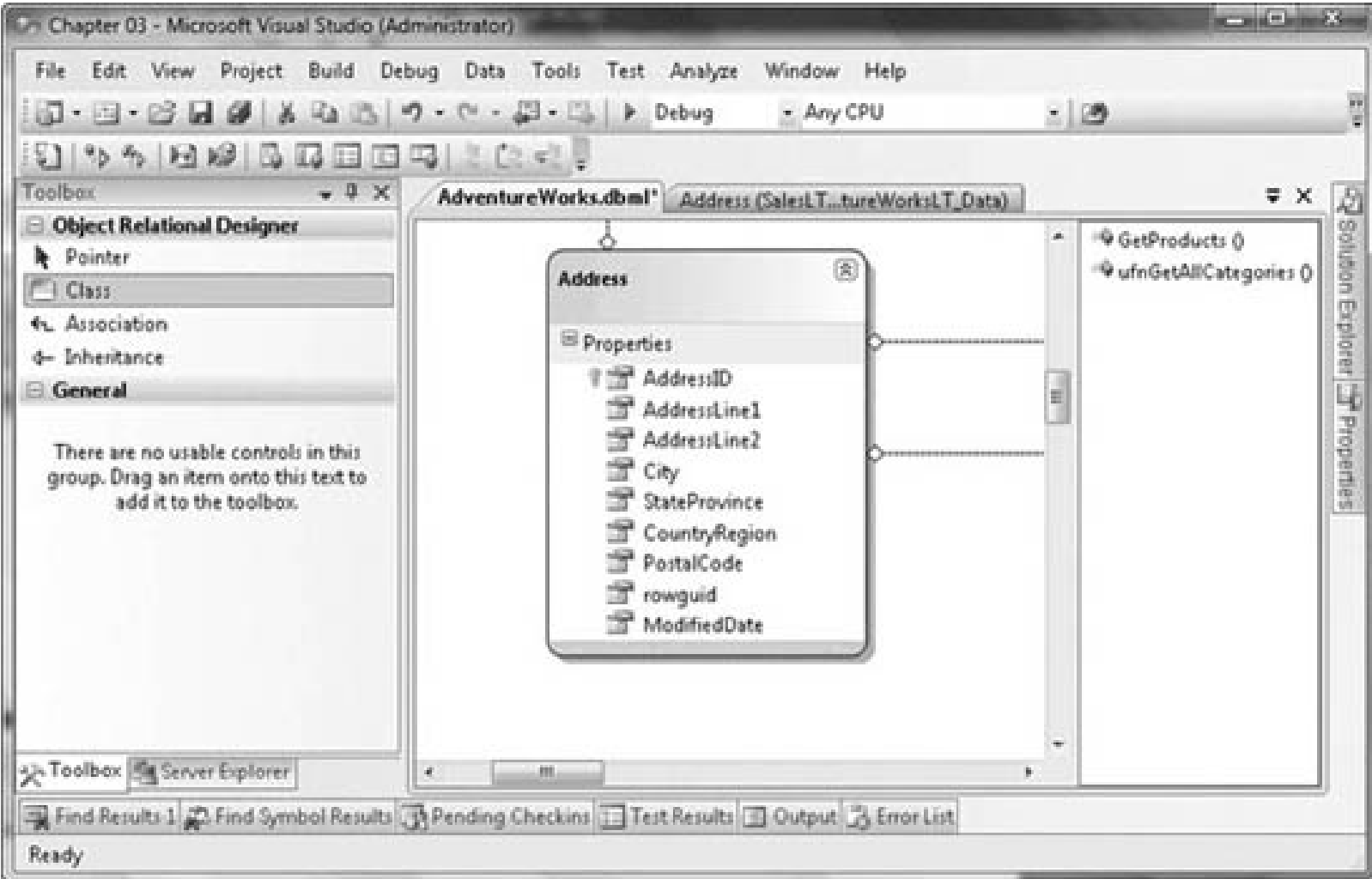
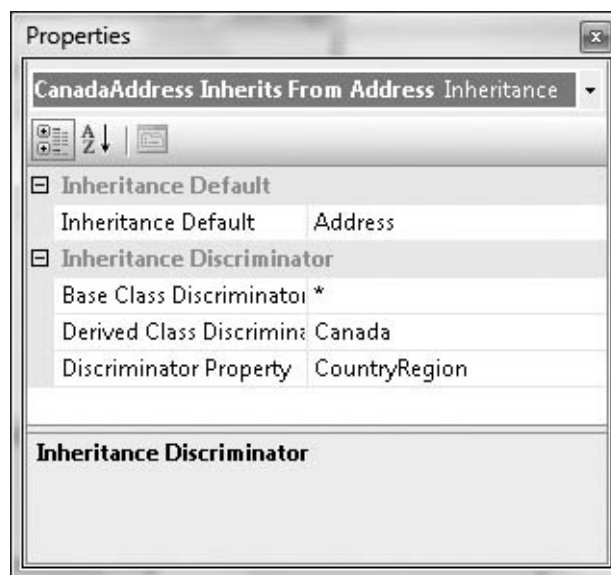


FIGURE 3-11 The LINQ to SQL Object Relational Designer

3. With the new class selected, open the property window and change the name to **CanadaAddress**.
4. Add two new classes under *Address* named **UnitedKingdomAddress** and **UnitedStatesAddress**, the same way you did for *CanadaAddress*.
5. For each new class, click on the Inheritance icon in the Toolbox, click on the derived class, that is, *CanadaAddress*, and then click on *Address*. This will show an arrow from the derived class to the base class.
6. Select the inheritance relation from *Canada* to *Address* and set its properties: Inheritance Default to **Address**, Base Class Discriminator Value to *, Derived Class Discriminator Value to **Canada**, and Discriminator Property to *CountryRegion*. [Figure 3-12](#) shows what the settings should be.
7. Select the inheritance relations for *United Kingdom* and *United States*, one at a time, and set their properties the same as the *CanadaAddress* inheritance relation, except that Derived Class Discriminator Value should be set to **United Kingdom** and **United States**, respectively. The Derived Class Discriminator Value property must match the values being used in the database, so double-check the *CountryRegion* column in the *Address* table to make sure your spelling is correct.

FIGURE 3-12 Inheritance settings in the Properties window



With the Discriminator Property set to *CountryRegion*, *Address* table records with *CountryRegion* set to *Canada* will become *CanadaAddress* objects. Similarly, when the *CountryRegion* is set to either *United Kingdom* or *United States*, the entity will become a *UnitedKingdomAddress* or *UnitedStatesAddress*, respectively. Since the Inheritance default property of each inheritance relationship is set to *Address*, any value of *CountryRegion* that is not set to *Canada*, *United Kingdom*, or *United States*, will become an *Address* entity, which is the base class. [Figure 3-13](#) shows the resulting new entities and their inheritance relationship with the *Address* table.

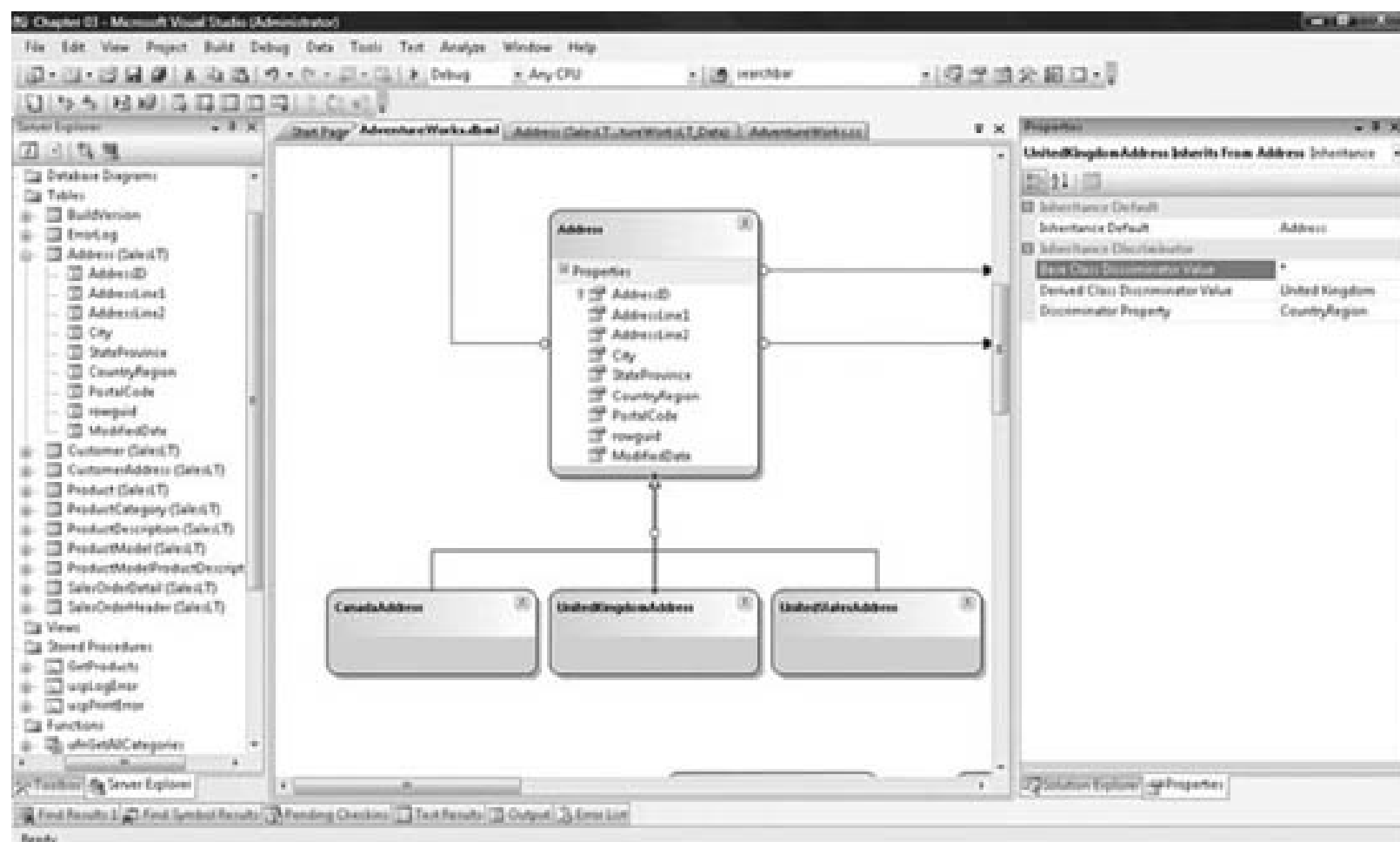


FIGURE 3-13 Entity inheritance in the Visual Designer

When saved, these designer changes add an *InheritanceMapping* attribute to the *Address* class in

AdventureWorks.designer.cs for each derived entity and the *Address* entity:

```
[Table(Name="SalesLT.Address")]
[InheritanceMapping(Code = "*", Type = typeof(Address), IsDefault=true)]
[InheritanceMapping(Code = "Canada", Type = typeof(CanadaAddress))]
[InheritanceMapping(Code="United Kingdom", Type=typeof(UnitedKingdomAddress))]
[InheritanceMapping(Code="United States", Type=typeof(UnitedStatesAddress))]
public partial class Address : INotifyPropertyChanging, INotifyPropertyChanged
```

Code is the discriminator code, which is *CountryRegion*. *Type* is the entity type that will be instantiated when a database record has a discriminator value matching the *Code* of the same attribute. The *IsDefault* property means the entity specified by *Type* will be used when the *Code* doesn't match any codes in other *InheritanceMapping* attributes.

Now that you know how to implement inheritance, there is one side-effect of using the visual designer. If you ever remove and replace any of the objects in an inheritance relationship, remember to re-create the inheritance relationships because they will be gone when you remove one or more associated objects.

Summary

Now, you've learned about how LINQ to SQL allows you to use SQL Server as a data source. Central to LINQ to SQL is the DataContext, and you now know about the services it provides in the way of connection management, object tracking, object mapping, and query execution.

You saw how LINQ to SQL query syntax is the same as that used by LINQ to Objects, which is an important LINQ concept. This chapter dug deeper, showing you the results from when LINQ to SQL translates your queries into SQL statements.

You learned how to perform inserts, updates, and deletes. You also saw how to use stored procedures and functions. If you ever have a need, there was a section on implementing raw SQL queries. Finally, you saw how LINQ to SQL enables you to take a more object-oriented approach to data with entity inheritance features.

In the next chapter, we'll build upon the LINQ story by showing you how to use LINQ with existing ADO.NET technology—DataSets.

PART II

LINQ to Any Data Source

CHAPTER 4

Working with ADO.NET Through LINQ to DataSet

CHAPTER 5

Programming Objects with LINQ to Entities

CHAPTER 6

Programming Objects with LINQ to XML

CHAPTER 7

Automatically Generating LINQ to SQL Code with SqlMetal

CHAPTER 4

Working with ADO.NET Through LINQ to DataSet

DataSets have been used by .NET developers for years. Entire applications are built using the DataSet for working with data sources. Microsoft recognized the widespread use of DataSets in .NET application development and included a LINQ provider appropriately named LINQ to DataSet.

When working with LINQ to DataSet, you'll want to read from and write to DataSet tables. You can also work with strongly typed DataSets. There are also operations you'll need to use on occasion, including the ability to compare DataRow objects and create DataTables from query results. You'll learn about each of these capabilities in this chapter.

Setting Up the DataSet

To get started, you'll need a DataSet to query. Continuing with the same theme from previous chapters, the DataSet will contain the *Product* and *ProductCategory* tables from the AdventureWorksLT database. When using a DataSet, remember to add a *using* declaration for the *System.Data.SqlClient* namespace. [Listing 4-1](#) contains a common method that will be used throughout this chapter to get a working DataSet.

Listing 4-1 Populating a DataSet with Adventure-WorksLT Database Objects

```
public static DataSet CreateDataSet()
{
    var connStr =
        @"Data Source=.\sqlexpress;
        AttachDbFilename=""C:\Program Files\Microsoft SQL Server\
MSSQL.1\MSSQL\Data\AdventureWorksLT_Data.mdf"";
        Initial Catalog=AdventureWorksLT_Data;
        Integrated Security=True";

    var prodCmd =
        @"select ProductID, Name, ProductCategoryID
        from SalesLT.Product";

    var prodCatCmd =
        @"select ProductCategoryID, Name
        from SalesLT.ProductCategory";

    var conn = new SqlConnection(connStr);

    var daProducts = new SqlDataAdapter(prodCmd, conn);
    var daProductCategories = new SqlDataAdapter(prodCatCmd, conn);

    var ds = new DataSet("AdventureWorksLT");

    daProducts.Fill(ds, "Products");
    daProductCategories.Fill(ds, "ProductCategories");

    ds.Relations.Add(
        ds.Tables["ProductCategories"].Columns["ProductCategoryID"],
        ds.Tables["Products"].Columns["ProductCategoryID"]);

    return ds;
}
```


The connection string in [Listing 4-1](#) has a couple anomalies that you'll need to correct before using the code. First, replace it with a connection string to your own database, which is probably different. Second, the connection string is formatted so that it will fit on the page in the book, so you'll need to eliminate any line wrap, being sure not to accidentally remove spaces such as the one that should be between the name of the Program Files directory.

Let's start with querying the contents of a DataSet.

Querying a DataSet

When querying a DataSet, you need know how to transform the DataSet objects so they are queryable with LINQ. You'll also need to use generic helper methods to read DataRow information. In addition, you can use LINQ to modify DataSet data. The first section begins by specifying the data for querying DataSet tables.

Querying DataSet Tables

DataSet queries are made on DataTables. Since DataTables aren't *IEnumerable<T>* or *IQueryable<T>*, you'll need to convert them to an *IEnumerable<T>* collection that you can query. The following example queries the DataSet returned by calling *CreateDataSet* from the previous section:

```
DataSet ds = CreateDataSet();

EnumerableRowCollection<DataRow> products =
    from product in ds.Tables["Products"].AsEnumerable()
    select product;
```

Here, you can see that the query source is the *Products* DataTable. The call to *AsEnumerable* returns an *EnumerableRowCollection<TRow>*, which derives from *IEnumerable<T>*.

The *AsEnumerable* method is an extension method of the *DataTableExtensions* class, which belongs to the *System.Data* namespace and resides in the *System.Data.DataSetExtensions.dll* assembly. Therefore, you'll want to add a *using* declaration for *System.Data* and, if it's not there already, a project reference for the *System.Data.DataSetExtensions* assembly.

Once you've queried data from a DataSet, you'll want to read the results.

Reading DataRow Results

In the previous section, you saw how the results of a LINQ to DataSet query are produced as a collection of DataRow objects. In a typical ADO.NET implementation, you would use convenience methods of the DataRow to read the data. The DataRow methods are weakly typed, meaning that you receive an object that must be converted to the type you need. Additionally, if the object returns *DBNull*, you have to convert that to *null*. Recognizing this, the LINQ to DataSet implementation includes a *DataRowExtensions* class with generic convenience methods for accessing DataRow columns. The following *foreach* loop shows how to use the *Field* method to read the *Name* column from each *Product* that was retrieved via the query in the previous section:

```
foreach (var product in products)
{
    Console.WriteLine(
        "Name: " + product.Field<string>("Name"));
}
```

```
}
```

Using *product.Field<string>("Name")*, you can read the *Name* column from the *DataRow* instance, *product*, resulting in a *string* type variable. If the column were to allow *null* values, such as *null* for a *DateTime*, you could use a nullable type, such as *product.Field<DateTime?>("MyNullableDate")*.

You can also use *Field* inside of queries. Here's an example of using *Field* to filter on *Product* records whose names start with the letter *R*:

```
var ds = CreateDataSet();

var prodTbl = ds.Tables["Products"];

var products =
    from product in prodTbl.AsEnumerable()
    where product.Field<string>("Name").StartsWith("R")
    select product;
```

In this example, the *Field* extension method pays big dividends because the query would otherwise be cumbersome to write with additional logic to handle *DBNull* and type conversions.

In addition to reading *DataSet* data, you can also modify it.

Modifying a DataSet

Just as with other data sources, you need a way to modify the data in addition to reading it. To facilitate modifications, the *DataRowExtensions* class has a sister extension method to *Field*, called *SetField*. Just like *Field*, it is a generic method, encapsulating the details of *null* data and enabling type safety. The following example shows how to add a new record to a table with *SetField*:

```
var ds = CreateDataSet();

var prodTbl = ds.Tables["Products"];

DataRow dr = prodTbl.NewRow();

dr.SetField<string>("Name", "Zune");

prodTbl.Rows.Add(dr);
```

The preceding example demonstrates how you can use *SetField* on a *DataRow*. The call to *NewRow* obtains a reference to a new row, having the *Product* table schema. The call to *SetField* sets the *Name* column, which is type *string*, to the value *Zune*. The generic approach, implemented through *SetField*, helps you avoid the extra lines of code that you would have needed to type to handle null values and type conversions. As you might already know, a *DataSet* is an in-memory image of the data. So, calling *SetField* will only operate on the in-memory representation, consistent with how a *DataSet* works. You would call *Update* on a *SqlDataAdapter*, as has always been done with *DataSets*, to persist *DataSet* contents.

This example showed you how to work with a normal (weakly typed) *DataSet*, but you can also use LINQ to *DataSet* on strongly typed *DataSets*.

Working with Strongly Typed DataSets

If you work with strongly typed DataSets, you can still use LINQ. This section will describe how to create a strongly typed DataSet containing *Product* and *ProductCategory* tables and then show how to query that strongly typed DataSet.

Creating a Strongly Typed DataSet

Creating a strongly typed DataSet takes only a few minutes with VS 2008. The following steps show you how to create a strongly typed DataSet that contains the *Product* and *ProductCategory* tables from the AdventureWorks database:

1. Create a new console project in VS 2008.
2. Right-click on the project, select Add | New Item, select Data in the Categories list (Express editions don't have a Categories list, so proceed), select DataSet in the Templates list, name the DataSet **AdventureWorks.xsd**, and click the Add button. This will open the DataSet designer.
3. Open Server Explorer, find the Tables branch of the AdventureWorks database, select the *Product* and *ProductCategory* tables, and drag-and-drop the *Product* and *ProductCategory* tables onto the DataSet designer.

Now, you have a strongly typed DataSet as shown in [Figure 4-1](#).

The strongly typed DataSet is called *AdventureWorks*, and it contains a *Product* and *ProductCategory* table with relationships, as shown in [Figure 4-1](#). Now you can query the strongly typed DataSet.

Querying a Strongly Typed DataSet

One of the conveniences of working with strongly typed DataSets is that it's easy to access tables without a lot of extra code for indexing and conversion. The following example demonstrates how to query the strongly typed DataSet created in the last section, with database objects from the AdventureWorks database:

```
var prodTbl = new AdventureWorks.ProductDataTable();
var prodTblAdapter = new ProductTableAdapter();
prodTblAdapter.Fill(prodTbl);

var products =
    from product in prodTbl
    where product.ListPrice < 150.00m
    select product;
foreach (var product in products)
{
    Console.WriteLine(product.Name);
}
```

To get this code to run, remember to add *using* declarations for the containing namespace and for the namespace of the table adapters, which are *Chapter_04* and *Chapter_04.AdventureWorksTableAdapters*, respectively for my code.

To access the table, you use the DataSet, *AdventureWorks*, and access the Product table, *ProductDataTable*. Then instantiate the TableAdapter, *ProductTableAdapter*. The *ProductTableAdapter* only needs a reference to the *ProductDataTable* instance for a data *Fill*. It already has a connection to the database, whose connection string was defined in app.config when the strongly typed DataSet was created in the previous section.

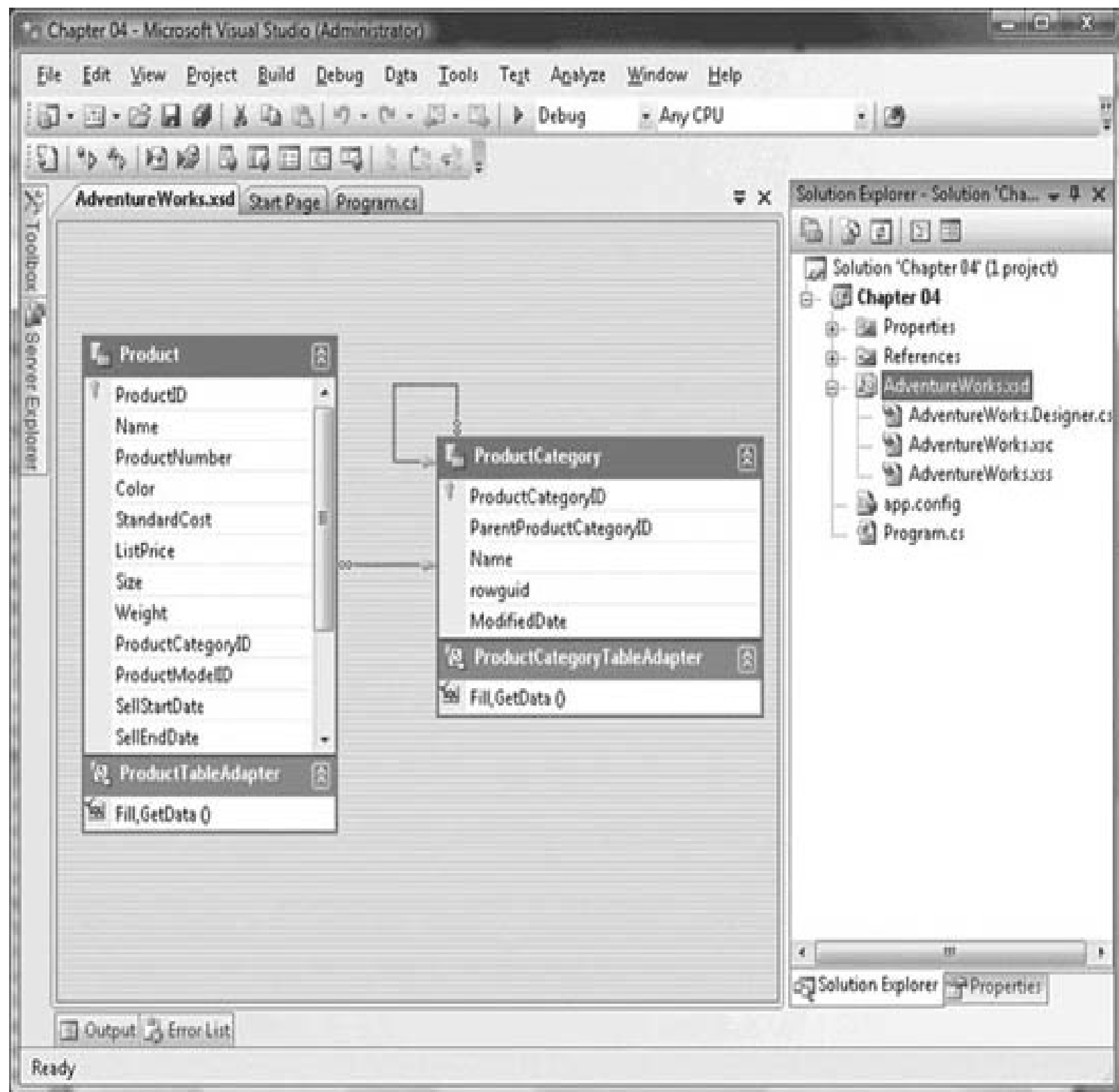


FIGURE 4-1 A strongly typed DataSet

Notice how the query can access properties like *ListPrice* without needing to use the *Field* extension method. Another example of strongly typed access is the *Name* field in the *foreach* loop, which doesn't need to use the *Field* extension method either.

In addition to querying DataSets, you can use special methods to manipulate DataSet data, which is discussed next.

Special DataSet Operations

You saw one of the extension methods of the *DataTableExtensions* class earlier—*AsEnumerable*. There

are two other methods that you should know about too: *AsDataView* and *CopyToDataTable*. The *AsDataView* extension method allows you to generate DataViews from LINQ to DataSet query results, and the *CopyToDataTable* extension method allows you to copy LINQ to DataSet results to a DataTable. The following sections show how you can use these extension methods.

Creating Data Views from LINQ to DataSet Queries

Instead of manually filtering and sorting for a DataView, you can perform those operations in a LINQ to DataSet query and then generate a DataView. The following example shows how this works:

```
var prodTbl = new AdventureWorks.ProductDataTable();
var prodTblAdapter = new ProductTableAdapter();
prodTblAdapter.Fill(prodTbl);
```

```
var products =
    from product in prodTbl
    where product.ListPrice < 150.00m
    orderby product.ListPrice descending
    select product;
```

```
DataView prodDV = products.AsDataView();
```

```
var prodEnumerator = prodDV.GetEnumerator();
```

```
while (prodEnumerator.MoveNext())
{
    Console.WriteLine(
        (prodEnumerator.Current as DataRowView)
        .Row.Field<string>("Name"));
}
```

The example shows the same query that was used in the previous section on strongly typed DataSets, with the addition of an *orderby* clause so we can have both sorting and filtering operations in the query. The query result, *products*, is used to generate a DataView, which will have the sorting and filtering operations. The example needed to dig into the DataView to extract results, but you would probably be assigning the DataView to a UI control's *DataSource* property instead.

Copying LINQ to DataSet Queries to DataTables

The typical way to populate a DataSet table is by using either a DataAdapter or TableAdapter. However, LINQ to DataSet offers another way, via the *CopyToDataTable* extension method, from the *DataTableExtensions* class. The following code shows how to create a DataTable with a LINQ to DataSet query:

```
var prodTbl = new AdventureWorks.ProductDataTable();
var prodTblAdapter = new ProductTableAdapter();
prodTblAdapter.Fill(prodTbl);
```

```
var products =
    from product in prodTbl
    where product.ListPrice < 150.00m
    orderby product.ListPrice descending
    select product;
```

```
DataTable productsDT = products.CopyToDataTable();
```

```
foreach (DataRow product in productsDT.Rows)
{
    Console.WriteLine(product.Field<string>("Name"));
}
```

After performing the query, which returns an *IEnumerable<DataRow>*, you can call *CopyToDataTable* with the query results, *products*. This creates a *DataTable* that you can add to a *DataSet*, bind to a UI control, or perform any other action that you might need with a *DataTable*.

Summary

Since so many applications have used *DataSets*, Microsoft added a LINQ provider named LINQ to *DataSet*. This chapter showed you how to query a *DataSet* and then explained the *Field* and *SetField* extension methods, which help you access *DataRow* data in a type safe way and handle null data more elegantly. You also learned how to use LINQ to *DataSet* with strongly typed *DataSets*. In addition to querying and modifying *DataSet* objects, you can also use a couple extension methods to create *DataView* and *DataTable* objects from LINQ to *DataSet* queries.

CHAPTER 5

Programming Objects with LINQ to Entities

LINQ and its ability to query multiple data sources inherently reduce the impedance mismatch between your code and the data sources you query. However, in many cases the storage and schema of the data must still be accounted for in code, meaning that you still have lingering impedance mismatch. For example, you still must reason from a relational perspective with LINQ to SQL and from a hierarchical perspective with LINQ to XML. It might be nice to have a LINQ provider that reduces the impedance mismatch even further by providing either an object-based data source or an object model to reason about your data with.

One LINQ provider, LINQ to Entities, aims to meet the need for a data source that exposes an object-oriented entity model for you to work with. Instead of reasoning about your data from the storage medium, you can work directly with objects. LINQ to Entities allows you to work with objects via a mapping mechanism, which you'll learn about in this chapter. You'll learn how to create an entity model, which is an object-oriented view of the data, directly from a database. You'll also see how to build entities first and then to map the entities to database tables. Most of all, you'll query and work with entities with the familiar LINQ query syntax you've learned and continued to use since [Chapter 2](#).

Now that you're curious about LINQ to Entities, let's start off by defining exactly what an entity is.

Introduction to Entities

An entity is a high-level object that represents some form of data. From the perspective of the ADO.NET Entity Framework (AEF), for which LINQ to Entities is the provider, an entity is a member of an Entity Relationship Model (ER Model). In 1976, Dr. Pin-Shan (Peter) Chen wrote a paper titled “The Entity Relationship Model – Toward A Unified View of Data,” which has become one of the most-cited papers in the field of computer science. AEF is based on Dr. Chen's work. Therefore, the entities queried with LINQ to Entities are based upon an ER Model of a database. [Figure 5-1](#) shows parts of the AdventureWorksLT database as an ER Model.

This model happens to map exactly to the AdventureWorksLT database schema, mostly because I generated it in Visio via reverse engineering. Some coders work with data from the perspective of their database, which is often referred to as a bottom-up approach. Other software engineers prefer a more object-oriented approach, allowing them to reason about data from the perspective of the customer requirements where the software development life cycle process flows top-down into an object model. Nonetheless, the boxes in [Figure 5-1](#) are entities, and the directed arrows between them are relationships (or associations). The AEF uses the term “Entity Data Model” (EDM) synonymously with “ER Model”; for consistency with Microsoft usage, which you'll encounter more often in practice, I'll use the acronym “EDM” for the rest of this chapter.

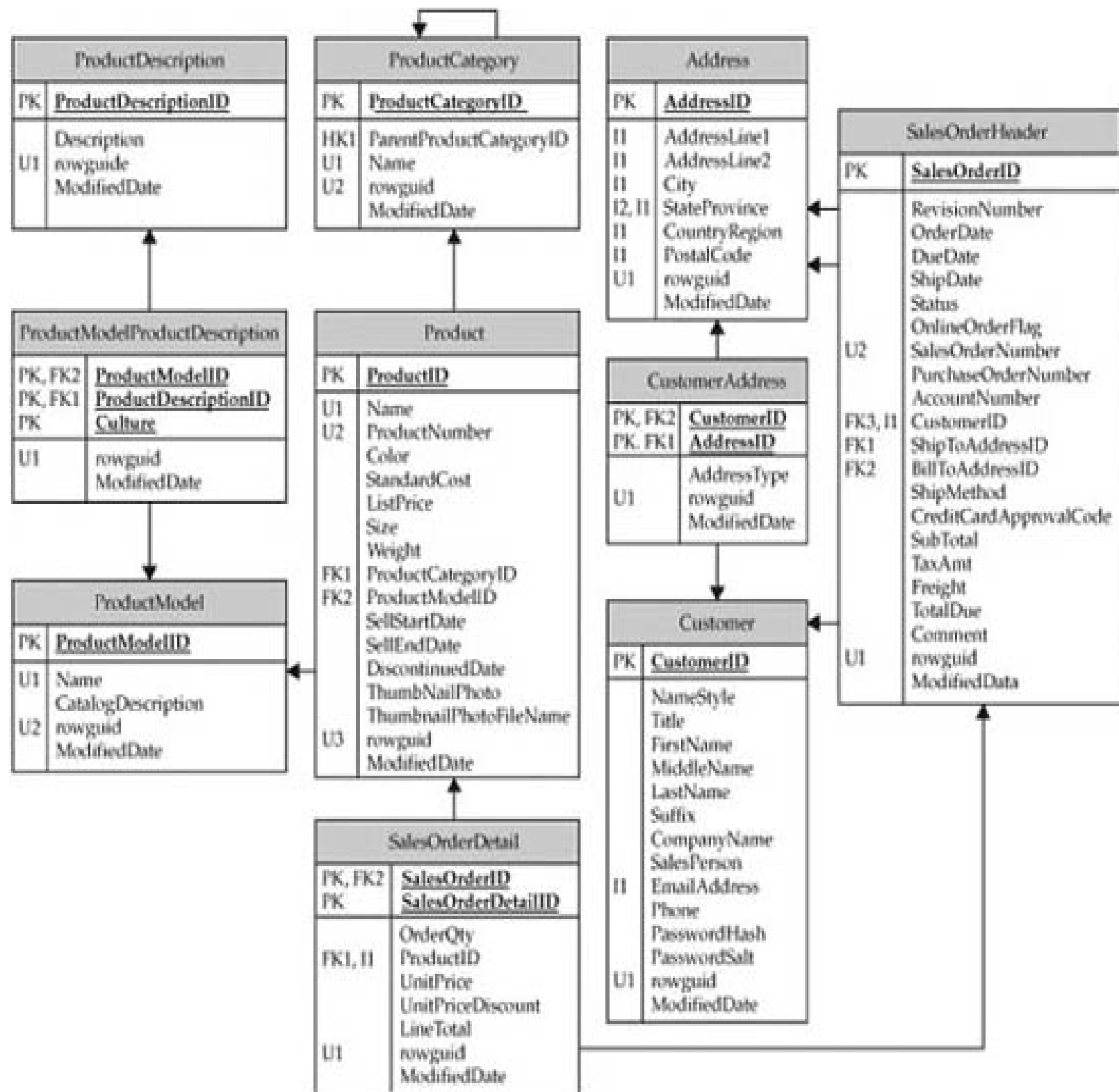


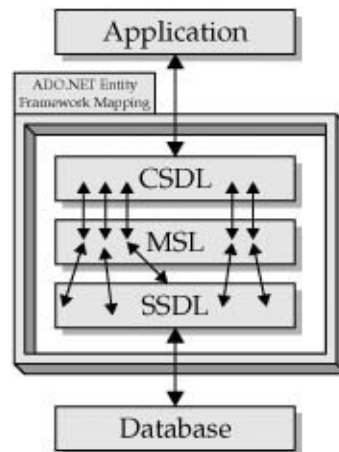
FIGURE 5-1 ER Model of the AdventureWorksLT database

The ADO.NET Entity Framework (AEF) Architecture

Part of the data design process, aligned with Dr. Chen's approach, includes a conceptual, a logical, and a physical design process, each with levels of abstraction between the model and database itself. The conceptual model is the highest level in the design process, giving you a general idea of what data you need in your system. The logical model is a more refined step in the process where relationships between entities become more defined. Finally, the physical model maps directly to the specific DBMS that you are working with, many with their own idiosyncrasies that require a unique physical design step.

The architecture of a system using AEF is somewhat similar to the data design process that incorporates conceptual, logical, and physical design. It consists of three mapping layers, defined via XML documents: Conceptual Schema Definition Language (CSDL), Mapping Schema Language (MSL), and Storage Schema Definition Language (SSDL). The CSDL file defines entities that you work with in your code; SSDL corresponds directly to the database schema; and MSL is a mapping between CSDL and SSDL. [Figure 5-2](#) shows the relationship of these documents.

FIGURE 5-2 ADO.NET Entity Framework application architecture



As [Figure 5-2](#) shows, the SSDL is closest to the database, meaning that it is also the document that matches the database most closely. The arrows going into and out of the MSL demonstrate how it maps from CSDL to SSDL. If the CSDL had a one-to-one match with the database schema, as in [Figure 5-1](#), I would have shown that as direct arrows that match exactly in the MSL. Instead, the arrows, in [Figure 5-2](#), point in many different directions, meaning that the CSDL doesn't have to match to SSDL. More importantly, the CSDL can represent an object-oriented view of the data, and the MSL will take care of the translation between objects and relational data. [Figure 5-2](#) shows that the application communicates with the AEF through the CSDL, which consists of entities queried in your LINQ code.

When working with VS 2008, you aren't forced to work with the CSDL, MSL, and SSDL XML files. Instead, you have a visual designer, which (along with LINQ to Entities) is the focus of this chapter. The next section shows you how to add an EDM to your application, which contains CSDL, MSL, and SSDL.

Creating an Entity Data Model (EDM)

VS 2008 has an EDM item that you can add to any project, making it easy to get started with AEF. The following steps explain how to add an EDM to your project:

1. Create a new Console project (we'll keep it simple).
2. Right-click on the project, select Add | New Item, and select ADO.NET Entity Model. Name the item **AdventureWorks.edmx**, and click the Add button. You'll see the Entity Data Model Wizard, shown in [Figure 5-3](#).
3. As shown in [Figure 5-3](#), you have a choice of generating the EDM from the database or starting with an Empty Model. For simplicity, choose Generate From Database, which sets up the EDM with entities matching one-to-one with database tables. Then click the Next button, and you'll see the next page of the Entity Data Model Wizard, shown in [Figure 5-4](#).
4. In [Figure 5-4](#), the drop-down list of data connections contains all of the database connections that you've defined in Server Explorer. You can see that I selected the AdventureWorksLT_Data.mdf connection, which was added previously. On my



FIGURE 5-3 The Entity Data Model Wizard: Choose Model Contents with Generate From Database

system, the name of the database file, AdventureWorksLT_Data.mdf, was established when I installed the database. The name could be different on your system if you have a different version of the database installed for a different version of SQL Server. Since the Entity Data Model Wizard uses the physical database file name for several default names, which you'll see in upcoming steps, the names in this book can differ from what you'll see on your system. I'll identify where default names come from so you can synchronize what you see on your system with the descriptions and figures in this book. You can click the New Connection button to add a new database to the list. The AdventureWorksLT_Data.mdf connection on my machine was created with Windows Integrated Security for login credentials, which is why the connection string security text is grayed out. If the connection had been created with User ID and Password credentials, the connection string text would be active, and you would have the option of excluding or including sensitive data (User ID and Password). Notice the AdventureWorks.csdl, AdventureWorks.ssdl, and AdventureWorks.msl files in the connection string,



Choose Your Data Connection

Which data connection should your application use to connect to the database?

AdventureWorksLT_Data.mdf

New Connection...

This connection string appears to contain sensitive data (for example, a password), which is required to connect to the database. However, storing sensitive data in the connection string can be a security risk. Do you want to include this sensitive data in the connection string?

☐ No, exclude sensitive data from the connection string. I will set it in my application code.

☐ Yes, include the sensitive data in the connection string.

Entity connection string:

```
metadata=res://*/AdventureWorks.cSDL|res://*/AdventureWorks.ssd|res://*/AdventureWorks.msl;provider=System.Data.SqlClient;provider connection string='Data Source=.\sqlexpress;AttachDbFilename="C:\Program Files\Microsoft SQL Server\MSSQL.1\MSSQL\Data\AdventureWorksLT_Data.mdf";Initial Catalog=AdventureWorksLT_Data;Integrated
```

☒ Save entity connection settings in App.Config as:

AdventureWorksLT_DataEntities

< Previous

Next >

Finish

Cancel

FIGURE 5-4 The Entity Data Model Wizard: Choose Your Data Connection

corresponding to the CSDL, SSDL, and MSL files, respectively, that I discussed in the previous section on AEF architecture. The *.csdl, *.ssdl, and *.msl files are named, after the *.edmx file name entered in Step 3, which is AdventureWorks.edmx. Finally, you can choose to save your connection string settings in app.config. The default connection string name appends “Entities” to the database file name, without extension, and you can change it to whatever you like. After you’ve set data connection options, click Next and you’ll see the Choose Your Database Objects window, shown in [Figure 5-5](#).

5. The Choose Your Database Objects window in [Figure 5-5](#) lets you select the database object that you want to be a part of the EDM. I’ve selected all Tables, Views, and Stored Procedures, but you could pick and choose the ones you need. The default name of the Model Namespace is the name of the physical database file, with “Model” appended, which will be the name of the model in the Model Browser, discussed in the next step. You can change the Model Namespace name if



Choose Your Database Objects

Which database objects do you want to include in your model?

- ☒ Tables
- ☒ Views
- ☒ Stored Procedures

Model Namespace:

AdventureWorksLT_DataModel

< Previous

Next >

Finish

Cancel

FIGURE 5-5 The Entity Data Model Wizard: Choose Your Database Objects

you like. Click the Finish button, which lets VS 2008 create your EDM. At this point, the Entity Data Model Wizard creates the EDM, resulting in the project items you see in [Figure 5-6](#).

The windows in [Figure 5-6](#) show the tools available to you in managing an EDM. The Model Browser allows you to view each of the entities and associations through the AdventureWorksLT_DataModel branch, and the AdventureWorksLT_DataModel.Store branch lets you work with the physical database. The Mapping Details is a graphical tool for mapping entities to database objects. When I show you how to build an EDM from objects, you'll learn how to use the objects in the Toolbox. Finally, the designer in the middle of the screen is a diagram of entities in your EDM. Selecting an entity invokes the context-

sensitive Mapping Details window with details for the selected entity. Not shown in [Figure 5-6](#) is the Properties window, which allows you to rename the entity, set access, make the class abstract, set a base type, and add documentation. The Properties window works the same as all other project types.

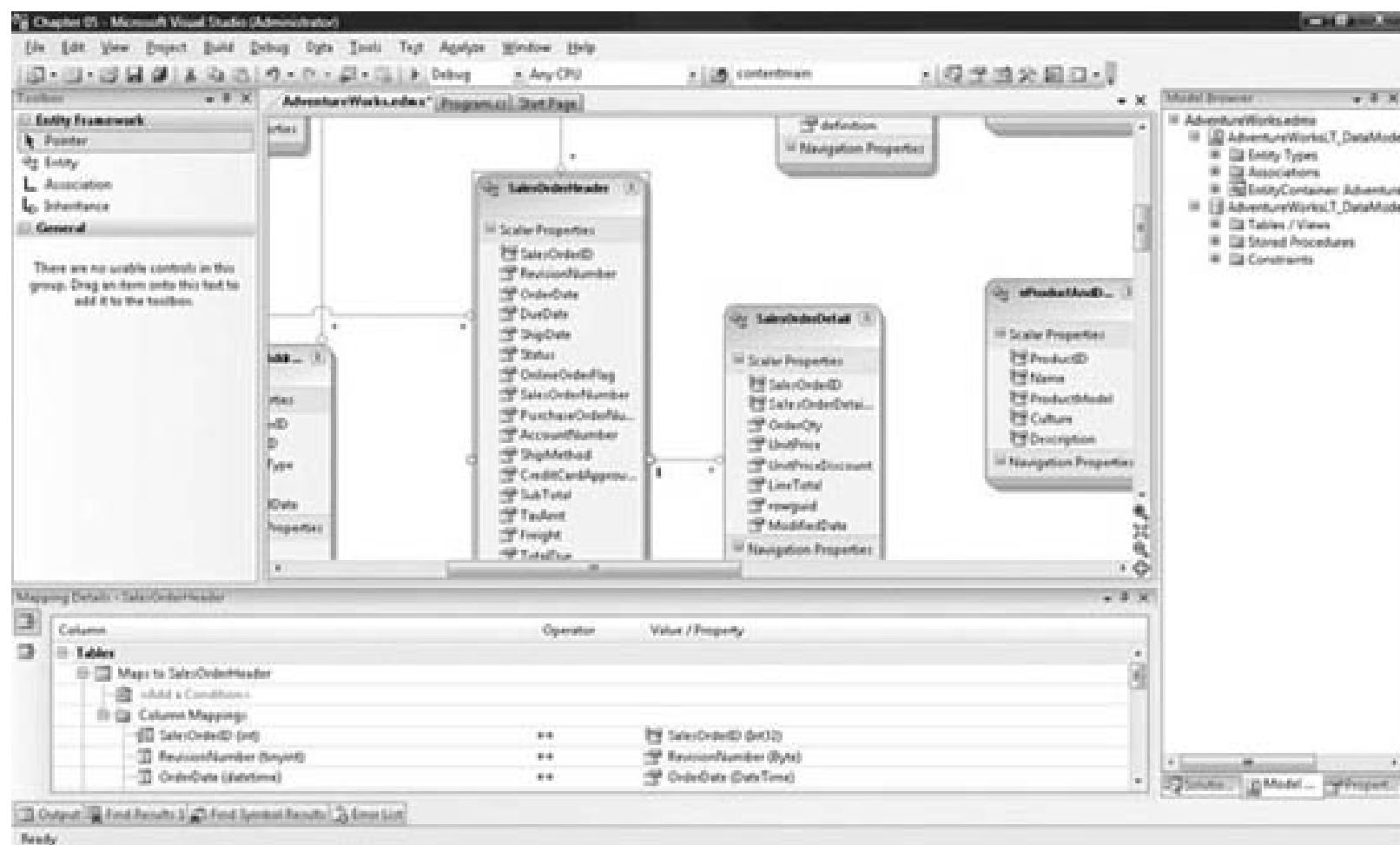


FIGURE 5-6 An Entity Data Model in VS 2008

Inside the Entity Data Model (EDM)

Another item created by the EDM wizard is the AdventureWorks.edmx file, which you can find in Solution Explorer. An associated file, AdventureWorks.designer.cs, contains source code generated by the EDM wizard. To get the most out of the EDM, you'll need to know what's available for you to use. The following sections describe theObjectContext, which is the interface you use to access the EDM in code, and the EntityObject, which is the entity itself. Looking inside the AdventureWorks.designer.cs file, the next section describes theObjectContext.

Accessing the EDM with theObjectContext

TheObjectContext contains everything you need for creating an EDM and accessing entities. The following sections describe what anObjectContext is, the constructors you can use, the entity properties that let you access the entities, and methods allowing you to add new data.

Understanding theObjectContext

TheObjectContext is the primary object for accessing the EDM. It also provides services, such as object tracking, query support, database communication, and data manipulation. You can instantiate it with connection strings and access all of the information in your EDM through it. [Listing 5-1](#) describes the

ObjectContext declaration, through the derived class, *AdventureWorksLT_DataEntities*, that the EDM wizard creates.

Listing 5-1 EDM Class Declaration

```
public partial class AdventureWorksLT_DataEntities :
    global::System.Data.Objects.ObjectContext
{
    // members removed
}
```

The *AdventureWorksLT_DataEntities* class demonstrates the convention being used for naming the class used for your EDM. Running the wizard, I named the EDM *AdventureWorksLT.edmx*, which is where the class name came from. You can also see that the EDM base class is *ObjectContext*. To use the *ObjectContext*, you'll first need to instantiate its derived class, which is discussed next.

Instantiating an ObjectContext

The *ObjectContext* has three constructors, each exposing a unique level of granularity in how you can instantiate the *ObjectContext*. You can see how to instantiate the EDM, via the constructors in [Listing 5-2](#).

Listing 5-2 EDM Constructors

```
public AdventureWorksLT_DataEntities() :
    base("name=AdventureWorksLT_DataEntities",
        "AdventureWorksLT_DataEntities")
{
    this.OnContextCreated();
}

public AdventureWorksLT_DataEntities(string connectionString) :
    base(connectionString, "AdventureWorksLT_DataEntities")
{
    this.OnContextCreated();
}

public AdventureWorksLT_DataEntities(
    global::System.Data.EntityClient.EntityConnection connection) :
    base(connection, "AdventureWorksLT_DataEntities")
{
    this.OnContextCreated();
}

partial void OnContextCreated();
```

Each of the constructors in [Listing 5-2](#) offers a different way of instantiating the *ObjectContext*, specifying database connection information. You can pass a raw connection string to a constructor that takes a string. Another constructor accepts an *EntityConnection*, enabling you to specify database connection information at a more granular level. Many implementations will use the default constructor, which uses a connection string located in the configuration\connectionStrings section of the app.config file. The following code shows an example of how you could use each of these constructors.

```
var edmDefault = new AdventureWorksLT_DataEntities();

var connStr =
```

```
@“metadata=res://*/AdventureWorks.csd|
res://*/AdventureWorks.ssd|
res://*/AdventureWorks.msl;
provider=System.Data.SqlClient;
provider connection string=
'Data Source=.\sqlexpress;
AttachDbFilename=
“”C:\Program Files\Microsoft SQL Server\MSSQL.1
\MSSQL\Data\AdventureWorksLT_Data.mdf“””;
Initial Catalog=AdventureWorksLT_Data;
Integrated Security=True;
MultipleActiveResultSets=True””;
var edmConnStr = new AdventureWorksLT_DataEntities(connStr);

var entConn = new EntityConnection(connStr);
var edmConnEnt = new AdventureWorksLT_DataEntities(entConn);
```

I formatted the preceding connection string so it would fit in the book, but be careful about line breaks. The EDM doesn’t recognize line breaks in a connection string and won’t be able to connect to the database unless the entire connection string is on one line. Notice that the connection string contains CSDL, SSDL, and MSL files, which are automatically generated by the EDM. You must add the provider, *System.Data.SqlClient* in the preceding section, to the connection string also. I used the same connection string to instantiate the *EntityConnection*, but you could set *EntityConnection* properties just as well.

NOTE The Entity Framework was built to support multiple DBMS systems. Third parties such as Oracle and MySql are expected to offer providers.

The last line of [Listing 5-2](#) contains a partial method for *OnContextCreated*, which is called by each of the constructors. This is an extensibility point where you can build your own partial type of the *ObjectContext*, *AdventureWorksLT_DataEntities*, and implement an *OnContextCreated* partial method.

Accessing Entities via ObjectContext Properties

You can access EDM entities via properties on the *ObjectContext*. Each property is of type *ObjectQuery*, which helps to manage construction and execution of database queries. [Listing 5-3](#) shows the *Product* and *ProductCategory* properties of the EDM.

Listing 5-3 Entity Properties

```
[BrowsableAttribute(false)]
public ObjectQuery<Product> Product
{
    get
    {
        if ((this._Product == null))
        {
            this._Product =
                base.CreateQuery<Product>(
                    “[Product]”);
        }
        return this._Product;
    }
}
private ObjectQuery<Product> _Product;

[BrowsableAttribute(false)]
```

```

public ObjectQuery<ProductCategory> ProductCategory
{
    get
    {
        if ((this._ProductCategory == null))
        {
            this._ProductCategory =
                base.CreateQuery<ProductCategory>(
                    "[ProductCategory]");
        }
        return this._ProductCategory;
    }
}
private ObjectQuery<ProductCategory> _ProductCategory;

```

The *Product* and *ProductCategory* properties are of type *ObjectQuery<T>*. They are read-only and have private backing stores. Properties cache entity instances in their backing store, checking for *null*; They retrieve the entity instance on-demand, when the backing store is *null*. The call to *CreateQuery<T>* uses Entity SQL, which is a special SQL-like dialect for database queries that supports selection only. Entity SQL is out of our scope, but you’ll learn how to perform LINQ to Entities queries, which can do just as much in this chapter.

The *Browsable* attribute is normally used on control properties, to determine whether the control property should appear in the Properties window. A control’s properties will display in the Properties window by default and so will *ObjectContext* properties. However, the *ObjectQuery<T>* properties of the *ObjectContext* shouldn’t show in the Properties window, which is the reason each *ObjectQuery<T>* property is decorated with the *Browsable* attribute set to *false*—to override the default behavior.

Putting together what you learned in the previous section about instantiating an *ObjectContext* and the information in this section about accessing data through *ObjectContext* properties, the following snippet shows you how to make a LINQ to Entities query:

```

var edm = new AdventureWorksLT_DataEntities();

var products =
    from product in edm.Product
    select product.Name;

```

Here, you can see how to use the *ObjectContext* instance, *edm*, to access the *Product* entity. It is standard LINQ query syntax, which you learned in [Chapter 2](#) and have been using throughout this book. As you might expect, the following *foreach* loop will print the results to the console:

```

foreach (var product in products)
{
    Console.WriteLine(product);
}

```

The query itself doesn’t materialize (run on the database) until the *foreach* loop accesses the first record. For the same reason as in LINQ to SQL, LINQ to Entities has deferred execution, where the query won’t be sent to the server until some code tries to access the first value. This is a benefit that allows you to formulate dynamic queries, without the overhead of calling the database every time you modify the query. The result of deferred execution is better performance and scalability.

Adding New Data with ObjectContext Methods

Another set of *ObjectContext* members includes methods that allow you to add data. These methods have an *Add* prefix and exist for each entity type in the EDM. [Listing 5-4](#) shows the methods available for the *Product* and *ProductCategory* entities.

Listing 5-4 ObjectContext

```
public void AddToProduct (Product product)
{
    base.AddObject("Product", product);
}
public void AddToProductCategory(ProductCategory productCategory)
{
    base.AddObject("ProductCategory", productCategory);
}
```

Each of the *Add* methods in [Listing 5-4](#) calls the *AddObject* method in the base *ObjectContext*. To use them, you can instantiate the entity object and pass the entity object instance as a parameter to the *Add* method for that entity object. Here's an example of adding a new *Product*:

```
var product =
    new Product
    {
        ProductNumber = "Zune1",
        Name = "Zune",
        ListPrice = 250.00m,
        ModifiedDate = DateTime.Now,
        SellStartDate = DateTime.Now,
        rowguid = Guid.NewGuid()
    };

var edm = new AdventureWorksLT_DataEntities();

edm.AddToProduct(product);
edm.SaveChanges();
var newProductID = product.ProductID;
```

The preceding *Product* class is the *Product* entity that was generated when the EDM was created. After instantiating the *ObjectContext* and assigning it to *edm*, the code calls *AddToProduct* with the new product instance. When calling *SaveChanges*, the EDM saves the new product into the database. Until you call *SaveChanges*, the new entity resides only in memory.

The last line of the preceding code highlights an important feature of AEF; after calling the *SaveChanges* method, the EDM loads the auto-increment identity field of the entity with a new primary key. The *newProductID* will hold that key for subsequent logic.

In addition to the capability of saving a new object in the database, you'll also need to know how to update and delete existing database records, which is covered in the next section.

Updating Existing Data

In the previous section, you saw how I added a new record to the *Product* table by using the *AddToProduct* method of the *ObjectContext*. With updates, you don't need a special method; you perform an update by getting a reference to the entity that will change, modify entity properties, and save that entity's data back to the database. Here's a continuing example of how to modify the *Zune* product that

was added in the previous section:

```
var newProduct =  
    (from prod in edm.Product  
     where prod.ProductID == newProductID  
     select prod)  
    .First();  
  
newProduct.ListPrice = 225.00m;  
  
edm.SaveChanges();
```

The preceding code explicitly extracts the new *Zune* product, filtering on the *newProductID* key that was used in the previous section. With a reference, you can modify an entity, and the EDM will keep track of the fact that the entity has been modified, which occurs when modifying the *ListPrice* of *newProduct* preceding. Calling *SaveChanges* ensures that the update will be sent to the database. The changes will only exist in memory until you call *SaveChanges*.

Now that you have a new object in the database that has also been modified, the next section will show you how to remove that object so it doesn't stay in your database.

Deleting Database Data

When deleting a database record, you need to obtain a reference to an entity that contains the data from the database record. In the previous section, you saw how we used a key value, *newProductID*, to extract a record from the database into an entity object. The following code shows how to delete a database record with a reference to an entity object:

```
edm.DeleteObject(product);  
edm.SaveChanges();
```

As shown in the preceding code, you can call the EDM *DeleteObject* method to delete data from a database. The parameter to *DeleteObject* is an entity with the same key as the database record. Remember to call *SaveChanges*. Otherwise, the object will only be deleted in memory, but not from the database.

Examining EDM Entity Types

Besides the *ObjectContext* for managing the EDM, an EDM contains entity types (classes) for holding entity instances. Knowing how entities work opens the possibility for you to write code that extends entities, if you should ever have the need. In particular, entities have properties, and those properties contain logic with events and partial methods that you can hook up to for additional functionality.

Defining an Individual Entity

In this section, I'll show you the *Product* entity and how it's defined. This example is typical of all entities you'll work with; an entity contains properties, events, and partial methods. [Listing 5-5](#) shows the *Product* entity. [Listing 5-5](#) doesn't show the entire class, which I'll explain soon.

Listing 5-5 An Entity Class Declaration

```
[EdmEntityTypeAttribute(  
    NamespaceName="AdventureWorksLT_DataModel",  
    Name="Product")]
```

```

[DataContractAttribute(IsReference=true)]
[Serializable()]
public partial class Product : EntityObject
{
    public static Product CreateProduct(
        int productID, string name, string productNumber,
        decimal standardCost, decimal listPrice,
        DateTime sellStartDate, Guid rowguid, DateTime modifiedDate)
    {
        Product product = new Product();
        product.ProductID = productID;
        product.Name = name;
        product.ProductNumber = productNumber;
        product.StandardCost = standardCost;
        product.ListPrice = listPrice;
        product.SellStartDate = sellStartDate;
        product.rowguid = rowguid;
        product.ModifiedDate = modifiedDate;
        return product;
    }
}

```

I formatted the code in [Listing 5-5](#) to make it more readable. For example, *EdmEntityTypeAttribute* and *EntityObject* are in the *System.Data.Objects.DataClasses* namespace, and *DataContractAttribute* is in the *System.Runtime.Serialization* namespace.

Each of the attributes on the *Product* class, or on any entity, specifies information about the class in the CSDL of the EDM. The *EdmEntityTypeAttribute* maps the class to an entity in the EDM. In [Listing 5-5](#), you can see that *Product* maps to the *Product* entity in the *AdventureWorksLT_DataModel* entity namespace.

You might be familiar with the *DataContract* attribute if you've worked with Web Services lately. It is the same *DataContract* attribute used in WCF Web Services, and it allows you to pass an entity as an argument to or to return value from a WCF Web Service. This is an instrumental capability in implementing ADO.NET Data Services, a new technology from Microsoft that allows you to make queries across the Internet (aka the cloud), via a Representational State Transfer (REST) interface. ADO.NET Data Services can use AEF as its underlying data store. Therefore, decorating entity classes with *DataContract* is essential to enabling this service where the entity can be serialized and transmitted across the Internet.

Entity classes include the *Serializable* attribute for any other serialization scenarios you might need to accommodate, such as .NET Remoting, ASP.NET Session state (non-InProc), or any other code that relies on a type being serializable.

Entity classes contain properties that manage access to data. These properties access private fields (backing stores), which hold the data. [Listing 5-6](#) shows two properties of the *Product* entity: *ProductID* and *Name*. The *ProductID* property demonstrates a property that maps to a primary key, and the *Name* property demonstrates a non-key field.

Listing 5-6 Entity Class Properties

```

[EdmScalarPropertyAttribute(
    EntityKeyProperty=true, IsNullable=false)]
[DataMemberAttribute()]

```

```

public int ProductID
{
    get
    {
        return this._ProductID;
    }
    set
    {
        this.OnProductIDChanging(value);
        this.ReportPropertyChanging("ProductID");
        this._ProductID =
            StructuralObject.SetValidValue(value);
        this.ReportPropertyChanged("ProductID");
        this.OnProductIDChanged(); }
}

private int _ProductID;
partial void OnProductIDChanging(int value);
partial void OnProductIDChanged();

[EdmScalarPropertyAttribute(IsNullable=false)]
[DataMemberAttribute()]
public string Name
{
    get
    {
        return this._Name;
    }
    set
    {
        this.OnNameChanging(value);
        this.ReportPropertyChanging("Name");
        this._Name =
            StructuralObject.SetValidValue(value, false);
        this.ReportPropertyChanged("Name");
        this.OnNameChanged();
    }
}

private string _Name;
partial void OnNameChanging(string value);
partial void OnNameChanged();

```

As in previous listings, I customized [Listing 5-6](#) to make it more readable. The *EdmScalarPropertyAttribute* attribute is in the *System.Data.Objects.DataClasses* namespace, and *DataMemberAttribute* is in the *System.Runtime.Serialization* namespace.

The *EdmScalarPropertyAttribute* attribute maps properties that have a single type value. In [Listing 5-6](#), you can see that both *ProductID* and *Name* are type *int* and *string*, respectively, which is a single value. If a property were to return a type that was another class, it would have the *EdmComplexPropertyAttribute* attribute. The *EntityKeyProperty* argument, which is set to *true*, of the *EdmScalarPropertyAttribute* that decorates *ProductID* indicates that *ProductID* is the primary key for this entity. The *IsNullable* argument, which is set to *false*, of *EdmScalarPropertyAttribute* indicates that you can't set this property to *null*.

The *DataMemberAttribute* attribute is consistent with usage of the *DataObject* attribute for the *Product* class. When *DataObject* is used on the class, you can use *DataMember* on the properties of the class to indicate that they will be serialized along with the entire *Product* instance when it is serialized.

Please see the previous discussion of *DataObject* for a better understanding of the purpose of the *DataMember* attribute.

The backing stores hold the data for the entity, with *_ProductID* and *_Name* holding data for *ProductID* and *Name*, respectively. What's probably more beneficial to know is the partial methods. These partial methods follow the pattern where *OnX Changing* occurs before modifying *X*, and *OnX Changed* occurs after modifying *X*; for the *ProductID* and *Name* properties, *X* is *ProductID* and *Name*, respectively. The benefit is that if you wanted to extend the functionality of the *Product* entity at the property level, that is, the *Name* property, you could write a partial *Product* class and define partial methods for the *OnNameChanging* and/ or *OnNameChanged*.

The *ReportPropertyChanging* and *ReportPropertyChanged* are events belonging to the *ObjectContext*. They are part of the *ObjectContext* object tracking service, enabling the *ObjectContext* to keep track of all the changes you make in the EDM. Object tracking, through object tracking events and internal references to original and modified records in the EDM, is what enables calls to *SaveChanges* to know which objects to persist. The previous section described how to use *SaveChanges* to add, modify, and delete entities.

Top-Down EDM Design

In the previous part of this chapter, you created and worked with an EDM that was generated directly from the database. There wasn't any design involved because the instructions were to select an existing database, which produced an EDM that mirrored database objects exactly, which is the bottom-up approach. However, you don't have to design your entities that way. The following sections will discuss the top-down design approach and how to accomplish a top-down design with the ADO.NET Entity Framework. You'll then be able to use LINQ to Entities in your application to work with data.

What Is the Top-Down Design Approach?

Instead of using the bottom-up approach, you can perform top-down design of an EDM with the ADO.NET Entity Framework. Top-down is ideal when you want your conceptual model (corresponding to CSDL) to work according to an object-oriented model that you've already designed. For example, if you take an object-oriented design approach to building software, you could analyze requirements and begin modeling with a common modeling language, such as the Unified Modeling Language (UML). Though you would have to use a third-party tool to create your UML (Some integrate directly with VS 2008), you would have a design that is based on an object-oriented view of the application you want to build. To implement your application, you can use the EDM designer to create entities that match the objects in your design—your conceptual model.

It's possible that your object-oriented design could have occurred in isolation of the database design. That is, a software architect does the object-oriented design, and a database architect does the database design. Many other scenarios could cause the object model to differ from the data model. However, the database is designed from a relational perspective, as opposed to the object-oriented perspective of the UML. Another situation you might encounter is that you are building an application that uses an existing database, and you won't need to do any database design. In both situations, existing and new database, you have a relational model to work with. Your task then, is to use the ADO.NET Entity Framework to build an EDM that maps from your conceptual object-oriented model (design) to the storage-level relational design.

Setting Up an EDM Using Top-Down Design

1. Create a new Console project (we'll keep it simple).
2. Right-click on the project, select Add | New Item, and select ADO.NET Entity Model. Name the item **AdventureWorks.edmx** and click the Add button. You'll see the Entity Data Model Wizard, shown in [Figure 5-7](#).
3. As shown in [Figure 5-7](#), you should choose the Empty Model option. In the top-down approach, you don't want to base your EDM on the relational schema. Instead, you want to build the entities yourself. After selecting Empty Model, click the Finish button. This creates an EDM in your project, the results of which are shown in [Figure 5-8](#).

Now, you have an EDM in your project that looks like [Figure 5-6](#), but with a couple differences. You need to create entities that mirror your object-oriented design, set up a database connection, and map (MSL) your entities (CSDL) to database objects (SSDL). Fortunately, you have the VS 2008 Toolbox and designer to create CSDL and the Mapping Details window to help create MSL. VS 2008 will derive the SSDL from the database connection you designate in Server Explorer. The following sections show you how to create the CSDL, MSL, and SSDL visually with VS 2008, starting with CSDL and creating entities with the visual designer.

Creating New Entities Yourself

If you already have objects to work with, you might have a desire to dig into the technical documentation, extracting attributes, and mapping the classes and their properties to the



Choose Model Contents

What should the model contain?



Generate
from d...



Empty model

Creates an empty model as a starting point to visually design a conceptual model from the toolbox. Classes are generated from the model when the project is compiled. You can specify a database connection later to map the conceptual model to the storage model.

< Previous

Next >

Finish

Cancel

FIGURE 5-7 Entity Data Model Wizard: Choose Model Contents with Empty Model

database manually. However, if you're building a new application, using a top-down design approach that allows your design to flow directly from customer requirements, you would want to create your own custom entities. The following sections describe what entities should be and how to create them.

An Example Object to Model

Since I only have a part of one chapter to describe how this works, I'm going to truncate the object-oriented analysis and design process and pretend that it has already been done. We can also imagine an object model that an architect has designed to describe how this system should be built. As your first task,

you get a single object from the design to implement. Many more objects are in the design, but for explanation purposes, we'll work with only one: the *Product* object, which is shown in UML in [Figure 5-9](#).

Remember that the object in [Figure 5-9](#) represents the results of the object-oriented design process, which creates objects from the perspective of the domain that the application is being built for, rather than how the data should be stored. From that

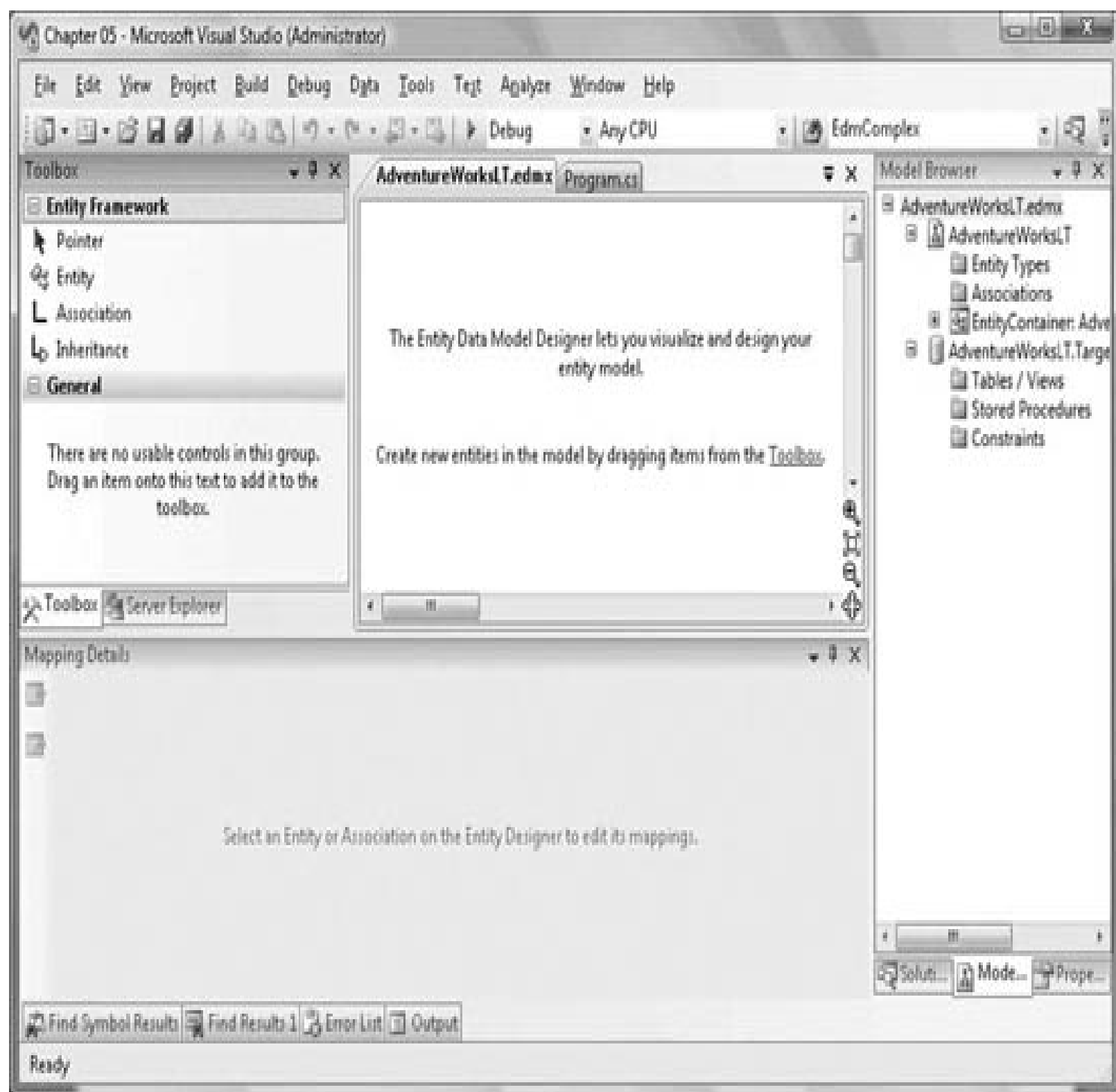


FIGURE 5-8 EDM Designer and tools in VS 2008

perspective, it could be reasonable to assume that property names might not be the same as column names

or that the relationships of data might not be exactly the same. For example, the *Product* table has columns named *ProductNumber* and *ListPrice*, but the *Product* class in [Figure 5-9](#) has *Number* and *Price* instead. Furthermore, the *Product* class has a *Category* property, but the *Product* table has a foreign key reference to the *ProductCategory* table that has a name. While this might not be the perfect scenario for all occasions, it does demonstrate that your object model will often differ from your relational model because of the way you want your code to work and how the design evolved, based on requirements. Now that you have an object to model, you can create an entity for it.

FIGURE 5-9 UML diagram of a Product class



Creating an Entity to Represent an Object

The next step is to add an entity to the EDM that represents the object in our design. It's now time to use the Toolbox, which contains three object types: entity, association, and inheritance. The *entity* is a class that contains data that we will work with, *Associations* map relationships between entities, and *inheritance* lets you derive one entity from another. The following steps describe how to create an entity in the VS 2008 designer.

1. Drag an *entity* object from the Toolbox onto the entity design surface.
2. Double-click on the header, and change the name from *Entity1* to **Product**. Alternatively, you could select the entity, open the Properties window, and change the *Name* property. The Properties window also gives you a way to add documentation and to set other class properties. Change the *Entity Set Name* property to *Product* through the Properties window.
3. When you add a new entity to the designer, it comes with an *ID* property by default. You should change the name of this property by selecting *ID* and changing the *Name* property to *ProductID* in the Properties window.

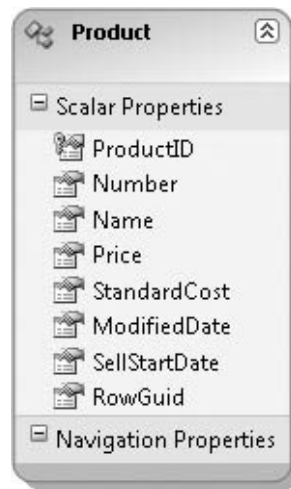
NOTEYou'll see part of the entity titled "Navigation Properties," which is where you can configure associations between entities. I'll discuss associations in a later section.

4. Next, right-click on the entity and select Add | Scalar Property. Make sure that either the whole entity or the Scalar Properties title is selected; otherwise you won't see the Add context menu item. The designer will add a new property, which is in edit mode. Type **Number** for the property name. Then open the Properties window and change the type to **String** (capitalized).
5. Just as with the *Number* property in Step 4, add the following properties with the specified Name/Type:
 - Name/String
 - Price/Decimal
 - StandardCost/Decimal
 - ModifiedDate/DateTime
 - SellStartDate/DateTime
 - RowGuid/Guid
6. Select the *Name* property, open the Properties window, and set *Nullable* to *false*. Set the *Nullable* property of each of the *Product* entity properties to *false*, just like the *Name* property. This highlights one of the important issues of building custom entities with an existing database—the properties you add must conform to the constraints of the underlying database schema they will map to. You won't see any errors while building the entity, but as soon as you perform mapping, where VS 2008 will

know what the constraints are, you'll begin receiving errors when constraints, such as type and nullability, don't match.

After completing the preceding steps, you'll have an entity that looks like [Figure 5-10](#).

FIGURE 5-10 A custom entity



Once you've completed the custom entity, you might want to examine the code in `AdventureWorksLT.designer.cs`. It looks much like the code that you saw for the EDM and entity classes earlier in this chapter. It is very important to remember that you should work through the Properties window to make any changes to this code, rather than directly coding. The code in `AdventureWorksLT.designer.cs` (or in any `X.designer.cs` file) is auto-generated, and you stand a great chance of losing your work if you make manual changes.

Creating Associations

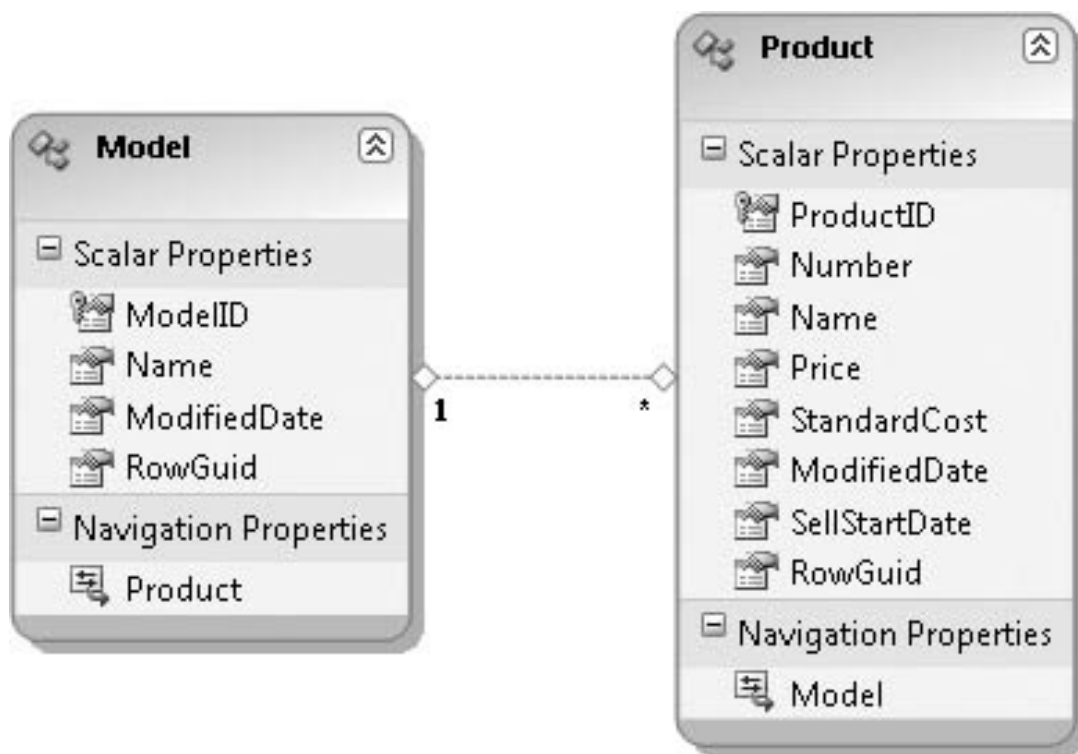
To create an association, you'll need another entity with a relationship to *Product*. Going back to our imaginary design from the previous section, another object called *Model* is a *Model* for the *Product*. The relationship is that *Product* has a *Model*. The following steps describe how to add the *Model* and define the association between *Product* and *Model*:

NOTE While creating relationships, you'll notice that VS 2008 begins showing you errors. This is because you haven't defined a database connection and object mapping yet. That's okay for now; I'll show you in an upcoming section how to create the connection and remove the errors you'll be seeing.

1. Add an entity to the designer, name it **Model**, change its *ID* property name to **ModelID**, and add a property named **Name** of type **String**. Change the Model entity's *Entity Set Name* property to **Model** through the Properties window. Then add a *ModifiedDate* property of type **DateTime** and a *RowGuid* property of type **Guid**. Set the *Nullable* property for all *Model* entity properties to *false*. If you don't recall how to build an entity, please refer to the previous section, which has steps to create the *Product* entity.
2. Add an association by clicking on the Association object in Toolbox, click on *Model*, and then click on *Product*. This will draw an association line between *Model* and *Product* as shown in [Figure 5-11](#).

The association in [Figure 5-11](#) shows that there is a one-to-many relationship between *Model* and *Product*, marked with a 1 on the *Model* and * on the *Product*. Additionally, notice that the *Navigation Properties* section of each entity now has an entry that reflects the association with the other entity. That is, the navigation property for *Model* is named *Product*

FIGURE 5-11 Association between entities



and the navigation property for *Product* is named *Model*. You can click on the association line in the designer or on either of the navigation property items and configure the association via the Properties window. The Properties window allows you to configure the multiplicity of each end of the association, name, documentation, and other object properties that map to code.

Besides associations, another type of relationship that AEF supports is inheritance, which is covered in the next section.

Implementing Inheritance

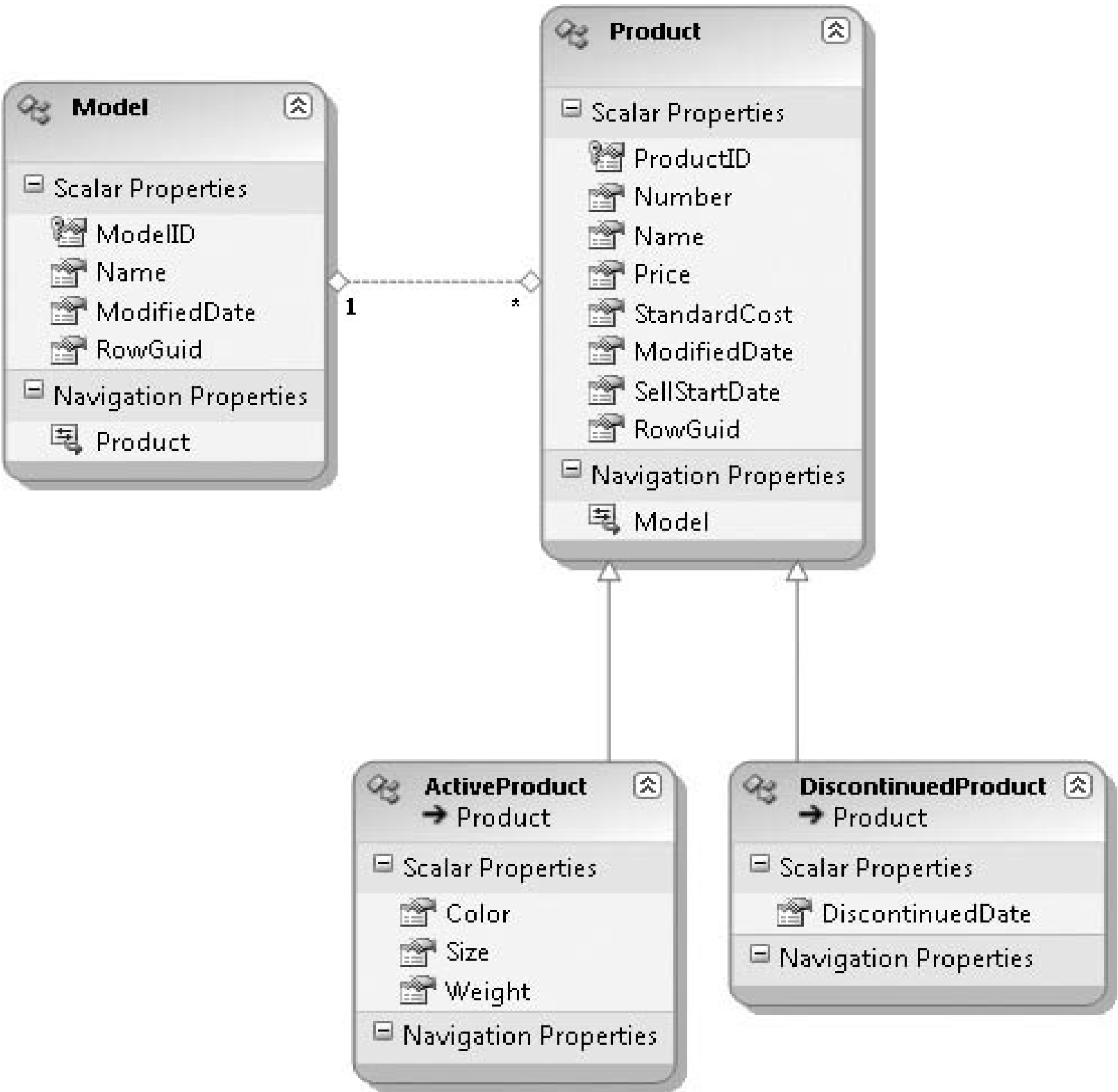
The AEF allows you to define inheritance relationships between entities. When you define an entity and set it to derive from a base class, it will inherit properties of the base class. This section will show you how to implement inheritance in AEF by adding two entities, named *ActiveProduct* and *DiscontinuedProduct*, as derived classes to the *Product* class. Here are the steps to accomplish this:

1. Set the *Abstract* property to *true* for the *Product* class. This adds the abstract modifier to the generated C# code, meaning that you can't create an instance of *Product*. Since derived class constraints will be set so that all *Product* instances can be categorized as a derived class, there isn't a need for instantiating *Product*. This decision is a part of the particular design example in this chapter, so I'll show you how the *Abstract* property works, and you have the option of whether to make your own base class abstract or not, depending on your own design.
2. Add an entity named **ActiveProduct**, giving it the following properties with Name/ Type settings:
 - Color/String
 - Size/String
 - Weight/Decimal
3. Add an entity named **DiscontinuedProduct**, giving it a property named **DiscontinuedDate** of type *DateTime*.
4. Set the *Nullable* property to *false* for each of the properties in both the *ActiveProduct* and *DiscontinuedProduct* entities.
5. Delete the *ID* properties that were added to the *ActiveProduct* and *DiscontinuedProduct* by default. Since *ActiveProduct* and *DiscontinuedProduct* derive from *Product*, they will inherit *ProductID* from *Product*.
6. Click on the *Inheritance* object in Toolbox, click on *ActiveProduct*, and then click on *Product*. This has created an inheritance relationship where *ActiveProduct* derives from *Product*.
7. Click on the *Inheritance* object in Toolbox, click on *DiscontinuedProduct*, and then click on *Product*. This creates an inheritance relationship where *DiscontinuedProduct* derives from *Product*. [Figure 5-12](#) shows the resulting EDM.

In [Figure 5-12](#), you can see the arrows going from *ActiveProduct* and *DiscontinuedProduct* to *Product*, showing that *ActiveProduct* and *DiscontinuedProduct* derive from *Product*. In the next section, on mapping, you'll learn how to define constraints to specify which records are either *ActiveProduct* or *DiscontinuedProduct*.

In addition to creating entities and defining associations between entities, you have now successfully implemented inheritance in your EDM. There are still errors in the IDE because the MSL isn't yet defined to map the CSDL to the SSDL. The next section describes how to accomplish that.

FIGURE 5-12 Inheritance in an EDM



Mapping Entities to the Database

As you've seen in the previous section, VS 2008 generates errors when you don't have mappings between entities and a database. In this section I'll show you how to create these mappings, but first you must learn how to create the database connection for the database that represents the SSDL you must map to.

Defining a Database Connection for an EDM

So far, we've designed the CSDL portion of the EDM by defining entities, but still need to define the SSDL so we can write an MSL to map CSDL to SSDL. To create the SSDL, you can use the Update Wizard, which also creates the SSDL if you haven't defined it yet. The following steps describe how to use the Update Wizard:

1. Open the visual designer (AdventureWorks.edmx in this chapter's examples), right-click on the design surface (not entities, associations, or inheritance objects), and select Update Model From Database. [Figure 5-13](#) shows the Update Wizard.



Choose Your Data Connection

Which data connection should your application use to connect to the database?

AdventureWorksLT_Data.mdf

New Connection...

This connection string appears to contain sensitive data (for example, a password), which is required to connect to the database. However, storing sensitive data in the connection string can be a security risk. Do you want to include this sensitive data in the connection string?

☐ No, exclude sensitive data from the connection string. I will set it in my application code.

☐ Yes, include the sensitive data in the connection string.

Entity connection string:

```
metadata=res://*/AdventureWorksLT.csdl|res://*/AdventureWorksLT.ssdl|res://*/AdventureWorksLT.msl  
;provider=System.Data.SqlClient;provider connection string='Data  
Source=.\sqlexpress;AttachDbFilename="C:\Program Files\Microsoft SQL Server\MSSQL.1  
MSSQL\Data\AdventureWorksLT_Data.mdf";Initial Catalog=AdventureWorksLT_Data;Integrated
```

☒ Save entity connection settings in App.Config as:

AdventureWorksLTContainer

< Previous

Next >

Finish

Cancel

FIGURE 5-13 The Update Wizard

2. [Figure 5-13](#) should be familiar if you've read the first part of this chapter because it is the same as the connection page in the EDM wizard. Once you've configured your database connection, click Next. You'll see the Choose Your Database Objects window, shown in [Figure 5-14](#).
3. Instead of grabbing the entire database, [Figure 5-14](#) shows how we're taking only the *Product*, *ProductCategory*, and *ProductModel* tables. This demonstrates that you have the ability to work with parts of the database, rather than all of it. In addition to Add, there are tabs on the page in [Figure 5-14](#) for Refresh and Delete. If the database has changed, you can click on the Refresh tab to update the SSDL for specified objects that are already in your SSDL. You can also click on the Delete tab and remove SSDL objects that are no longer part of the database schema. Schema items that are removed also remove MSL mappings for those items only. The next section discusses MSL mapping. Click the Finish button to complete the update.
4. After you run the Update Wizard, the wizard will add three new entries to your design surface. Delete these three new entities,

because you want to use the entity model you designed and not the ones generated from the database table.

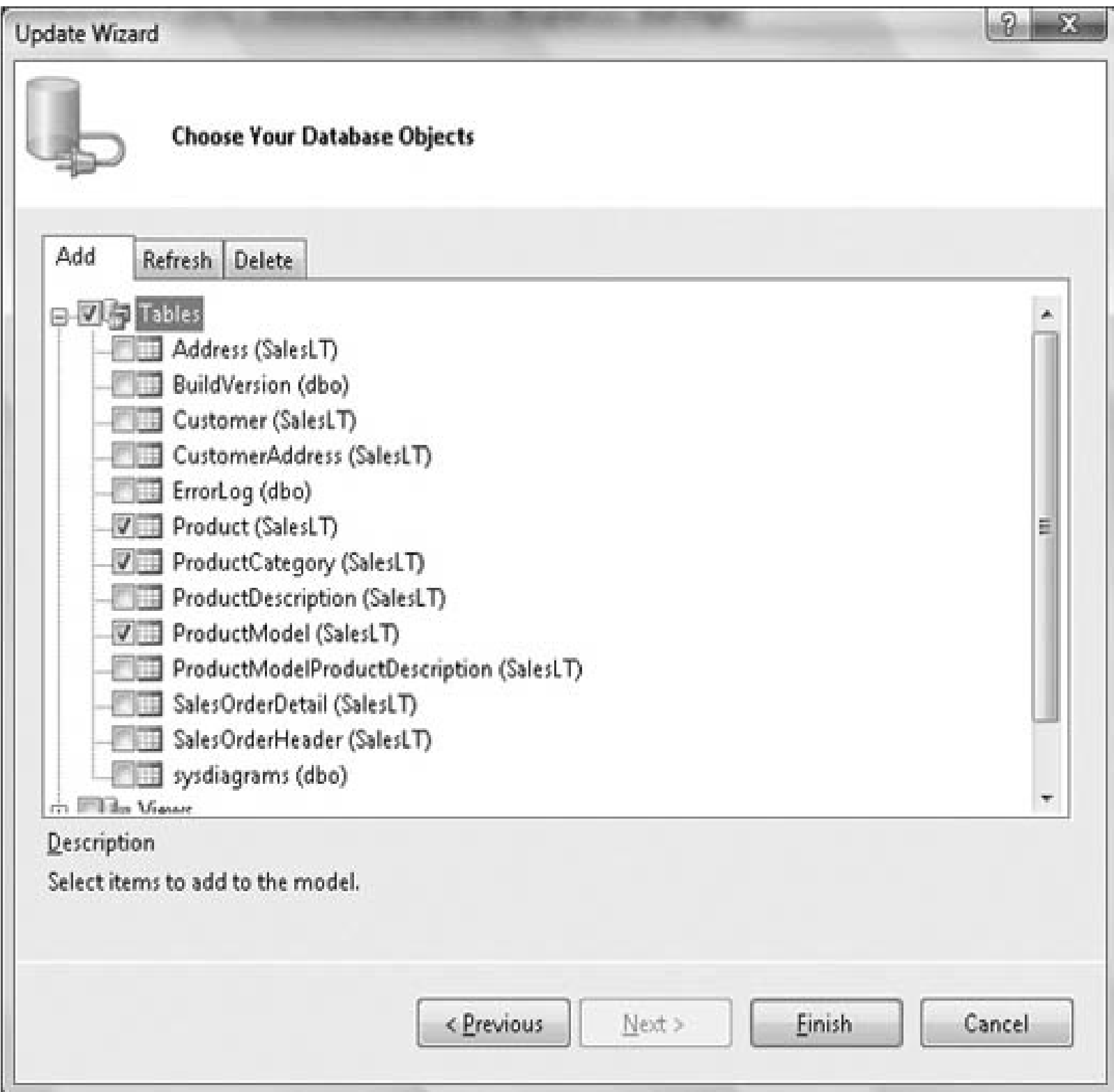


FIGURE 5-14 Choosing EDM data objects

You now have a CSDL, which is the entities you see in the designer, and an SSDL, which is created through the Update Wizard. The next section will show you how to create the MSL so you can map the CSDL to the SSDL.

Mapping EDM Entities to Database Tables

Just like the entity designer for CSDL and the Update Model from Database for SSDL, VS 2008 also gives you a visual tool for generating MSL, which is the Mapping Details window. This section will

show you how to use the Mapping Details window to map the CSDL to the SSDL. The following steps explain how to use the Mapping Details window:

1. Select the *Product* entity in the designer. When you select the *Product* entity, you'll see the Mapping Details window activate.
2. In the Mapping Details window, select Add A Table Or View. You'll see a drop-down list activate, and it will contain the *Product*, *ProductCategory*, and *ProductModel* tables from the SSDL that were defined by the Update Wizard in the previous section. Select *Product* and observe that Mapping Details defines all of the mappings it can between SSDL and CSDL as shown in [Figure 5-15](#).
3. [Figure 5-15](#) shows that properties such as *ProductID*, *Name*, and others mapped directly to the table because their names were exactly the same, eliminating a lot of manual work you must do to perform the mapping. However, some properties

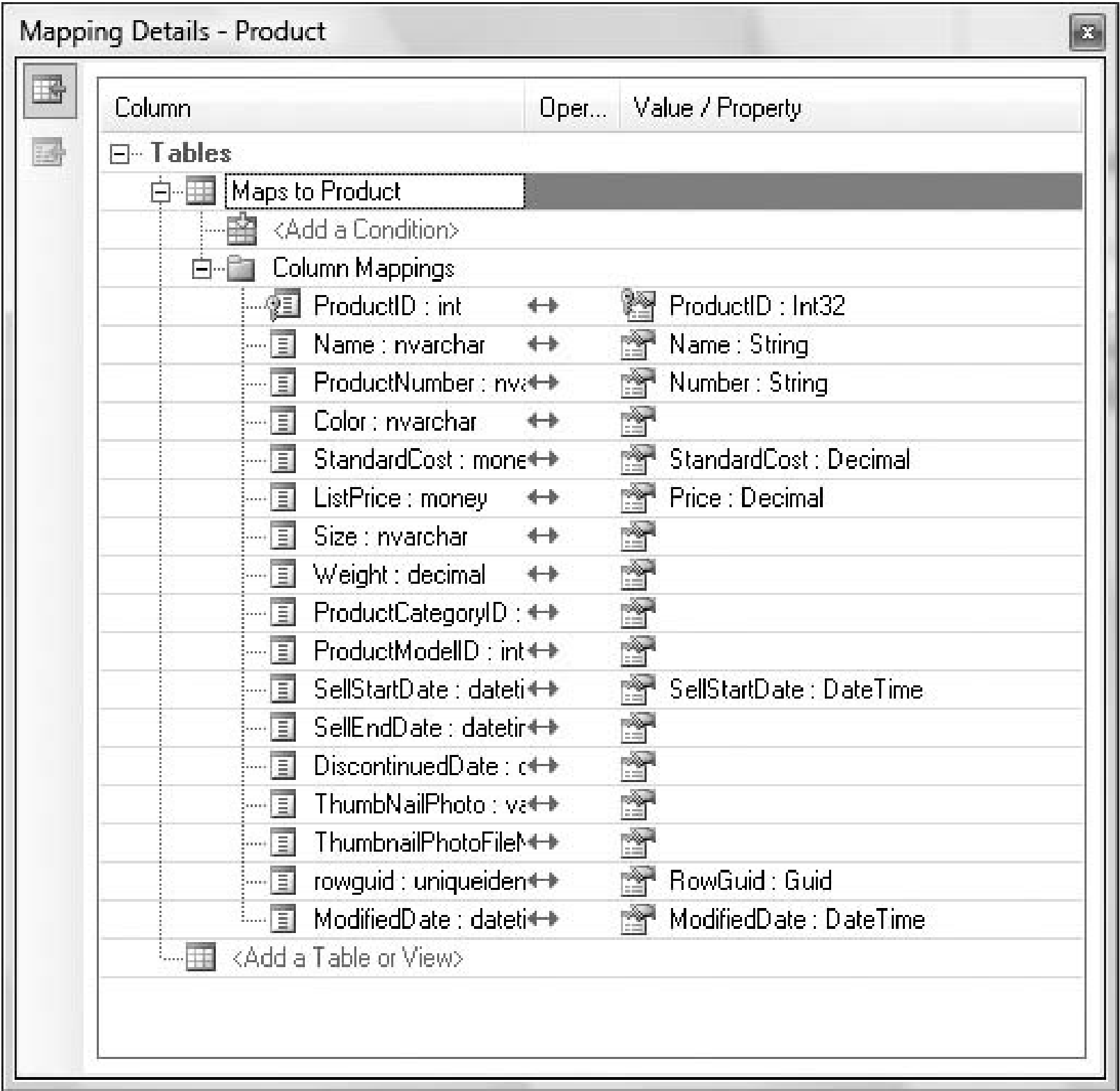


FIGURE5-15 Mapping Details for mapping entities (CSDL) to data (SSDL)

should map to the SSDL, but their names are different: *Number* and *Price*. To fix this, select *Number* from a pull-down list that displays in the *Value/Property* column for *Product Number*, and *Price* in the *Value/Property* column for *ListPrice*.

4. Next, you'll need to map the *Model* entity. This will be simpler than mapping the *Product* entity, because it is only a couple properties, and all properties map to the same table. To provide the *Model* entity mapping, select the *Model* entity in the designer, select Add A Table Or View in the Mapping Details window, and select *ProductModel*. The *Name* property will map to the *Name* column because they have the same identifier, but you'll need to manually map the *ModelID* property to the *ProductModelID* column. Now the *Model* entity is mapped, as shown in [Figure 5-16](#).
5. You'll also need to map the *Product* derived entities, *ActiveProduct* and *DiscontinuedProduct*. First, select *ActiveProduct* and set its mapping to the *Product* table in the Mapping Details window. The identifiers for all of the properties match column names in the *Product* table, so that completes mapping for *ActiveProduct*. Next, create a mapping between *DiscontinuedProduct* and *Product*. As with *ActiveProduct*, the *DiscontinuedDate* property identifier is the same as the *DiscontinuedDate* column in the *Product* table, so the *DiscontinuedProduct* entity is now mapped to the *Product* table.

FIGURE 5-16 Mapping for the *Model* entity

Mapping Details - Model

Column	Oper...	Value / Property
Tables		
Maps to ProductModel		
<Add a Condition>		
Column Mappings		
ProductModelID : int ↔		ModelID : Int32
Name : nvarchar ↔		Name : String
CatalogDescription : ↔		
rowguid : uniqueiden ↔		RowGuid : Guid
ModifiedDate : dateti ↔		ModifiedDate : DateTime
<Add a Table or View>		

- Next, you need to map the *ModelProduct* association. To do so, select the *ModelProduct* association, select Add A Table Or View, and select *Product*. This will give you a mapping grid for the *Model* and *Product* tables where you can define the keys for each table. The *ProductID* property is already selected because the identifiers match, but you'll need to select the *ProductModelID* column for the *ModelID* in the *Model* table. You can see the results of mapping the association between the *Model* and *Product* entities in [Figure 5-17](#).
- Finally, you need to map the inheritance relationships. To do so, select *ActiveProduct* in the designer or the Model Browser, click on Add A Condition in Mapping Details, select *DiscontinuedDate*, set Operator to *Is*, and set Value/Property to *Null*. Now, *ActiveProduct* entities will only hold *Products* where the *DiscontinuedDate* is set to *Null*, meaning that they are active, as opposed to discontinued, which is the next step.
- Map inheritance for *DiscontinuedProduct* by selecting the *DiscontinuedProduct* entity, click on Add A Condition in Mapping Details, select *DiscontinuedDate*, set Operator to *Is*, and set Value/Property to *Not Null*. When the *DiscontinuedDate* column has a value, it is not null, and now all *DiscontinuedProducts* will hold products that have been discontinued. Since the *DiscontinuedDate* is mapped when its value is not null, change its *Nullable* property to *false*.

When you build, you'll see an error message because an association was not added between *Product* and *ProductCategory*. This is not a compiler error, but appears because *ProductCategory* wasn't added to our EDM, and there is a foreign key relationship between *Product* and *ProductCategory*, which the EDM knows about. The code will still run.

You now have a custom EDM with entities (CSDL) mapped (MSL) to database tables (SSDL). Build the project to ensure there is a good compile and that all errors from previous sections are resolved. You can now use LINQ to Entities to query the EDM.

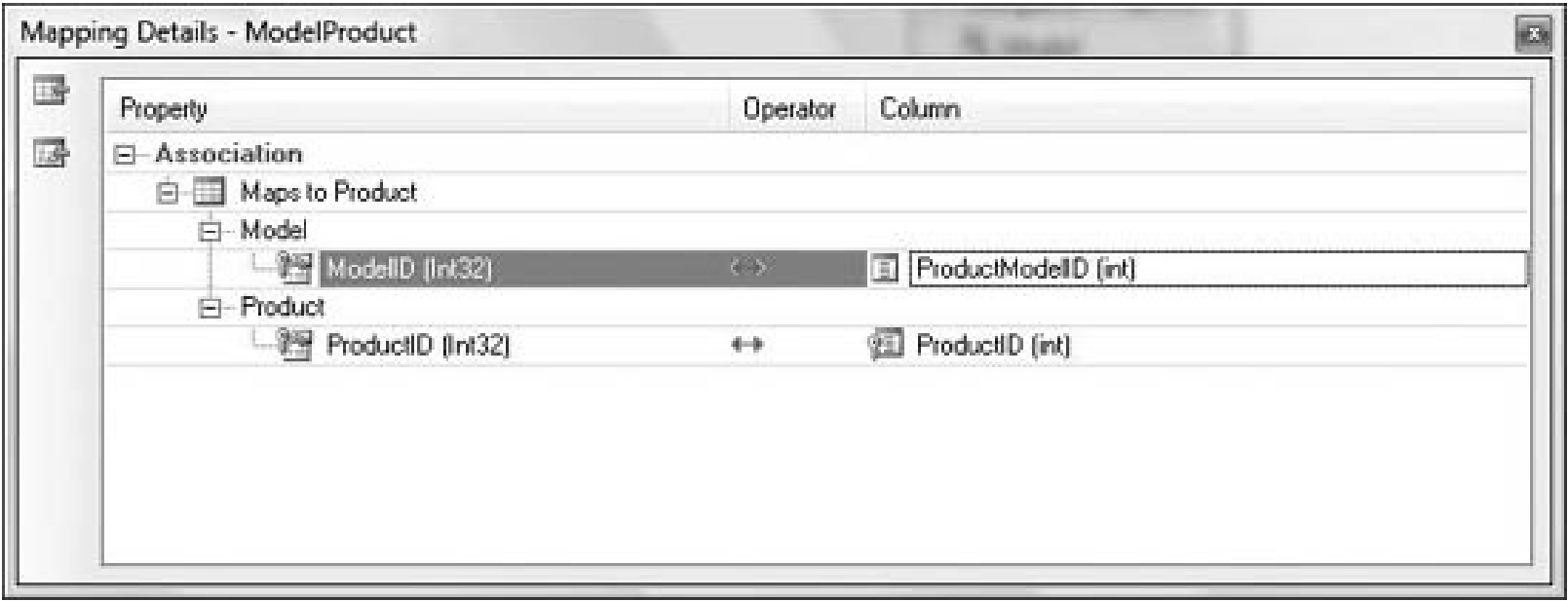


FIGURE 5-17 Association mapping

Querying the EDM with LINQ to Entities

After you've set up an EDM, you can query it with LINQ to Entities. Fortunately, LINQ query syntax is ubiquitous and applies to all providers, including LINQ to Entities.

If you recall from the previous section, there is an inheritance relationship where *ActiveProduct* and *DiscontinuedProduct* derive from *Product*, shown in [Figure 5-12](#). You can't query *Product* because it is abstract (its *Abstract* property is set to *true*). Therefore, you can only query through derived types. The following example shows you how to use the *OfType* operator to query derived types in the EDM:

```
var edm = new AdventureWorksContainer();

var products =
    from product in edm.Product.OfType<ActiveProduct>()
    select product;
```

AdventureWorksContainer is the *ObjectContext* in the current example. The EDM Wizard adds the *Container* suffix to the name you give it for creating the *ObjectContext* derived type identifier. The query itself is much like any other LINQ query, except that it uses the *OfType<T>* operator on the *edm.Product* data source. The type, *T*, must be one of the *Product* derived types, either *ActiveProduct* or *DiscontinuedProduct*. When the query executes, it uses the constraints specified in the Mapping Details window for the entity. That means that the code preceding will extract all *Product* records where the *DiscontinuedDate* is *null*.

To test *DiscontinuedProduct* types, change the *DiscontinuedDate* on one of the records in the *Product*

table from null to a valid date and run the following code:

```
var discontinuedProducts =  
    from product in edm.Product.OfType<DiscontinuedProduct>()  
    select product;
```

This time the *OfType* type is set to *DiscontinuedProduct*, which will only return records where the *DiscontinuedDate* is not null—as specified in the mapping.

You now have a custom EDM, representing an object-oriented design that you can query with LINQ syntax, via LINQ to Entities.

Summary

The ADO.NET Entity Framework (AEF) gives you an Entity Data Model (EDM) that you can query with LINQ to Entities. You can use familiar LINQ query syntax, just as you do with any other LINQ provider.

LINQ to Entities is configurable through VS 2008, and you can run wizards to set up a new EDM within minutes. You saw how to create instances of the EDM and then learned how to add, update, and delete entities.

After creating a bottom-up EDM, you learned how to create a top-down EDM, where you built custom entities that mirrored an object-oriented design, rather than the relational model of the database. You saw how you could define custom entities, build associations between entities, and create inheritance hierarchies to allow one entity to derive from another. After setting up the EDM, you saw how easy it is to use LINQ to Entities to query the EDM.

In the previous chapter, you learned how to query SQL Server data with LINQ to SQL, and this chapter showed you how to use LINQ to Entities, which you can connect to other DBMS systems that offer LINQ to Entities providers. The next chapter moves you from relational data sources to learning how to work with hierarchical data via XML by using LINQ to XML.

CHAPTER 6

Programming Objects with LINQ to XML

XML is a form of data that permeates the entire .NET development experience. Whether you're using DataSets, communicating via Web Services, or working with configuration files, XML touches much of what you do. For a data format of such importance, it only makes sense that Microsoft built a LINQ provider named LINQ to XML.

LINQ to XML allows you to read, manipulate, and create XML documents with LINQ syntax and special operators. LINQ to XML also includes a special object library for representing XML artifacts. You can use LINQ to XML to build XML in-memory via functional construction APIs that are intuitive and easy to use. You'll learn about all of these capabilities in this chapter, enabling you to use LINQ to XML for a large part of your XML development needs.

The LINQ to XML Objects

To help you become more acquainted with LINQ to XML, this section provides an overview of the objects you'll be using with LINQ to XML. Even if you're familiar with the primary XML types in .NET in use before LINQ to XML, you'll find this information useful because of some of the differences in LINQ to XML. This section provides an overview of these objects, and the rest of the chapter shows you how to use them. LINQ to XML objects are displayed in [Figure 6-1](#).

Some of the items in [Figure 6-1](#)—attributes, elements, and namespaces—show the differences between the way older .NET XML APIs process XML and the way LINQ to XML processes XML. Attributes, represented by *XAttribute*, are not considered nodes, which is clear by the fact that *XAttribute* derives from *XObject*, rather than from *XNode*. Both *XElement* and *XDocument* can contain XML—they are at the same level in the object hierarchy, under *XContainer*. The point is that *XElement* can hold XML, independent of an



FIGURE 6-1 LINQ to XML object diagram

LINQ to XML Object Name	Used For
XAttribute	XML attribute
XCData	Free text/ <i>CDATA</i>
XComment	XML comment

XContainer	Base class for objects that hold other XML objects— XElement and XDocument
XDeclaration	Declaration at top of XML file
XDocument	Containing an entire XML document
XDocumentType	DTD declaration
XElement	Contains single element or entire element hierarchy independently (without XDocument)
XName	Attribute or element name
XNamespace	XML namespace
XNode	Base class for XComment, XContainer, XDocumentType, XProcessingInstruction, and XText
XObject	Base class for XNode and XAttribute
XProcessingInstruction	Processing instruction
XText	Text
XStreamingElement	Processing large documents with a small memory footprint

TABLE 6-1 LINQ to XML Object Descriptions

XDocument. Finally, namespaces are managed via *XNamespace* types, which is a simple way to manage namespaces. [Table 6-1](#) summarizes the purpose of each of these objects.

Each of the LINQ to XML objects is a member of the *System.Xml.Linq* namespace, which means that you’ll need to add a *using* declaration for *System.Xml.Linq* for each listing you see in this chapter. You’ll learn how these objects are used throughout this chapter, starting with XML document creation.

Creating XML Documents

An XML document originates from somewhere, whether you create it on-the-fly or retrieve it from another source. This section will discuss the various issues associated with the creation of XML, saving XML, and retrieval of XML from various sources. A very special feature of LINQ to XML is the ease with which you can construct an XML document via a technique referred to as functional construction, discussed next.

Functional Construction

LINQ to XML makes constructing XML documents quick and easy. It uses a technique called *functional*

construction, which means that the code to create the XML can be created in a readable way that represents the structure of the document. The benefit of functional construction is that you have a strongly typed way to construct an XML tree, along with a visual representation that matches the structure of the XML. [Listing 6-1](#) shows how to create XML for a set of products, much like the *Product* table from the AdventureWorksLT database.

Listing 6-1 Using Functional Construction to Create an XML Document

```
var products =  
    new XElement("products",  
        new XElement("product",  
            new XAttribute("productID", 1),  
            new XElement("name", "Zune"),  
            new XElement("listPrice", 250.00m)),  
        new XElement("product",  
            new XAttribute("productID", 2),  
            new XElement("name", "iPod"),  
            new XElement("listPrice", 299.00m)));
```

```
Console.WriteLine(products.ToString());
```

And here's the output:

```
<products>  
  <product productID="1">  
    <name>Zune</name>  
    <listPrice>250.00</listPrice>  
  </product>  
  <product productID="2">  
    <name>iPod</name>  
    <listPrice>299.00</listPrice>  
  </product>  
</products>
```

The first thing you might notice from [Listing 6-1](#) is that the creation of the XML document looks like the XML that is being built, where the hierarchical relationship between elements is visually evident. It also helps that indentation and spacing are possible, yielding a familiar appearance that makes the objects resemble the XML text being produced.

The example in [Listing 6-1](#) uses *XElement* for the elements and *XAttribute*, for element attributes. Being able to show *XElement*, *XAttribute*, and other LINQ to XML objects via functional construction is facilitated by constructor overloads. Notice that each *XElement* or *XAttribute* is instantiated with the *new* operator and that each instantiation has arguments that are other *XAttribute* and *XElement* objects. The constructor overloads take *params* parameters, allowing a variable number of arguments to be assigned. Combined with understandable formatting, functional construction makes creation of XML documents intuitive.

Saving XML Documents

You can save an XML document into a file by calling the *Save* method on the *XElement*. Here's an example that saves the document created in [Listing 6-1](#):

```
products.Save("Products.xml");
```


The preceding argument is a string with the filename. It could have also included the file path. Other overloads of the *Save* method accept *TextWriter* or *XmlWriter* types. Also, a couple of overloads take a *SaveOptions* enum parameter, which specifies whether you can add whitespace formatting or not. *SaveOptions* has two values, *DisableFormatting* and *None*. *DisableFormatting* prevents formatting whitespace, and *None*, the default if you don't specify *SaveOptions*, adds formatting whitespace. The following code demonstrates how to create XML from a string and to save without formatting:

```
var products = XElement.Parse(
    "<products> <product>SomeProduct</product> </products>");
products.Save("ProductsNoFormat.xml", SaveOptions.DisableFormatting);
```

As specified by the *DisableFormatting* option of *SaveOptions* the output occurs without any additional formatting whitespace. I added a line break for this book to fit the line comfortably on the page, but in the output file, all of the content is on one line. On the other hand, if you either use *SaveOptions.None* or an overload of the *Save* method that doesn't use *SaveOptions*, the XML file will include formatting whitespace. Here's an example of using *SaveOptions.None*:

```
var products = XElement.Parse(
    "<products> <product>SomeProduct</product> </products>");
products.Save("ProductsWithFormat.xml", SaveOptions.None);
```

In the preceding example, the *SaveOptions* argument is set to *None*. Here's the output from the *Save* method, with the preceding *SaveOptions.None*:

```
<?xml version="1.0" encoding="utf-8"?>
<products>
  <product>SomeProduct</product>
</products>
```

As specified by the *SaveOptions.None* argument, the result is formatted.

This section showed you how to create XML with functional construction. The next section will show you how to retrieve XML from various sources.

Retrieving XML from Various Sources

The source of the XML can be strings that are passed to a method, created as the result of calling another method, or the string can be built on-the-fly. Of course, there isn't much need to create XML strings on-the-fly because now you have functional construction, which is strongly typed and quick to build with tools like VS 2008 and IntelliSense. XML can also reside in a file or in a URI across a network. The following sections show you how to access XML, regardless of data source, with LINQ to XML.

Creating XML from Strings

You've already seen from the previous section how to create XML from a string. The *XElement* has a *Parse* method that will convert a string into an *XElement*, shown next:

```
var products = XElement.Parse(
    "<products> <product>SomeProduct</product> </products>");
```

Now, *products* is an *XElement* that contains XML from the string argument to the *Parse* method. In addition to strings, there are other ways to load XML, continuing next with streams.

Reading XML from Streams

The *Load* method of *XElement* has an overload that accepts a *System.IO.TextReader*. The following examples demonstrate a couple ways you could use this overload to retrieve XML data with *TextReader* derived types *StringReader* and *StreamReader*. The following example shows how to read raw string data, in addition to the techniques in the previous section, by using a *StringReader*:

```
var stringRdr = new StringReader(
    "<products> <product>SomeProduct</product> </products>");
var strgRdrElem = XElement.Load(stringRdr);
```

This example used a *StringReader* instantiated with a string of XML data. The overload of the *Load* method that takes a *TextReader* works because *StringReader* derives from *TextReader*.

Another *TextReader* derived type, *StreamReader*, lets you read data from a file. Here's an example that reads the *products.xml* file that was created in a previous section:

```
var streamRdr = new StreamReader("products.xml");
var strmRdrElem = XElement.Load(streamRdr);
```

Notice how the *StreamReader* instantiates with the name of the XML file to read. The *StreamReader* itself has additional constructor overloads, such as accepting a *FileStream*, that you could use also. Explaining *StreamReader* is out of scope, but I just want to give you an idea of the flexibility you have in being able to get to an XML source in multiple ways.

While using *StreamReader* is a possibility if you need it, the next section shows you a more direct route to reading XML from a file.

Reading XML from Files

The *Load* method of *XElement* has an overload that takes a string, which you can use to specify a file to read. The following example shows you how to use the *Load* overload that takes a string so you can specify the *Products.xml* file that was created in a previous section:

```
var file = XElement.Load("Products.xml");
```

All you need to do is specify the name of the file. You could also specify the path to the file. The following example is equivalent to the preceding example that uses a string for the filename, except that it uses a fully qualified path:

```
var file = XElement.Load(
    @"C:\LinqProgramming\Chapter 06\Chapter 06\bin\Debug\Products.xml");
```

Of course, you would probably pass *string* variables instead of raw strings like this, but it does demonstrate that you can access XML anywhere in your file system that your application has access to.

This particular overload of the *Load* method, with the *string* parameter, can interpret the string in different ways. The next section shows you how to pass a URI in the form of a string.

Reading XML from URIs

You can also use the *Load* method of *XElement* to retrieve XML from a network or Internet address. The following example reads the RSS feed from Charlie Calvert's Community Convergence Blog:

```
var rss = XElement.Load(  
    "http://blogs.msdn.com/charlie/rss.xml");
```

Just as with the preceding URL, you can specify any URI to read information across a network. Another source of information is *XmlDocument* data, discussed next.

Reading XML from XmlReaders

If you're working with legacy .NET XML APIs such as *XmlDocument*, you'll need a way to get to that data. The overload of the *Load* method of the *XElement* class that takes an *XmlReader* will help you read XML from an *XmlDocument*. Here's an example that sets up an *XmlDocument* instance, based on the *products.xml* file from previous sections, and then uses an *XmlNodeReader* to bring the data to LINQ to XML:

```
var doc = new XmlDocument();  
doc.Load("products.xml");  
  
var nodeRdr = new XmlNodeReader(doc);  
  
var xmlRdrElem = XElement.Load(nodeRdr);
```

The preceding example instantiates an *XmlDocument*, and that *XmlDocument* instance loads the *products.xml* file. We've used this technique to load the *XmlDocument*, but an *XmlDocument* can be coded to hold XML data in many different ways, which is out of scope. You should just be aware of this capability as another line of flexibility that you have in getting to XML data sources. The *XmlNodeReader*, instantiated with an instance of the *XmlDocument*, derives from *XmlReader*, meaning that you can pass the *XmlNodeReader* instance to the overload of the *Load* method that takes an *XmlReader*. Now you have access to the XML from the XML document in the LINQ to SQL *XElement* instance, *xmlRdrElem*.

In addition to the overloads of the *Load* method shown in this and previous sections, additional methods accept a *LoadOptions* enum, which has four members: *None*, *PreserveWhitespace*, *SetBaseUri*, and *SetLineInfo*. The *None*, option doesn't preserve insignificant whitespace but *PreserveWhitespace* does. *SetBaseUri* gets the base URI and makes it available via the *BaseUri* property. *SetLineInfo* allows you to associate elements with the lines they occur on.

The examples you've seen have been simple, but for real work, you'll need to use namespaces, which you'll learn about in the next section.

Working with LINQ to XML Namespaces

Namespaces are relatively easy to use in LINQ to XML, because you can build and use them just like strings. The following sections show you how to create and use a namespace and how to use namespace prefixes.

Creating and Using a Namespace

You set namespaces in LINQ to XML with *XNamespace*. The following example shows how to create a namespace and add it to an *XElement*,:

```
XNamespace lp = "http://www.mcgraw-hill.com/linqprogramming";  
  
var productsNS =
```

```

new XElement(lp + "products",
    new XElement("product",
        new XAttribute("productID", 1),
        new XElement("name", "Zune"),
        new XElement("listPrice", 250.00m)),
    new XElement("product",
        new XAttribute("productID", 2),
        new XElement("name", "iPod"),
        new XElement("listPrice", 299.00m)));

```

```
Console.WriteLine(productsNS.ToString());
```

The *XNamespace*, *lp*, is created by assigning a string, which is possible because *XNamespace* has an implicit conversion from string. To see how the namespace can be used, notice that *lp* is concatenated to *"products"*, resulting in the following output:

```

<products xmlns="http://www.mcgraw-hill.com/linqprogramming">
  <product productID="1" xmlns="">
    <name>Zune</name>
    <listPrice>250.00</listPrice>
  </product>
  <product productID="2" xmlns="">
    <name>iPod</name>
    <listPrice>299.00</listPrice>
  </product>
</products>

```

The preceding example shows that we've set the default namespace with the previous code. Frequently, you'll need to assign namespace prefixes instead, which is explained in the next section.

Creating Namespace Prefixes

LINQ to XML allows you to create namespace prefixes so you can assign namespaces to selected elements without all of the verbosity of repeating the namespace. Since a namespace declaration is essentially an attribute, you would do this with an *XAttribute* in LINQ to XML, shown in the following example:

```
XNamespace lp = "http://www.mcgraw-hill.com/linqprogramming";
```

```

var productsPfx =
    new XElement("products",
        new XAttribute(XNamespace.Xmlns + "lp", lp),
        new XElement(lp + "product",
            new XAttribute("productID", 1),
            new XElement("name", "Zune"),
            new XElement("listPrice", 250.00m)),
        new XElement(lp + "product",
            new XAttribute("productID", 2),
            new XElement("name", "iPod"),
            new XElement("listPrice", 299.00m)));

```

```
Console.WriteLine(productsPfx.ToString());
```

The important line to pay attention to is the *XAttribute* where *XNamespace.Xmlns* is concatenated to the *"lp"* string and associated with the *lp* namespace, defining the namespace prefix. Next, notice how I've concatenated *lp* to the first *product* element. Instead of adding the fully qualified namespace on the

first *product* element, LINQ to XML will recognize the *XAttribute* associating the prefix to this namespace and emit the prefix instead. Here's the output:

```
<products xmlns:lp="http://www.mcgraw-hill.com/linqprogramming">
  <lp:product productID="1">
    <name>Zune</name>
    <listPrice>250.00</listPrice>
  </lp:product>
  <product productID="2">
    <name>iPod</name>
    <listPrice>299.00</listPrice>
  </product>
</products>
```

You can see in the output that the first product, *lp:product*, has the *lp* prefix.

Working with Documents

Previous sections focused on using the *XElement*, which is a common way to work with either fragments or entire XML documents. However, you can still work with an entire XML document via the *XDocument* class. Here's an example:

```
var xDoc = new XDocument(
    new XElement("products",
        new XElement("product",
            new XAttribute("productID", 1),
            new XElement("name", "Zune"),
            new XElement("listPrice", 250.00m)),
        new XElement("product",
            new XAttribute("productID", 2),
            new XElement("name", "iPod"),
            new XElement("listPrice", 299.00m))));

Console.WriteLine(xDoc.ToString());
```

And here's the output:

```
<products>
  <product productID="1">
    <name>Zune</name>
    <listPrice>250.00</listPrice>
  </product>
  <product productID="2">
    <name>iPod</name>
    <listPrice>299.00</listPrice>
  </product>
</products>
```

As you can see, you can create an *XDocument* in the same way as an *XElement*. The *XDocument* class also has a static *Load* method that you can use, just like the *XElement Load* method.

Adding Declarations, Text, and Processing Instructions

Additional items you can add to XML with LINQ to XML are declarations, text, and processing instructions, which are covered in the following sections.

Including Declarations

If you don't add any declaration to an *XDocument*, you'll receive a default declaration. For example, you could code the following *XDocument*:

```
var xDocNormalDeclaration =  
    new XDocument(  
        new XElement("products",  
            new XElement("product",  
                new XAttribute("productID", 1),  
                new XElement("name", "Zune"),  
                new XElement("listPrice", 250.00m)),  
            new XElement("product",  
                new XAttribute("productID", 2),  
                new XElement("name", "iPod"),  
                new XElement("listPrice", 299.00m))));  
  
xDocNormalDeclaration.Save("normalDeclaration.xml");
```

The point in the preceding code is that no explicit statements specify what the XML declaration should be. When this code executes, the output will include the following declaration:

```
<?xml version="1.0" encoding="utf-8"?>
```

I left out the rest of the document because previous sections have already shown you what it should look like. You can change this declaration by adding an *XDeclaration* instance to the parameter list of the *XDocument*. Here's an example that customizes the declaration:

```
var xDocCustomDeclaration = new XDocument(  
    new XDeclaration("1.0", "UTF-16", "yes"),  
    new XElement("products",  
        new XElement("product",  
            new XAttribute("productID", 1),  
            new XElement("name", "Zune"),  
            new XElement("listPrice", 250.00m)),  
        new XElement("product",  
            new XAttribute("productID", 2),  
            new XElement("name", "iPod"),  
            new XElement("listPrice", 299.00m))));  
  
xDocCustomDeclaration.Save("customDeclaration.xml");
```

This time, the first parameter to the new *XDocument* constructor is an *XDeclaration* instance. Notice how the parameters customize the results. From left to right, the parameters are *version*, *encoding*, and *standalone*. Here's the declaration from the output:

```
<?xml version="1.0" encoding="utf-16" standalone="yes"?>
```

You can see that the output reflects the custom parameters specified in the *XDeclaration* instance, passed as the first parameter to the *XDocument* constructor in the previous example.

Another way to add content to XML is via text, which you'll learn about next.

Defining CDATA Text

Adding raw text via *CDATA* elements is as easy as adding an *XCDATA* instance to your XML tree. Here's an example of an *XElement* that adds a *CDATA* section for free-form customer comments on a product:

```
var cDataElem =
```

```
new XElement("product",
    new XCData("Customer comment..."));
```

```
Console.WriteLine(cDataElem.ToString());
```

And here's the output:

```
<product><![CDATA[Customer comment...]]></product>
```

As just shown, by adding the *XCData* instance, initialized with the text you want to appear, to the *XElement*, you can add a *CDATA* section to your XML.

Another item you can add to your XML is processing instructions, discussed next.

Handling Processing Instructions

If you have a tool that requires some type of special handling within an XML document that is unique to your application, you can add a processing instruction. The following example shows how to add a processing instruction to your XML:

```
var procInstElem =
    new XElement("product",
        new XProcessingInstruction("myTarget", "special-data"));
```

```
Console.WriteLine(procInstElem.ToString());
```

You can add the *XProcessingInstruction* anywhere in the XML tree that you need it to appear. The parameters, from left to right, are target, *myTarget*, and data, *special-data*. Here's the output:

```
<product>
  <?myTarget special-data?>
</product>
```

You can see the processing instruction in the XML preceding, which matches the *XProcessingInstruction* instance in the code preceding it.

While this and previous sections concentrated on creating XML, you'll often need to read XML from external sources that you don't have control over. The next section shows you how to validate XML.

Validating XML

When you get XML from external sources, you'll want to validate it to ensure it conforms to your schema. This section will validate the following XML document:

```
<?xml version="1.0" encoding="utf-8"?>
<products>
  <product productID="1">
    <name>Zune</name>
    <listPrice>250.00</listPrice>
  </product>
  <product productID="2">
    <name>iPod</name>
    <listPrice>299.00</listPrice>
  </product>
</products>
```

This is the *products.xml* document created by an earlier program in this chapter. It will be validated

with the following schema document, products.xsd:

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema attributeFormDefault="unqualified"
    elementFormDefault="qualified"
    xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="products">
<xs:complexType>
<xs:sequence>
<xs:element maxOccurs="unbounded"
    name="product">
<xs:complexType>
<xs:sequence>
<xs:element name="name"
    type="xs:string" />
<xs:element name="listPrice"
    type="xs:decimal" />
</xs:sequence>
<xs:attribute name="productID"
    type="xs:unsignedByte"
    use="required" />
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:schema>
```

In the following examples, be sure to place both of these documents in the same folder as the executable files that will use them.

VS 2008 makes it easy to generate a schema. Just open the XML file, select the XML menu, and then select Create Schema. Then VS 2008 will create a schema file for you to use as is or to modify as you need. The preceding products.xsd file was generated this way. The following code validates the products.xml XML document with the products.xsd schema document. Remember to add a *using* declaration for *System.Xml.Schema*:

```
var schemaSet = new XmlSchemaSet();
schemaSet.Add(string.Empty, "products.xsd");

var products = XDocument.Load("products.xml");
bool isValid = true;
products.Validate(
    schemaSet,
    (sender, e) =>
    {
        Console.WriteLine(
            "Error Occurred: " +
            e.Exception.ToString());

        isValid = false;
    });

if (isValid)
{
    Console.WriteLine("No Errors");
}
```


As shown in the preceding code, you must define the schema file via an *XmlSchemaSet* instance. The first parameter to the *Add* method is the *namespace*, and the second parameter is the *name* of the schema file. Then you use the static *Load* method of the *XDocument* to load the XML. The *IsValid* is an indicator to tell us if there were no errors during processing.

Calling *Validate* on the *XDocument*, *products*, executes the validation. The first parameter to *Validate* is the *schemaSet*, which holds the schema document used to validate the contents of the products *XDocument*. The second parameter, implemented via a *lambda*, is an error handler that the *Validate* method calls if any errors occur. Notice that the error handler sets *IsValid* to *false*, letting later statements in the algorithm know that an error occurred.

Queries and Axes

LINQ to XML queries use the same query syntax as with other LINQ providers. What's different in LINQ to XML is an entire array of extension methods for managing axes and navigation. The following sections show you how to perform queries and how to use the axes methods of LINQ to XML.

Basic LINQ to XML Queries

Queries in LINQ to XML use the same syntax as all of the other providers. This section will show you simple queries for elements and attributes. The following example shows how to perform a LINQ to XML query that requests the *Zune* from the *products.xml* file that we've been using throughout this chapter:

```
var products = XElement.Load("products.xml");

var zune =
    (from prod in products.Elements("product")
     where prod.Element("name").Value == "Zune"
     select prod)
    .Single();

Console.WriteLine(zune);
```

Here, you can see that the data source is *Elements* with an argument that is the name of the element you are searching for. The data source returned by *products.Elements("product")* returns all of the *product* elements in the XML. The filter uses *Element* (singular) to specify which sub-element to filter by, and *Value* extracts the text between tags. The query returns an *IEnumerable* with one element, and the code uses the *Single* operator to get just one node. The output looks like this:

```
<product productID="1">
  <name>Zune</name>
  <listPrice>250.00</listPrice>
</product>
```

This is the result of searching with a filter on an element, but you can also filter by attribute. Here's an example that retrieves the *Zune* element by using *Attribute* in the *where* clause, producing the same output as the preceding:

```
var products = XElement.Load("products.xml");

var zune =
    (from prod in products.Elements("product")
     where prod.Attribute("productID").Value == "1"
     select prod)
```

```
.Single();
```

```
Console.WriteLine(zune);
```

This filter used *prod.Attribute(“productID”).Value* to return the product with a *productID*, attribute equal to 1.

In addition to the *Elements* method, LINQ to XML offers many axes methods for querying XML, as discussed in the next section.

LINQ to XML Axis Methods

An *axis* is a point of reference from which to view related elements. To demonstrate how they work, we’ll use the following XML document, *products.xml*, to show how each of the LINQ to SQL *axis* methods works:

```
<?xml version=“1.0” encoding=“utf-8”?>
<products>
  <product productID=“1”>
    <name>Zune</name>
    <listPrice>250.00</listPrice>
  </product>
  <product productID=“2”>
    <name>iPod</name>
    <listPrice>299.00</listPrice>
  </product>
</products>
```

If we use the preceding XML document and navigate to the first *product* node, with *productID* attribute of value 1, then that is the current *axis*, which is the point of reference. From the axis, the *products* element is an ancestor; the *name* element with the value *Zune* is a child; and the *product* element with the *productID* attribute of value 2 is a sibling. We’ll use this code to get the *axis* node:

```
var products = XElement.Load(“products.xml”);

var axis =
  (from prod in products.Elements(“product”)
   where prod.Attribute(“productID”).Value == “1”
   select prod)
   .Single();
```

This code is exactly the same as the example in the previous section, except that the variable identifier that holds the query results is named *axis*, instead of *Zune*. Methods in the following sections help work with axes in LINQ to XML, using the *axis* node just created.

Ancestors

You can get the parent element with the *Ancestors* axis method. Here’s an example:

```
var ancestors = axis.Ancestors();

foreach (var ancestor in ancestors)
{
  Console.WriteLine(ancestor);
}
```

Using the *axis* node from the previous section, we call *Ancestors*, which is the entire *products* node.

The *axis* was at *product*, where *productID*=1, so the ancestor is the parent, *products*. There is also an *AncestorsAndSelf* method that will return both the ancestors, as shown earlier, and the node of the axis. Here's an example:

```
var ancestorsAndSelf = axis.AncestorsAndSelf();

foreach (var ancestorOrSelf in ancestorsAndSelf)
{
    Console.WriteLine(ancestorOrSelf);
}
```

The *AncestorsAndSelf* method returned an *IEnumerable<XElement>* of *product* and *products* nodes, shown next:

```
<product productID="1">
  <name>Zune</name>
  <listPrice>250.00</listPrice>
</product>
<products>
  <product productID="1">
    <name>Zune</name>
    <listPrice>250.00</listPrice>
  </product>
  <product productID="2">
    <name>iPod</name>
    <listPrice>299.00</listPrice>
  </product>
</products>
```

This output shows two elements: *product*, where *productID*=1, and the *products* element. When you call one of the *axis* methods with *Self* in the name, you know that it will return the *axis* element, along with other elements for the type of *axis* method. In addition to ancestors, you can retrieve descendants.

Descendants

A descendant is any element that is a child of the *axis*, element which can be retrieved via the *Descendants* axis method. Here's an example:

```
var descendants = axis.Descendants();

foreach (var descendant in descendants)
{
    Console.WriteLine(descendant);
}
```

Calling *Descendants* returns an *IEnumerable<XElement>*. Here's the output:

```
<name>Zune</name>
<listPrice>250.00</listPrice>
```

As just shown, *name* and *listPrice* are children (descendants) of *product* where *productID*=1. In addition to *Descendants*, there is also a *DescendantsAndSelf* method that returns the preceding elements with the *axis* element too.

Elements After/Before Self

You’ve seen in the previous section that the *Elements*, method which is another *axis* method, returns all elements with a specified tag name. You can also use a form of the *Elements*, method that either returns the preceding sibling or following sibling elements of the *axis* element: *ElementsAfterSelf* and *ElementsBeforeSelf*. Here’s an example that finds the elements following the axis:

```
var elementsAfterSelf = axis.ElementsAfterSelf();

foreach (var element in elementsAfterSelf)
{
    Console.WriteLine(element);
}
```

As do all of the other axes methods shown previously, the *ElementsAfterSelf* method returns an *IEnumerable<XElement>*. Here’s the output:

```
<product productID="2">
  <name>iPod</name>
  <listPrice>299.00</listPrice>
</product>
```

In the output, you can see that this is the product with *productID*=2 that follows the *axis* element. The current value of the *axis* variable used in this section doesn’t have any peer elements preceding it, so calling *ElementsBeforeSelf* returns no values. However, we can change the axis, as shown following, to the next product and then call *ElementsBeforeSelf* to get a result:

```
var elementsBeforeSelf =
    elementsAfterSelf
        .First()
        .ElementsBeforeSelf();

foreach (var element in elementsBeforeSelf)
{
    Console.WriteLine(element);
}
```

The preceding code uses a little bit of trickery to get its result. The *elementsAfterSelf* is the result of the previous example, and calling the *First* operator returns the first element in the collection where *productID*=2. Then the code invokes *ElementsBeforeSelf*, on the element returned by *First*. In other words, calling *First* on *elementsAfterSelf* moves the axis to the second *product* element. So, when you call *ElementsBeforeSelf*, you end up with the first *product* element.

This shows that the *axis* methods give you a lot of flexibility to select the nodes you need. In addition to what you’ve seen here, each of the *axis* methods has overloads that allow you to specify the name of the element you are searching for.

Axis Method Filters

All of the preceding examples return all elements meeting the criteria of the *axis* method, but you have more control than that through overloads that allow you to filter the results further. For example, *DescendantsAndSelf*(“name”) would return the *name* element under the product and the *axis* element where the element tag name is “name”:

```
var descendantNameAndSelf = axes.DescendantsAndSelf("name");
```

```
foreach (var element in descendantNameAndSelf)
{
    Console.WriteLine(element);
}
```

Notice that the call to *DescendantsAndSelf* passes in the string argument, “name”, which is a string, but the parameter type is *XName*. Like *XNamespace*, *XName* has an implicit string conversion operator. In addition to overloads with no parameters, some of the *axis* methods have overloads that will accept an *XName* argument, just like *DescendantsAndSelf*. The *XName* argument filters the results to only elements with matching tags. The output is shown next:

```
<name>Zune</name>
```

It’s important to recognize that the preceding element does not contain the *axis* element and that the tag for the element is *name*. This demonstrates how *XName* arguments to *axis* methods work. The *axis* method invoked was *DescendantsAndSelf*, which would have returned the *axis* element and all child nodes if called without parameters. However, the *XName* argument “name” is passed in this example. Since the tag for the *axis* element is *product*, there isn’t a match, and the *axis* element is filtered out of the results. The child *name* element is included, as you can see in the results, but the *listPrice* element isn’t included in the results because its tag doesn’t match the filter.

Now that you know how to find any element in an XML document, you need to know how to manipulate the contents of the XML document, which is covered next.

Manipulating XML

You have the ability to modify the contents of an XML document through insert, update, and delete operations. Actually, specific methods in the LINQ to XML API allow you to do this. All of the examples use the same *products.xml* file that we’ve been using throughout the chapter. The following code shows how we are reading that file:

```
var root = XElement.Load("products.xml");
```

This is the same code that was shown earlier in the chapter, when showing you how to load XML from a file, except that the variable name for the results is now called *root*. The following sections will show you how to add, modify, and remove elements by using the *IEnumerable<XElement>* variable, *root*.

Adding New Elements

You can add a new element to an XML document by calling the *Add* method. Here’s an example that adds a new *product* element for a Smart Phone to the *products* list:

```
var smartPhone =
    new XElement("product",
        new XAttribute("productID", "3"),
        new XElement("name", "Smart Phone"),
        new XElement("listPrice", "575.00"));

root.Add(smartPhone);

Console.WriteLine(root);
```

Here, I used functional construction to create the new element and then passed that new element to the

Add method, which is invoked through *root*. The result is that the new element is added to the child list of *root*, shown here:

```
<products>
  <product productID="1">
    <name>Zune</name>
    <listPrice>250.00</listPrice>
  </product>
  <product productID="2">
    <name>iPod</name>
    <listPrice>299.00</listPrice>
  </product>
  <product productID="3">
    <name>Smart Phone</name>
    <listPrice>575.00</listPrice>
  </product>
</products>
```

The new Smart Phone product shows up as the last product in the preceding list. There are additional *Add* methods: *AddFirst* adds an element as the first child, rather than the last; *AddBeforeSelf* adds a sibling element preceding the axis where the method was called; and *AddAfterSelf* adds a sibling element following the axis where the method was called. For more specific positioning, use the *axis* methods to get a reference to an element to use as the axis, and then call *AddBeforeSelf* or *AddAfterSelf* to insert the new element.

You can also modify elements, which is discussed next.

Modifying Existing Elements

When an element already exists, you can change its values. First, use the *axis* methods to get a reference to the element. Then you can either change the *Value* property of the element you have or use additional *axis* methods to modify attributes or child elements. Here’s an example that changes the *listPrice* of a product:

```
var prodToChange =
  (from prod in root.Elements("product")
   where prod.Attribute("productID").Value == "1"
   select prod)
  .Single();

Console.WriteLine("Before: \n" + prodToChange);

prodToChange.Element("listPrice").Value = "239.95";

Console.WriteLine("After: \n" + prodToChange);
```

First, you need a reference to the element that you want to change. To start, the preceding code gets a reference to the *product* element to change. The *product* element is really the parent element of the node that will be changed, *prodToChange*. With the parent element, the code uses the *Element* method to reference the *listPrice* child element. The *listPrice* is the element to be changed, so the example assigns 239.95, to the *Value*, property of the reference to the *listPrice* element. This demonstrates the effective implementation of *axis* methods and shows that you should access or modify the *Value* property of an element reference to change the contents of an XML element. Here’s the output:

Before:

```

<product productID="1">
  <name>Zune</name>
  <listPrice>250.00</listPrice>
</product>
After:
<product productID="1">
  <name>Zune</name>
  <listPrice>239.95</listPrice>
</product>

```

The output shows that setting the *Value* property of the element changed the text in the XML from *250.00* to *239.95*.

The final data manipulation task to show you is how to remove elements, covered next.

Removing Elements

You can delete items from an XML document by calling *Remove* on the object reference. Here's an example that removes an element:

```

var prodToDelete =
    (from prod in root.Elements("product")
     where prod.Attribute("productID").Value == "2"
     select prod)
    .Single();

prodToDelete.Remove();

Console.WriteLine(root);

```

The first task was to get a reference to the item to be removed, *prodToDelete*. After you have a reference to the element to delete, call the *Remove* method on the instance of the element to delete. Calling *Remove* on *prodToDelete* will delete *prodToDelete* from the XML, as shown here:

```

<products>
  <product productID="1">
    <name>Zune</name>
    <listPrice>239.95</listPrice>
  </product>
  <product productID="3">
    <name>Smart Phone</name>
    <listPrice>575.00</listPrice>
  </product>
</products>

```

This shows that the element with *productID=2* is no longer part of the XML.

Summary

You can now work with XML effectively by using LINQ to XML. This chapter showed you how to create an XML document via several methods, including functional construction. You can also save and load your XML and understand several of the *Load* method overloads you can use. A section of this chapter described how to use namespaces. The section on *axis* methods demonstrated how to navigate an XML document via an *axis* element. With knowledge of how to navigate and select elements in the document, you then learned how to manipulate XML contents, performing add, modify, and remove operations.

In previous chapters, you learned about LINQ to Objects, LINQ to SQL, LINQ to DataSet, and LINQ to Entities. This chapter rounded out coverage of all LINQ providers that ship with the .NET Framework by covering LINQ to XML. In the next chapter, you'll learn about various tools that you might need to use as you master using LINQ to build applications.

CHAPTER 7

Automatically Generating LINQ to SQL Code with SqlMetal

Visual Studio 2008 (VS 2008) has great tools for working with LINQ, but they aren't the only tools available, nor do they constitute the complete set you need for working with LINQ. For example, you can use the SqlMetal command-line tool (which ships with the .NET Framework 3.5 SDK) to generate DBML files, data context and entity class files, and external mapping files for use with LINQ.

A DBML file is the same file type as you see in a LINQ to SQL project item with the *.dbml file extension. When you're working with VS 2008, DBML files are an attractive option because they facilitate working in the visual designer. Creating the DBML file by using SqlMetal outside of Visual Studio might be desirable to support automated tools or as the quickest way to create the DBML file for large databases.

When you're working with VS 2008, the LINQ to SQL project creates a *.designer.cs file that holds your data context and entity class source code. Similar to the reason you would want to generate a DBML file outside of VS 2008, the database might be so large that it would take a long time for the visual designer to accommodate all of the entities. Generating the file with SqlMetal would be quicker.

Another option for working with LINQ to SQL data outside of VS 2008 is to define an external mapping file, which is an XML file that maps database objects to code. This could be a viable option if you don't want to add attributes to your code and would prefer the cleaner interface of performing the mapping outside of the code. Additionally, you might have an external tool that helps you manage the mapping better than SqlMetal, and now you have a standard XML format that you can generate yourself.

The following sections describe the SqlMetal options to create DBML and class files. You'll then learn the format of the external mapping file. Finally, you'll learn about a couple other tools to make your LINQ to SQL work easier.

Using SqlMetal.exe

SqlMetal is a command-line tool that can generate new object models, entity code, or external XML mapping files. You can find SqlMetal at C:\Program Files\Microsoft SDKs\Windows\v6.0A\bin\SqlMetal.exe. Fix the path to work with the computer that you'll be working on. Via command-line options, you can specify and control SqlMetal output to produce Database Markup Language (DBML) files, data context and entity class files, and external mapping files.

Database Connection Options

When specifying the database to connect to, you can either use a set of database connection options or specify a connection string. The database connection options include */server*, */database*, */user*, and */password*. The following example (type this on the command line) uses these options and adds the */dbml* option to create a DBML file:

```
SqlMetal.exe /server:.\sqlexpress /database:adventureworkslt_data  
/user:MyUserID /password:MyPassword /dbml:adventureworks.dbml
```

The */server* option specified the database server being used. In this case, it's the named instance for SQL Server 2005 Express on the local machine. The */database* option holds the name of the database,

which could be different on your system. The `/user` and `/password` options hold login credentials, are optional, and default to windows authentication if you omit them. The `/dbml` option generates a *.dbml file that you can include with your project. I'll describe the DBML file and how to use it in the next section.

When you run this command, SqlMetal will also make checks for any potential incompatibilities. For example, when running the preceding command on the AdventureWorks database, you'll receive the following warning message:

```
Warning DBML1008: Mapping between DbType 'Decimal (38,6) NOT NULL' and Type
'System.Decimal' in Column 'LineTotal' of Type 'SalesLT_SalesOrderDetail'
may cause data loss when loading from the database.
```

This highlights an important point. While LINQ to SQL reduces the impedance mismatch between object-oriented and relational paradigms, it doesn't eliminate it. You still must be aware of some type system differences. In the preceding example, the warning occurred because the column type is Decimal (38,6), but a C# decimal has 28 significant digits. Because of the size difference, there is potential for data loss. In this particular case, the column in question, *LineTotal*, is a computed column with the following formula:

```
isnull(([UnitPrice]*((1.0)-[UnitPriceDiscount]))*[OrderQty],(0.0)),0)
```

which I rewrote as:

```
(CONVERT([money],isnull(([UnitPrice]*((1.0)-
[UnitPriceDiscount]))*[OrderQty],(0.0)),0))
```

to eliminate the warning. Depending on the column being converted—its meaning, your requirements, and potential for data loss—this may or may not make sense for you. You'll have to look at your own situation for resolving such warnings to see if it makes sense.

In a way, SqlMetal can be handy in helping you find such potential problems.

Another way to specify the database connection is via a connection string. Here's an example that uses the `/conn` option to specify a connection string:

```
SqlMetal.exe /conn:"Data Source=.\sqlexpress;AttachDbFilename='C:\Program
Files\Microsoft SQL Server\MSSQL.1\MSSQL\Data\AdventureWorksLT_Data.mdf';
Initial Catalog=AdventureWorksLT_Data;Integrated Security=True"
/dbml:AdventureWorks.dbml
```

When entering the connection string, pay particular attention to quotes. The value of the connection string following `/conn` is enclosed in double quotes. Remember that the preceding connection string should be typed on the same line. Any contents of the connection string that need to be quoted must have single quotes as does the value of the *AttachDBFilename*.

You have a choice of using either the combination of `/server` and `/database` (and optionally the `/user` and `/password`) options, or only the one `/conn` option, but not both.

Both of the examples in this section used the `/dbml` option to create a DBML file. The next section describes what that option means and how to use it.

Working with DBML Files

A DBML file is used by the VS 2008 designer to manage entities. You could create DBML files in the VS

2008 designer, via the LINQ to SQL project item wizard. If your database is large, VS 2008 will take a long time to generate files. Therefore, generating the DBML via SqlMetal might be a desirable option. Another reason to auto-generate a DBML file is if you have your own automated tool to modify DBML file contents before generating C# code. The SqlMetal command executed in the last section generates an XML file that looks like this:

```
<?xml version="1.0" encoding="utf-8"?>
<Database Name="AdventureWorksLT_Data"
  xmlns="http://schemas.microsoft.com/linqtosql/dbml/2007">
  <Table Name="SalesLT.Address"
    Member="SalesLT_Address">
    <Type Name="SalesLT_Address">
      <Column Name="AddressID"
        Type="System.Int32"
        DbType="Int NOT NULL IDENTITY"
        IsPrimaryKey="true"
        IsDbGenerated="true"
        CanBeNull="false" />
      ...
    </Type>
  </Table>
</Database>
```

I've removed much of the content and formatted the AdventureWorks.dbml content just shown. As you can see, it is simply an XML file that holds database schema definitions. Because the *.dbml is based on the database it maps, you are likely to see differences between what you see here and the XML produced in your own project. As you know, *.xml files have their own editor in VS 2008, so the DBML file is named with the *.dbml extension so that VS 2008 can recognize its purpose and open the proper designer for it.

The preceding command only extracted tables from the database, but there are options to extract functions, stored procedures, and views too. The */functions* option will extract all functions; */sprocs* will extract all stored procedures; and */views* will extract all views. Type the following to extract all database objects:

```
SqlMetal.exe /server:.\sqlexpress /database:adventureworkslt_data
/functions /sprocs /views /dbml:adventureworks.dbml
```

As you can see, */functions*, */sprocs*, and */views*, don't have parameters. You just specify them, and they'll be added to the DBML file.

To use this DBML file in VS 2008, create a new Console project, right-click on the project in Solution Explorer, select Add | Existing Item, set the File Name filter to All Files, select AdventureWorks.dbml, and click the Add button. VS 2008 will add the *.dbml file and will also generate an adventureworks.dbml.layout and an adventureworks.designer.cs file. The adventureworks.dbml.layout file holds all of the information for positioning entities on the design surface. If you were to look inside it, you would see XML that looks similar to this:

```
<?xml version="1.0" encoding="utf-8"?>
<ordesignerObjectsDiagram dsIVersion="1.0.0.0"
  absoluteBounds="0, 0, 12, 18"
  name="adventureworks">
  <DataContextMoniker Name="/AdventureWorksLT_Data" />
  <nestedChildShapes>
```

```

<classShape Id="d9364730-b124-4ba0-9e7d-06892787f0e2"
    absoluteBounds="0.75, 5.375, 2, 2.5401025390625005">
    <DataClassMoniker Name="/AdventureWorksLT_Data/SalesLT_Address" />
    <nestedChildShapes>
    ...

</ordesignerObjectsDiagram>

```

I’ve formatted and removed a lot of the content from the preceding file, but you can see that it keeps track of all the objects in the designer, sizes, locations, and so on. The other file that VS 2008 automatically generated, `adventureworks.designer.cs`, contains C# code for the `DataContext` and entities defined by the DBML file, which match the contents of the database by default.

The `adventureworks.designer.cs` file is the same `*.designer.cs` file that VS 2008 generates when you use the LINQ to SQL project item wizard. However, you can generate this file on your own with another `SqlMetal` command-line option, `/code`. The `*.designer.cs` file reflects the contents of the DBML file, so you will need the DBML file to generate it. Here’s an example of how to use the `adventureworks.dbml` file to generate a code file containing `DataContext` and *Entity* classes:

```
SqlMetal.exe /code:adventureworks.designer.cs adventureworks.dbml
```

The `/code` option specifies the C# file that will be generated, and the `adventureworks.dbml` file specifies the DBML file that is the source for the C# file. Since the `/code` option is being used to generate code from existing DBML, the `/dbml` switch isn’t valid here. Here’s an excerpt from the `adventureworks.designer.cs` file generated from the preceding command line:

```

#pragma warning disable 1591
//-----
// <auto-generated>
//   This code was generated by a tool.
//   Runtime Version:2.0.50727.3033
//
//   Changes to this file may cause incorrect
//   behavior and will be lost if
//   the code is regenerated.
// </auto-generated>
//-----

using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Data.Linq;
using System.Data.Linq.Mapping;
using System.Linq;
using System.Linq.Expressions;
using System.Reflection;

[System.Data.Linq.Mapping.DatabaseAttribute(
    Name="adventureworkslt_data")]
public partial class Adventureworkslt_data :
    System.Data.Linq.DataContext
{
    ...

    public System.Data.Linq.Table<SalesLT_Address>
        SalesLT_Address

```

```

    {
        get
        {
            return this.GetTable<SalesLT_Address>();
        }
    }
    ...
}

[Table(Name="SalesLT.Address")]
public partial class SalesLT_Address :
    INotifyPropertyChanging,
    INotifyPropertyChanged
{
    ...
}

```

For readability, I've reformatted the preceding code and removed code to save space. I would like to make several points about this code that will contribute to the discussion on SqlMetal source code options in the next section.

- All of the code is in the global namespace.
- The base class, *Adventureworkslt_data*, is generated from the DBML file.
- *Adventureworkslt_data*, derives from *DataContext*.
- *SalesLT_Address* is the singular name of the table where the schema is *SalesLT* and the table name is *Address*.
- The source code is C#.
- Neither the *DataContext* nor any of the entity classes, for example, *SalesLT_Address*, are serializable.

All of the items in the preceding list are configurable, and I'll show you how to change them in the next section.

SqlMetal Source Code Options

SqlMetal includes several options for managing source code: language, namespace, DataContext name, base class, type name pluralization, and serialization. Here's an example for you to type on the command line, which sets all of these options at one time:

```

SqlMetal.exe /language:C# /namespace:McGrawHill.LINQProgramming
/context:AdventureWorksDataContext /entitybase:LINQProgrammingBaseEntity
/pluralize /serialization:Unidirectional /code:AdventureWorks.designer.cs
adventureworks.dbml

```

Starting with the */language* option, you have a choice between VB.NET or C#, where the */language* option value is either *VB* or *C#*, respectively. The preceding code generates a code file in C#. There is an interesting default for this option in that the file extension of the */code* option value will determine the output file format too. A file extension of *.vb creates a VB file, and a file extension of *.cs creates a C# file. The */language* option overrides whatever file extension is used in the */code* option.

The */namespace* option puts the generated code in the specified namespace. The default is to generate the code in the global namespace. Code in global namespaces can be a problem because you might have more than one DataContext in your application, which could have many of the same names in each DataContext. Putting each code file in separate namespaces helps avoid this problem.

The default DataContext name comes from the DBML file. You can change this by specifying the */context* option. In the preceding example, the DataContext name will be *AdventureWorksDataContext*,

instead of the default `Adventureworkslt_data`.

You can have all of your entity classes derive from a common base class by using the `/entitybase` option. By default none of the entity classes has an explicit base class. The preceding example makes all of the entity classes derive from `LINQProgrammingBaseEntity`.

Pluralization means that entity names are generated in their plural form; for example, instead of *Product*, you would have *Products*. The `/pluralize` option turns pluralization on. The default is to not pluralize. The `/pluralize` option only works in English; any other language might produce funny results (or not so funny depending on the meaning of what is produced).

To understand the `/serialization` option, you need some background on where it's used and the problems that can occur. LINQ to SQL has a performance enhancement feature called *deferred loading*, which ensures that associated entities aren't loaded until requested. A later chapter discusses the mechanism of deferred loading, performance tips and traps, and how to control it. For right now, it's important to know that deferred loading can happen regardless of where your objects are in the architecture. For example, you might need to transfer entities from a web service or a separate AppDomain, requiring serialization to move the object across the wire. During serialization, an object graph is *serialized* (translated from object to wire format) and then *deserialized* (translated from wire format to object) when it arrives at the destination. The problem in serialization occurs because of associations between parent and child objects that result in circularities. This circularity is a problem because it would result in a never-ending loop, causing a stack overflow, if not controlled. The `/serialize` option is the mechanism through `SqlMetal` that you use to control this process. First, I'll discuss a more concrete example so you can see what the `/serialize` option does to code.

The example I'll use is the *Product* and *ProductCategory* classes. A *ProductCategory* class serves as the parent class because a single instance of *ProductCategory* categorizes many *Product* instances. If the *ProductCategory* is Mountain Bike, then the Mountain-100 Silver 38 and Mountain-W-100 Silver 46 *Products* would belong to the Mountain Bike *ProductCategory*.

The result of the `/serialize` option is either *None* or *Unidirectional*. *None* is the default and means that there is no serialization support. The only other serialization value is *Unidirectional*, which solves the circular association deserialization problem described earlier. *Unidirectional* means that only one side of an association contains serialization attributes. The one side of the association is the parent type in the relationship, which is why I made a point of defining *ProductCategory* as having a parent association with *Product*. The following code shows how the serialization option affects code. The following code is an excerpt from `adventureworks.designer.cs`. It contains the `SalesLT_ProductCategory` property of the *Product* class, which is an association from *Product* to *ProductCategory*:

```
[Association(
    Name="FK_Product_ProductCategory_ProductCategoryID",
    Storage="_SalesLT_ProductCategory",
    ThisKey="ProductCategoryID", IsForeignKey=true)]
public SalesLT_ProductCategory SalesLT_ProductCategory
```

As stated earlier, the parent in the association is decorated with the serialization attribute, but the child isn't. You can see from the preceding code that the only attribute decorating *SalesLT_ProductCategory* is *Association*. There is nothing else.

That was the child class, *Product*, of the parent, *ProductCategory*. Now, let's look at the parent class

in the relationship, *ProductCategory*. The following example is the association in *ProductCategory*, the parent, to *SalesLT_Product*, the child. It demonstrates that the serialization attribute, *DataMember*, appears in the parent class:

```
[Association(
    Name="FK_Product_ProductCategory_ProductCategoryID",
    Storage="_SalesLT_Product",
    OtherKey="ProductCategoryID",
    DeleteRule="NO ACTION")]
[DataMember(Order=6, EmitDefaultValue=false)]
public EntitySet<SalesLT_Product> SalesLT_Product
```

The *DataMember* attribute supports *DataContract* serialization, which is what Windows Communications Foundation (WCF) uses by default. Because only the parent association can be serialized, you avoid circularities during the deserialization process. In our SqlMetal command-line example shown earlier, */serialization* is set to *Unidirectional*, which ensures that *DataContract* attributes decorate entity classes and that *DataMember* attributes decorate entity members, unless they are child associations to parents.

Implementing External Mapping Files

An external mapping file is an XML file that performs the mapping between the .NET objects that you code and database objects. Instead of decorating your objects with attributes, as described in the LINQ to SQL chapter, you can manage the relationship between your objects and the database via an external XML file.

There are a few scenarios where using external mapping files might be useful: easier modifications to mapping, supporting different data sources, external mapping tool support, having a cleaner object model, or using objects when you don't have access to the source code. The following sections show you how to use external mapping files so that you can see how these scenarios could be implemented.

Generating an External Mapping File with SQLMetal.exe

In addition to the attributed object approach (where object mapping is defined through attributes), SQLMetal lets you generate an external mapping file, based on existing database schema. Once you've generated the first external mapping file, you can change it without needing to recompile the application. The following command line uses SQLMetal to generate an external mapping for the AdventureWorksLT database:

```
SqlMetal.exe /map:adventureworks.map /code:AdventureWorks.designer.cs
adventureworks.dbml
```

The */map* option in the preceding command specifies the filename of the external mapping file. To use this option, you'll need to provide both the */code* and DBML file. Here's an excerpt from the mapping file:

```
<?xml version="1.0" encoding="utf-8"?>
<Database Name="adventureworkslt_data"
xmlns="http://schemas.microsoft.com/linqtosql/mapping/2007">
...
<Table Name="SalesLT.Product"
    Member="SalesLT_Product">
    <Type Name="SalesLT_Product">
```

```

<Column Name="ProductID"
  Member="ProductID"
  Storage="_ProductID"
  DbType="Int NOT NULL IDENTITY"
  IsPrimaryKey="true"
  IsDbGenerated="true"
  AutoSync="OnInsert" />
<Column Name="Name"
  Member="Name"
  Storage="_Name"
  DbType="NVarChar(50) NOT NULL"
  CanBeNull="false" />
<Column Name="ProductNumber"
  Member="ProductNumber"
  Storage="_ProductNumber"
  DbType="NVarChar(25) NOT NULL"
  CanBeNull="false" />
...
<Column Name="ProductCategoryID"
  Member="ProductCategoryID"
  Storage="_ProductCategoryID"
  DbType="Int" />
...
<Association Name="FK_Product_ProductCategory_ProductCategoryID"
  Member="SalesLT_ProductCategory"
  Storage="_SalesLT_ProductCategory"
  ThisKey="ProductCategoryID"
  OtherKey="ProductCategoryID"
  IsForeignKey="true" />
...
</Type>
</Table>
<Table Name="SalesLT.ProductCategory"
  Member="SalesLT_ProductCategory">
<Type Name="SalesLT_ProductCategory">
  <Column Name="ProductCategoryID"
    Member="ProductCategoryID"
    Storage="_ProductCategoryID"
    DbType="Int NOT NULL IDENTITY"
    IsPrimaryKey="true"
    IsDbGenerated="true"
    AutoSync="OnInsert" />
  <Column Name="ParentProductCategoryID"
    Member="ParentProductCategoryID"
    Storage="_ParentProductCategoryID"
    DbType="Int" />
  <Column Name="Name"
    Member="Name"
    Storage="_Name"
    DbType="NVarChar(50) NOT NULL"
    CanBeNull="false" />
...
  <Association Name="FK_Product_ProductCategory_ProductCategoryID"
    Member="SalesLT_Product"
    Storage="_SalesLT_Product" ThisKey="ProductCategoryID"
    OtherKey="ProductCategoryID"
    DeleteRule="NO ACTION" />
  <Association Name=
"FK_ProductCategory_ProductCategory_ParentProductCategoryID_

```



```

ProductCategoryID”
    Member=“ParentProductCategory”
    Storage=“_ParentProductCategory”
    ThisKey=“ParentProductCategoryID”
    OtherKey=“ProductCategoryID”
    IsForeignKey=“true” />
<Association Name=

“FK_ProductCategory_ProductCategory_ParentProductCategoryID_
ProductCategoryID”
    Member=“ProductCategory”
    Storage=“_ProductCategory”
    ThisKey=“ProductCategoryID”
    OtherKey=“ParentProductCategoryID”
    DeleteRule=“NO ACTION” />
</Type>
</Table>
...

```

You can see the mapping in both the *Table* and *Type* elements. The *Table Name* attribute maps to the nested *Type* element’s *Name* attribute. In the case of the *Product* table, the *SalesLT.Product* table maps to the *SalesLT_Product* class—the difference being the dot that separates schema from the table name being translated to an underline in the class name.

Notice that the *Column* elements contain attributes that are the same as the C# *ColumnAttribute* attribute properties that I described in [Chapter 3](#). Similarly, *Association* elements have attributes that map to the C# *AssociationAttribute* attribute properties that you saw in [Chapter 3](#).

Using the External Mapping File in Your Code

To use the external mapping file, you can either use your own business objects or use auto-generated DBML entities. This opens another opportunity in that you can use existing business objects and define a mapping file to work with your custom business objects with LINQ. After you have business objects defined, you’ll need to define the mapping file. You saw how to do this automatically in the previous section, but I’ll show you how to define your own mapping file that works with your custom business objects. Then you need to instantiate a new *DataContext* that targets your mapping file, instead of the DBML file that you’ve seen used by the LINQ to SQL designer. We’ll still use the *AdventureWorksLT* database that we’ve been using throughout the book, including this chapter.

Define Custom Business Objects

The custom business object we’ll use in this example is a *Product* class that holds a subset of the columns available in the *Product* table, shown next:

```

public class Product
{
    private int _ProductID;
    private string _Name;

    public int ProductID
    {
        get
        {
            return _ProductID;
        }
    }
}

```

```

        set
        {
            _ProductID = value;
        }
    }

    public string Name
    {
        get
        {
            return _Name;
        }
        set
        {
            _Name = value;
        }
    }
}

```

This *Product* class has only two properties, *ProductID* and *Name*. The property actions don't include any of the DataContext tracking and management code that is typically automatically generated in DataContext derived types via the LINQ to SQL project item wizard or SqlMetal, but you could add the tracking and management code if you wanted. Just refer to [Chapter 3](#), and emulate similar code from the automatically generated entity file, *adventureworks.designer.cs*. Next, you'll see the mapping file.

Define a Custom External Mapping File

The external mapping file has a schema, named *LinqToSqlMapping.xml*, that is defined in the .NET Framework 3.5 documentation and located at *C:\Program Files\Microsoft Visual Studio 9.0\Xml\Schemas*. Using the schema defined in *LinqToSqlMapping.xml*, the following external mapping file, named *products.map*, maps the *Product* table to the custom *Product* business object. You could also create this file by generating a mapping file with SqlMetal, as described in the previous section, and removing all the XML except for what is required for the *Product* table:

```

<?xml version="1.0" encoding="utf-8"?>
<Database Name="adventureworkslt_data"
xmlns="http://schemas.microsoft.com/linqtosql/mapping/2007">
  <Table Name="SalesLT.Product" Member="Product">
    <Type Name="Product">
      <Column Name="ProductID"
        Member="ProductID"
        Storage="_ProductID"
        DbType="Int NOT NULL IDENTITY"
        IsPrimaryKey="true"
        IsDbGenerated="true"
        AutoSync="OnInsert" />
      <Column Name="Name"
        Member="Name"
        Storage="_Name"
        DbType="NVarChar(50) NOT NULL"
        CanBeNull="false" />
    </Type>
  </Table>
</Database>

```

In the preceding mapping file, you can see that the *SalesLT.Product* database table maps to the *Product*

class, and that the *ProductID* and *Name* columns map to the *ProductID* and *Name* properties in *Product*. Now, you'll need to write code to use this mapping file.

Loading External Mapping Files with a DataContext

Instead of the default *DataContext* constructor, you'll need to use an overload that accepts a connection string and an *XmlMappingSource*. The connection string specifies the database to connect to and is the same as any other connection string you use for .NET data access, depending on the database you are connecting to.

The *XmlMappingSource* will identify the *products.map* external mapping file that you saw earlier. To get the following code to work as is, make sure that you put the *products.map* file in the same folder as the executable. Otherwise, you'll need to specify the path where the *products.map* file is located. You'll need to add a project reference to the *System.Data.Linq.dll* assembly. Additionally, you'll need namespace declarations for the *System.Linq*, *System.Data.Linq*, and *System.Data.Linq.Mapping* namespaces. The following code sets up and uses the external mapping file:

```
var connStr = @"Data Source=.sqlexpress;AttachDbFilename=
""C:\Program Files\Microsoft SQL Server\MSSQL.1\MSSQL\Data\
AdventureWorksLT_Data.mdf"";Initial Catalog=AdventureWorksLT_Data;Integrated
Security=True";
var mapSrc = XmlMappingSource.FromUrl("products.map");
var ctx = new DataContext(connStr, mapSrc);

var products =
    from prod in ctx.GetTable<Product>()
    select prod;
```

As I promised, the connection string is the same that you use with the DBML file. The code uses the *FromUrl* method of *XmlMappingSource* to reference the *products.map* file, which is the external mapping file.

Since we don't have a DBML file in this example, we also don't have a *DataContext* derived class to use, like we did in [Chapter 3](#) for the *AdventureWorksDataContext* and the *Adventureworkslt_data* class that was automatically generated in the *adventureworks.designer.cs* file earlier in this chapter. That's okay because we can still use the *DataContext* class. Creating a *DataContext* derived type would allow you to provide table properties that encapsulate calls to *GetTable*, or to encapsulate functions and stored procedure calls, just like the automatically generated *DataContext* derived types created by the LINQ to SQL designer or the *SqlMetal* program described in this chapter.

The preceding example uses the *DataContext* overload that accepts a connection string and a *MappingSource*. The *XmlMappingSource* class derives from *MappingSource* and is the proper type to pass for using an external mapping source. There's also an *AttributeMappingSource* class that derives from *MappingSource*, allowing you to specify classes that use attributes, much like the automatically generated entities in *adventureworks.designer.cs*. The LINQ to SQL project item wizard generates an *AttributeMappingSource* for you by default. For simplicity, the preceding example only instantiates a *DataContext*, using the external mapping file, *products.map*.

If you recall from [Chapter 3](#), the *DataContext* derived type that is automatically generated by the LINQ to SQL wizard contains properties that encapsulate table access via calls to *GetTable<T>*. Since we're

instantiating a `DataContext` directly, we don't have those properties. The preceding example calls `GetTable<Product>`, as the data source in a LINQ to SQL query. There isn't anything different about this LINQ query that you haven't seen before, and you can iterate through products to get *Product* info.

Summary

You should now understand how `SqlMetal` works and what it is used for. You can generate DBML files, code files, and external mapping files. There are several options to call `SqlMetal` with, and they help you specify the database to connect to, which database objects to include, and options for generating output files.

While you learned how to use `SqlMetal`, you also learned why you would use it in certain scenarios. Essentially, `SqlMetal` gives you additional flexibility in performance and external tool support that you don't have with VS 2008.

This was the last chapter on working with LINQ providers that ship with the .NET Framework. The next chapter, which digs into expression trees and lambdas, starts you in another direction of understanding how LINQ works.

PART III

Extending LINQ

CHAPTER 8

Digging Into Expression Trees and Lambdas

CHAPTER 9

Constructing New Code with Extension Methods

CHAPTER 10

Building a Custom LINQ Provider—Introducing LINQ to Twitter

CHAPTER 11

Designing Applications with LINQ

CHAPTER 12

Concurrent Programming with Parallel LINQ (PLINQ)

CHAPTER 8

Digging Into Expression Trees and Lambdas

You’ve seen where lambda expressions offer a convenient syntax for event handling, in place of anonymous methods and delegates, but that’s only part of the story. Lambda expressions are part of the inner workings of LINQ technology. There must be some way to translate logic into a form such as SQL that can be sent to a data source. The code is executable, but you need a way to turn it into data at run time that is transmitted to the data source. Lambda expressions are the executable part of the LINQ equation.

Working hand-in-hand with lambda expressions is the expression tree, which is the data part of the LINQ equation. The relationship between lambda expressions and expression trees is that you can assign a lambda expression directly to an expression variable, building an expression tree at run time. What follows is an explanation of how lambda expressions work, followed by in-depth coverage of expression trees. Then you’ll see an example of how to apply expression trees by building dynamic LINQ queries.

Working with Lambda Expressions

In [Chapter 1](#), I gave you a quick overview of lambda expressions (lambdas) to help you understand the syntax because I’ve used lambda expressions throughout the book. There’s much more to lambdas than being surrogates for delegates and anonymous methods; you can create expression trees with lambdas, supporting the underlying infrastructure driving LINQ. Before getting into expression trees, I’ll explain more of the features of lambdas to help you gain more familiarity with how they can be used. I’ll assume you’ve read about lambdas in [Chapter 1](#) and understand the basic syntax.

More Lambda Examples

You’ve seen lambdas used in previous chapters of this book, but I want to give you more examples so you can become more comfortable using them in different scenarios. I’ll show you a couple lambdas that use a single parameter for the *Action<T>* and *Converter<TInput, TOutput>* delegates. Here’s an example of using a lambda with *Action<T>*:

```
new List<string>
{
    "One", "Two", "Three"
}
.ForEach(
    num => Console.WriteLine(num)
);
```

While you might not construct your *List<T>* explicitly, as just shown, you can invoke *ForEach* with a lambda on any collection. The *ForEach* method takes an argument of type *Action<T>* and infers the collection type, which is the type of the lambda parameter, *num*. Since the lambda body is a single statement, you don’t need curly braces or a semicolon.

Here’s an implementation of the *Converter<TInput, TOutput>* delegate that accepts an argument and returns a value. It shows you how to use the ternary operator to perform more complex logic with a single statement:

```
Console.WriteLine("Second Item: " +
    new List<string>
    {
```

```

        "One", "Two", "Three"
    }
    .ConvertAll<int>(<
        input =>
            input == "One" ? 1 :
            input == "Two" ? 2 :
            3
    >)[1]
);

```

This example is a little more complex, accepting a *string* as input and returning an *int* as output. Notice how I used nested ternary operators to keep the lambda in a single statement. Also, I appended an indexer to *ConvertAll* to read the second item from the newly converted list, which is the number 2. In this example, the return value was implied by the result of the statement.

Sometimes, a single statement is required in a LINQ query—most notably with the *where* clause. Therefore, some degree of sophistication in a single statement lambda might be acceptable. The problem is that anything beyond the ternary operator in the preceding example might be pushing the boundaries of prudence and code maintainability in a wide range of contexts. Fortunately, lambda expressions allow additional syntax for parameter type, blocks of multiple statements, and explicit return values. The following example rewrites the logic in the previous ternary operator example, showing you the additional lambda syntax:

```

Console.WriteLine("Second Item: " +
    new List<string>
    {
        "One", "Two", "Three"
    }
    .ConvertAll<int>(<
        (string input) =>
        {
            int convertedResult = 0;

            switch (input)
            {
                case "One":
                    convertedResult = 1;
                    break;
                case "Two":
                    convertedResult = 2;
                    break;
                case "Three":
                    convertedResult = 3;
                    break;
                default:
                    throw new ArgumentException(
                        input + " can't be converted.");
            }

            return convertedResult;
        }
    >)[1]);

```

The preceding example demonstrates a formal parameter list, multiple statements, and an explicit return value. The parameter list is enclosed in parentheses and declares the parameter type; you would

use parentheses to enclose a comma-separated parameter list for multiple parameters. The curly braces define the lambda block for multiple statements with semicolon line terminators. The *return* statement explicitly returns the *convertedResult* variable that is defined and assigned inside of the lambda block. In summary, the preceding example differs from the single statement lambda (with a single parameter), which doesn't require type and parentheses, an enclosing block, semicolons, or a *return* statement.

While lambdas can be used as handlers for any delegate type, .NET 3.5 adds a new set of delegates, *Func*, which are used in LINQ provider APIs.

Lambdas and the Func Delegates

Func delegates are ubiquitous in LINQ; they're a set of delegates with between one and five type parameters, making them flexible for any operations requiring lambdas of the number of parameters required. I'll show you a couple examples where *Func* can be used in LINQ.

What you'll see is the LINQ to Objects *Select* operator, which is a method. When you perform a projection, it is this *Select* operator that is actually called. In [Chapter 9](#), you'll learn about these operators in more depth. To stay on topic, the current discussion will show you how *Select* and overload accept parameters of delegate type *Func*<*T*, *TResult*>, and *Func*<*T1*, *T2*, *TResult*>, respectively. With the *Func* delegates, the last (or only) parameter is the return type. Each of the examples will use the following *List*<*Product*>, where *product* is defined as:

```
class Product
{
    public string Name { get; set; }
    public decimal ListPrice { get; set; }
}
```

and *List*<*Product*>, is defined as:

```
var products =
    new List<Product>
    {
        new Product
        {
            Name = "Bicycle",
            ListPrice = 500m
        },
        new Product
        {
            Name = "Zune",
            ListPrice = 250m
        }
    };
};
```

Subsequent examples will use *products*, which is the preceding *List*<*Product*>.

First, here's a call to *Select(Func*<*T*, *TResult*>), which is a projection that extracts only the *Name* of each product:

```
var productNames = products.Select(prod => prod.Name);
```

In the preceding example, *Func*<*T*, *TResult*> is inferred because the lambda conforms to its signature. To break it down a little further, the following example is logically equivalent to the *Select* just shown:


```
Func<Product, string> prodNameFunc = prod => prod.Name;
productNames = products.Select(prodNameFunc);
```

In the first line preceding, *prodNameFunc* is of type *Func<T, TResult>*, where *T* is *Product* and *TResult* is *string*. This means that the delegate takes one input parameter of type *Product* and returns a value of type *string*. You can see these types in action via the lambda assigned to *prodNameFunc*. The lambda parameter is type *Product*. Since *Product* has a *Name* property, the lambda body contains the *prod.Name* expression. The return type is inferred by the result of the lambda expression. Therefore, since the *Name* property of *Product* is type *string*, *prod.Name* results in a value of type *string*, which is the inferred return type. Since *prodNameFunc*'s type, *Func<Product, string>*, has a return type of *string*, the result matches the *Func<Product, string>* delegate signature. So, now you have a lambda assigned to a delegate.

The second statement preceding invokes *Select* with *prodNameFunc*. Since *Select* takes a parameter of type *Func<T, TResult>*, *prodNameFunc* matches because it is the constructed type, *Func<Product, string>*, of *Func<T, TResult>*. The first *Select* example accepted the lambda argument directly, but this second example assigned the lambda to a delegate and then passed the delegate as an argument to *Select*. They both perform the same task, but the second example was more explicit so you could see what is really happening.

The next example demonstrates how to implement a method that uses the *Func<T1, T2, TResult>* delegate. This example differs from the previous in that two parameters are passed to the lambda. We'll use an overload of the previous example: one parameter will be set with the 0-based index of the current collection item, and the other will contain the collection item. The result is an anonymous type with the item number, incremented by one, and the *Product Name* and *Price*:

```
var numberedProducts =
    products
        .Select(
            (prod, index) =>
                new
                {
                    Row = index+1,
                    Name = prod.Name,
                    Price = prod.ListPrice
                }
        );
```

The parameters in the preceding example are a comma-separated list, enclosed in parentheses. Parameter types are inferred: *prod* is type *Product*, inferred from *products* by *Select*, and *index* is type *int*, inferred from the second parameter type of the current *Select* overload. The specific delegate for the *Select* method is as follows:

```
Func<TSource, int, TResult>
```

In the preceding *Func* delegate, *TSource* is type *Product*, *int* is self-explanatory, and *TResult* is the anonymous type returned by the lambda.

The lambda body instantiates an anonymous type, which is also the return type. Since *index* is 0-based, the *Row* property of each anonymous type instance value increments by 1, allowing the numbering of the first anonymous type instance to start at row 1. *Name* and *Price* properties contain the values from the *Product* instance, *prod*, passed to the lambda.

By now, you should have a better idea of how lambda expressions can be used in a variety of scenarios. They simply can be used anywhere a delegate handler is required, have simplified syntax, and can be converted to expression trees. This last capability, converting lambdas to expression trees, is the subject of the next section.

Building Expression Trees

An expression tree is an in-memory data representation of a lambda expression. The key difference between an expression tree and a lambda expression is that an expression tree is data, but a lambda expression is executable code. Expression trees and lambda expressions are convertible to and from one another. In the following sections, you'll learn more about the relationship between expression trees and lambdas and about details of expression tree contents.

Expression Tree and Lambda Expression Conversions

It is significant that lambda expressions can be converted to expression trees. LINQ depends on being able to take query expressions, convert them to lambdas passed as parameters to extension methods, and finally to convert the lambdas to expression trees. With expression trees, a LINQ provider can build queries in a form that a data source expects. For example, SQL Server expects TSQL, web queries expect special URL parameters, and other providers have their own unique query syntax. In [Chapter 9](#), you'll see how the information presented here on expression trees works with extension methods, and [Chapter 10](#) will show you how it all fits together to build a custom LINQ provider. For right now, we have to build up to what is coming, which means that you need to learn how lambda expressions convert to expression trees.

To convert a lambda to an expression, just assign the lambda to an *Expression<TDelegate>* variable. The *Expression* class is a member of the *System.Linq.Expressions* namespace, meaning that you will need to add a *using* declaration to the top of your file. The following example shows how this works by assigning a lambda that concatenates strings to an *Expression<TDelegate>* variable, where *TDelegate* is the *Func<string, string>* constructed type:

```
Expression<Func<string, string>> greetTree =  
    name => "Hello, " + name;
```

The just shown *Expression<Func<string, string>>* variable, *greetTree*, now holds the data representation for the lambda that was assigned to it. In the next section, you'll see how to extract the pieces of *greetTree*, showing that it really is a tree of data. For now, we'll stay on track with looking at how conversions between lambdas and expression trees work.

The previous example showed how to convert a lambda to an expression tree, which is useful for LINQ providers that need to create data source-specific queries. Additionally, you can convert an expression tree to a lambda, which can be used for dynamically building executable expressions. The next example will take the *greetTree* expression tree, created in the previous example, and convert it to a lambda. It will call the *compile* method on the expression tree to produce a lambda:

```
Func<string, string> greetLambda =  
    greetTree.Compile();
```

The *Compile* method of *Expression<TDelegate>* infers the return type, which is the *TDelegate* type constructed when the *greetTree* was declared. Therefore, *Compile* returns a strongly typed instance of a

Func<string, string>, and the statement just shown assigns the results to *greetLambda*. If you recall, the original lambda concatenated the “Hello” string and the parameter name passed in. If you were to execute the lambda, the result would be the concatenated string, as follows:

```
string greetingResult = greetLambda(“Joe”);  
Console.WriteLine(greetingResult);
```

Passing the string “Joe” to the preceding *greetLambda* produces a string with the concatenated results, as defined by the original lambda body. Here’s the result from calling *Console.WriteLine*:

```
Hello, Joe
```

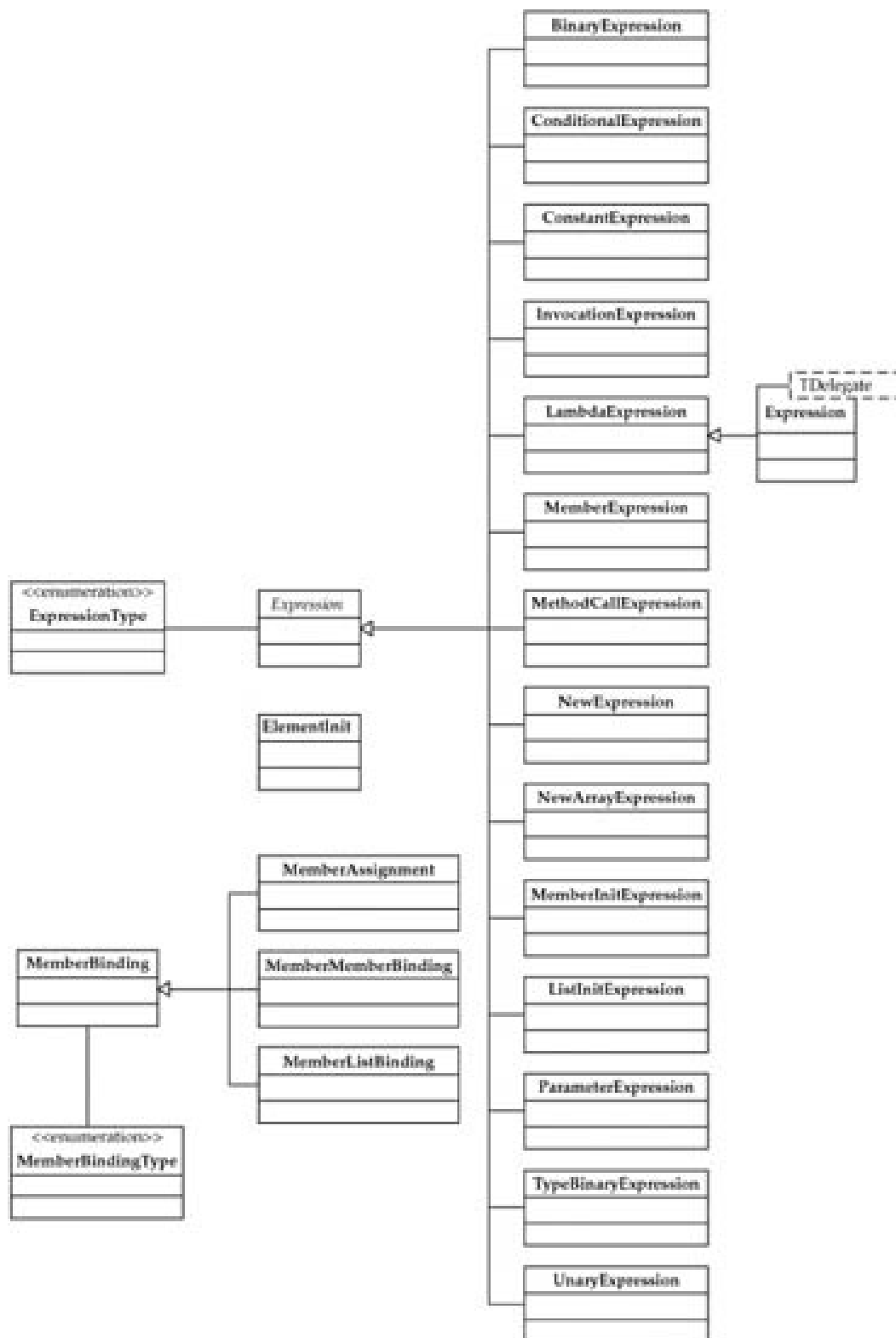
This demonstrates that the result of calling *Compile* on an *Expression<TDelegate>* is executable code in the form of a lambda. That was the lambda side of the story, but I’ve promised to show you the inside of the expression tree, which is coming up next.

Expression Tree Internals

This section digs deeper into answering the questions of just what an expression tree is. In the previous section, I defined it as the data representation of a lambda expression. Now, you will learn what the data is within an expression tree, how an expression tree is organized, and how to extract information from the expression tree. To show you expression tree internals, I’ll first introduce you to all of the types in the *System.Linq.Expressions* namespace so you will have an idea of what types are available or can be created. You’ll then see a sample lambda expression and examine its parts, helping you make the association between the lambda and the data in the expression tree. Finally, you’ll see how to extract each part of the expression tree and learn how to examine each expression for critical information that can be used to translate the expression tree to other query forms. First, let’s take a look at expression types, which are members of the *System.Linq.Expressions* namespace.

Understanding Expression Types

Expression trees are made of types in the *System.Linq.Expressions* namespace. Most of the time, parts of the expression tree are an *Expression* class derived type. The *System.Linq.Expressions* namespace also has classes for type initialization and supporting enums. [Figure 8-1](#) shows all of the available types in the *System.Linq.Expressions* namespace.



MemberBinding derived type.

The *Expression*, *MemberBinding*, and enum types shown in [Figure 8-1](#) are used to hold expression tree data, and you'll see some of their implementation soon. The next section will introduce the lambda that the example will be based upon.

Examining Parts of a Lambda Expression

To get a feeling for what is being held in an expression tree, it is helpful to visualize the lambda being used. We'll start off by examining something familiar—the expression tree created from the previous section, based on the lambda that concatenates strings to form a greeting. For your convenience, I've repeated it next:

```
Expression<Func<string, string>> greetTree =  
    name => "Hello, " + name;
```

Pay particular attention to the structure of the preceding lambda. It has a parameter, *name*, and a body, *"Hello, " + name*. Examining further, the body is a single statement that can be divided into three parts. The *"Hello, "* part is a literal string, which is a constant. The *name* part is the parameter variable that was passed in, and the *+* is a binary operator that works with the constant string and variable. We'll drill down into *greetTree* so we can see the data for each part of the lambda expression.

Extracting Lambda Data from an Expression Tree

As shown in [Figure 8-1](#), *Expression<TDelegate>*, the type of *greetTree*, derives from *LambdaExpression* which derives from *Expression* (abstract and non-generic). The *Expression*, class has two methods for accessing parts of an expression: *Parameters* and *Body*. The *Parameters* property is a collection of parameters belonging to the lambda, and the *Body* is an *Expression* derived type, representing the contents of the lambda body. Now, you might be able to draw associations between the description of the lambda in the previous paragraph and the properties of the *Expression* class. The following example shows how to use the *Parameters* and *Body* properties of the *Expression*, *greetTree*:

```
ParameterExpression nameParam =  
    greetTree.Parameters[0] as ParameterExpression;
```

```
BinaryExpression concat =  
    greetTree.Body as BinaryExpression;
```

The *Parameters* property is a collection and can have 0 or more members. In the current case, there is only one parameter, which is why the preceding example indexes into the first, index 0, element of *Parameters*. The lambda body is a *BinaryExpression* because it uses the concatenation operator to concatenate two strings. The exact *Expression* derived type of the body depends on the expression in the body and on the order of operations. Since the compile-time type of the *Parameters* and *Body* properties is *Expression*, you'll need to perform the conversion on each to work with them.

You can also break down the *Body* to get to parts of the expression, which is necessary for a LINQ provider to be able to examine the lambda expression. A *BinaryExpression* has *Left* and *Right* properties, which are both *Expression* types. The following example shows you how to use the *Left* and *Right* properties to view sub-expressions of the lambda body:

```
ConstantExpression helloString =  
    concat.Left as ConstantExpression;
```

```
ParameterExpression nameBody =  
    concat.Right as ParameterExpression;
```

If you recall from the previous example, which extracted *Body* from *greetTree*, the body was named *concat*. The preceding example uses *concat* to get the *Left* and *Right* expressions of the body. The *Left* expression is a literal string, which is a *ConstantExpression* in the expression tree. Since the *Right* expression is the lambda parameter, *name*, it is a *ParameterExpression*. Remember, all of the expressions evaluated are *Expression* derived types.

It's possible that you could have a much more complex expression tree, based on a more complex lambda expression. In that case, either the *Left* or *Right* expressions could be *BinaryExpression* types, allowing you to traverse the tree. For example, a depth-first recursive algorithm might be helpful. At this time, knowledge of compilers and interpreters would be beneficial, but beyond the scope of this book.

When you need to examine the nodes being visited, you can examine additional *Expression* properties. In many cases, each of the *Expression* derived types will have unique properties for their own contents. For example, *ConstantExpression* has a *Value* property, which would return “*Hello,* ” and *ParameterExpression* has a *Name* property that would return *name*. Of course, given some arbitrarily chosen lambda expression, you might not know what the type is. Fortunately, the *Expression* class has a *NodeType* property of type *ExpressionType* that you can examine to figure out what you're working with. Here's a demonstration of the *Expression* properties that you can use to find out more information about the expression nodes you're extracting:

```
new List<Expression>  
{  
    greetTree,  
    nameParam,  
    concat,  
    helloString,  
    nameBody  
}  
.ForEach(  
    expr =>  
        Console.WriteLine(  
            “\nNode Type: ” + expr.NodeType +  
            “\nClass Type: ” + expr.GetType() +  
            “\nExpr Type: ” + expr.Type +  
            “\nInfo:      ” + expr  
        )  
);
```

The preceding list includes all of the *Expression* nodes extracted from previous examples. The type information is printed and here is the output:

```
Node Type: Lambda  
Class Type: System.Linq.Expressions.Expression`1  
           [System.Func`2[System.String,System.String]]  
Expr Type: System.Func`2[System.String,System.String]  
Info:      name => (“Hello, ” + name)  
  
Node Type: Parameter  
Class Type: System.Linq.Expressions.ParameterExpression  
Expr Type: System.String  
Info:      name
```

Node Type: Add
Class Type: System.Linq.Expressions.BinaryExpression
Expr Type: System.String
Info: ("Hello, " + name)

Node Type: Constant
Class Type: System.Linq.Expressions.ConstantExpression
Expr Type: System.String
Info: "Hello, "

Node Type: Parameter
Class Type: System.Linq.Expressions.ParameterExpression
Expr Type: System.String
Info: name

Notice how the *Node Type* displays the *ExpressionType* enum value. The *Add* is the + symbol, and you'll have to note that it is a binary expression for two strings to formulate the proper semantics for whatever LINQ provider translation you're doing.

Now, you know how to create expression trees from lambdas. Additionally, you can build expression trees from scratch, which is covered in the next section.

Dynamic LINQ Queries with Expression Trees

Most of the LINQ providers offer a plethora of options, enabling you to accomplish nearly any task you need. However, you might have a need to develop additional functionality that isn't already covered by existing operators. The example in this section shows you how to fill that gap if you ever encounter it. In addition to creating applications in LINQ, you might come up with a potential application of expression trees in another scenario. Understanding how to build an expression tree might help you creatively solve a problem where you need to dynamically generate code.

The following sections help you learn how to build expression trees by describing a potential problem to solve, explaining how to build the expression tree, and then showing how to convert the expression tree to a lambda and to invoke the lambda in a LINQ to SQL query.

The Where Or Problem

A typical search function will use the *Where* operator to dynamically build a query. We haven't looked at extension methods or operators much in previous chapters, but I'll give you a glimpse into how you can use the *Where* operator. [Chapter 9](#) discusses extension methods in depth, and the Appendix lists all of the available operators. In a search scenario, you'll typically allow the user to enter search terms in a text box in your user interface (UI). Some event handler, normally a button *Click* event handler, grabs the search terms, processes them into a list, and passes that list to your business object that performs the search operation. For the sake of brevity, I'll assume that you know how to get those search terms from the UI and to pass them to the business object. The example here will detail the behavior of that business object, which demonstrates the problem to be solved.

The problem that you will see is based on the fact that the *Where* operator composes queries with AND conditions. In other words, if you combine multiple *Where* operators, each condition will be combined with AND logic. For example, if your search terms are "bicycle," "frame," and "red," the query will result in "bicycle AND frame AND red," meaning that all conditions must be satisfied.

However, what if you want the term to be “bicycle OR frame OR red,” where any one of the conditions would result in a match? You need something more than just the *Where* operator. This is what I call the *Where Or* problem.

First, I’ll show you how to compose the query with *Where* operators. You’ll need to add a LINQ to SQL project item to your project for the AdventureWorksLT database. If you need a refresher on how to do this, [Chapter 3](#) provides detailed instructions that explain how to set up a project to use LINQ to SQL. We’re primarily interested in the *Product* table. For the following example, you’ll need to ensure you have a *using* declaration for the *System.Data.Linq.SqlClient* namespace so you can use the *SqlMethods* class for a wildcard comparison via the *Like* method. Assuming you’ve set up the LINQ to SQL item in your project and it contains the *Product* table, the following code demonstrates composing queries with the *Where* operator:

```
var searchTerms =
    new List<string>
    {
        "bicycle",
        "frame",
        "red"
    };

var ctx = new AdventureWorksDataContext();

ctx.Log = Console.Out;

var products =
    from prod in ctx.Products
    select prod.Name;

foreach (var term in searchTerms)
{
    var currTerm = term;
    products =
        products.Where(
            prodName =>
                SqlMethods.Like(
                    prodName,
                    "%" + currTerm + "%")
        );
}

products.ToList();
```

The *searchTerms* collection simulates a list of search terms that might have been retrieved from a UI text box and parsed to individual terms. Since we’re using LINQ to SQL, you need a *DataContext*, and you’ll be able to see the output of the query via the console. The *products* is a deferred query that projects only on the *Name* property. Remember that deferred queries don’t execute until you execute a *foreach* or invoke an operator that forces execution of the query. Also, because it is a deferred query, *currTerm* ensures that each iteration of the loop adds a unique value in the *Like* clause, which can be tricky if you aren’t expecting this behavior.

The *foreach* loop demonstrates how to build upon the *products* query, adding *where* clauses for each search term. We want to know if the search term exists in the product name, so the example uses the *Like* method of the *SqlMethods* class, which performs a wildcard comparison of the search term to the *Name*.

Calling *ToList* materializes the query, resulting in the following console output:

```
SELECT [t0].[Name]
FROM [SalesLT].[Product] AS [t0]
WHERE ([t0].[Name] LIKE @p0) AND
      ([t0].[Name] LIKE @p1) AND
      ([t0].[Name] LIKE @p2)
-- @p0: Input NVarChar (Size = 5; Prec = 0; Scale = 0) [%red%]
-- @p1: Input NVarChar (Size = 7; Prec = 0; Scale = 0) [%frame%]
-- @p2: Input NVarChar (Size = 9; Prec = 0; Scale = 0) [%bicycle%]
-- Context: SqlProvider(Sql2005)
-- Model: AttributedMetaModel
-- Build: 3.5.30729.1
```

I’ve formatted the preceding output so it will fit into the book better, but the main point of this output is the *where* clause. Notice how each of the parameters—*red*, *frame*, and *bicycle*—is ANDed together. This is not the behavior we want, so the next section shows you how to fix the problem.

A Dynamic Query Building Algorithm

To build a dynamic query, you’ll need to instantiate and combine a set of expressions into a tree. The results will be a lambda, so you must combine the tree for the body with the parameter list into a final expression—proper form for a lambda. The algorithm in [Listing 8-1](#) takes a set of search terms for the *Name* property of the *Product* table and builds *where* clauses that OR each search term together.

Listing 8-1 A Method for Dynamically Building an Expression Tree

```
public static Expression<Func<TSearchObject, bool>>
    CreateOrExpression<TSearchObject>(
        string propertyName,
        List<string> searchTerms)
{
    ParameterExpression searchTermParam =
        Expression.Parameter(
            typeof(TSearchObject),
            "searchTerm");
    MemberExpression searchTermMember =
        LambdaExpression.PropertyOrField(
            searchTermParam,
            propertyName);

    BinaryExpression orExpression = null;
    MethodCallExpression leftExpr = null;

    foreach (var searchTerm in searchTerms)
    {
        var wildSearchTerm =
            "%" + searchTerm + "%";

        if (leftExpr == null) // first expression
        {
            leftExpr =
                Expression.Call(
                    typeof(SqlMethods)
                        .GetMethod(
                            "Like",
                            new Type[] {
```

```

        typeof(string),
        typeof(string) })),
        searchTermMember,
        Expression.Constant(
            wildSearchTerm,
            typeof(string)));
    }
    else if (orExpression == null) // second expression
    {
        MethodCallExpression rightExpr =
            Expression.Call(
                typeof(SqlMethods)
                .GetMethod(
                    "Like",
                    new Type[] {
                        typeof(string),
                        typeof(string) })),
                searchTermMember,
                Expression.Constant(
                    wildSearchTerm,
                    typeof(string)));
        orExpression =
            Expression.Or(leftExpr, rightExpr);
    }
    else // third and subsequent expressions
    {
        BinaryExpression tempExpr = orExpression;

        MethodCallExpression rightExpr =
            Expression.Call(
                typeof(SqlMethods)
                .GetMethod(
                    "Like",
                    new Type[] {
                        typeof(string),
                        typeof(string) })),
                searchTermMember,
                Expression.Constant(
                    wildSearchTerm,
                    typeof(string)));
        orExpression =
            Expression.Or(tempExpr, rightExpr);
    }
}

Expression<Func<TSearchObject, bool>> lambdaExpr;

if (orExpression == null)
{
    // only one expression to evaluate
    lambdaExpr =
        Expression.Lambda<Func<TSearchObject, bool>>(
            leftExpr,
            searchTermParam);
}
else
{
    // two or more expressions
    lambdaExpr =
        Expression.Lambda<Func<TSearchObject, bool>>(

```

```

        orExpression,
        searchTermParam);
    }

    return lambdaExpr;
}

```

[Listing 8-1](#) is quite long, so I'll break it down into pieces so you can see how it works. The method declaration takes a property name string and a list of search terms and returns an *Expression<TDelegate>*, shown next:

```

public static Expression<Func<TSearchObject, bool>>
    CreateOrExpression<TSearchObject>(
        string propertyName,
        List<string> searchTerms)

```

The list of search terms is required because those are the terms that will be compared. The property name specifies the property of the object whose type is *TSearchObject* to compare each search term with. So, if *TSearchObject* is *Product*, property is *Name*, and *searchTerms* contains *bicycle*, *frame*, and *red*, then the resulting query should be similar to `select Name from Product as p where p.Name like '%bicycle%' OR p.Name like '%frame%' OR p.Name like '%red%'`. You'll see output similar to this in the next section, but you need to understand how the algorithm works first.

The first problem to solve in [Listing 8-1](#) is how to access the *Name* property of each *Product* instance. The solution should be strongly typed and use expressions. Therefore, the proper approach is to create an expression for the *Product* type and then build an expression for *Product.Name*, as shown next:

```

ParameterExpression searchTermParam =
    Expression.Parameter(
        typeof(TSearchObject),
        "searchTerm");
MemberExpression searchTermMember =
    LambdaExpression.PropertyOrField(
        searchTermParam,
        propertyName);

```

In the lambda, *Product* will be represented by the parameter, which we've coded as *searchTerm*. As you learned in the previous section, a parameter is a *ParameterExpression* in the expression tree, which you can create by invoking the *Parameter* method of the *Expression* class and passing the type, *Product*, and parameter name, *searchTerm*. *Expression* has many static methods that you can use to build expression trees, and you'll see a few of them used in [Listing 8-1](#). The *searchTermParam* is a variable for holding the lambda parameter.

Next, you need to access the *Name* property of the object, which is represented as a *MemberExpression*, since properties are class members. This time, we use the static *PropertyOrField* method of *LambdaExpression* to define property access, passing in the parameter and property name, *Name*, via the *propertyName* parameter that was passed to the *CreateOrExpression* method.

Notice that *orExpression* and *leftExpression* are *null*, as shown next:

```

BinaryExpression orExpression = null;
MethodCallExpression leftExpr = null;

```

This algorithm will use the fact that *leftExpr* is *null* to know that it hasn't processed any *searchTerms* yet. When *leftExpr* is not *null* and *orExpression* is still *null*, then one search term has been processed, but not two. If neither *leftExpr* nor *orExpression* is *null*, then three or more search terms are being processed. Here's the logic for figuring this out:

```
if (leftExpr == null) // first expression
{
    ...
}
else if (orExpression == null) // second expression
{
    ...
}
else // third and subsequent expressions
{
    ...
}
```

These checks are necessary because we are building an expression tree and need to know how to connect the nodes. When you haven't processed any nodes, *leftExpr* is *null*; you build the OR expression without attaching it to an expression tree. In other words, the one expression is the tree. Here's how to handle a single expression:

```
leftExpr =
    Expression.Call(
        typeof(SqlMethods)
        .GetMethod(
            "Like",
            new Type[] {
                typeof(string),
                typeof(string) }),
        searchTermMember,
        Expression.Constant(
            wildSearchTerm,
            typeof(string)));
```

Invoking *Expression.Call* creates an expression that invokes a method. *Call* takes a *MethodInfo*, which can be obtained via reflection. You can see the reflection call through calling *GetMethod* on the *SqlMethods* type. The method name is *Like* and it takes two parameters of type *string*. The first parameter is *searchTermMember*. If you recall from earlier in this walkthrough that the first thing the algorithm does is create a *MemberExpression* called *searchTermMember*, that allows us to reference *Product.Name*. The next parameter is the *searchTerm*, surrounded by SQL wildcard characters. The preceding example assigns the results to *leftExpr*, which will be used as the left side of a binary expression in the expression tree if two or more expressions exist.

If there is a second search term, *leftExpr* is not *null*, but *orExpression* is. Here's the logic for handling the second search term:

```
MethodCallExpression rightExpr =
    Expression.Call(
        typeof(SqlMethods)
        .GetMethod(
            "Like",
            new Type[] {
                typeof(string),
```

```

        typeof(string) }),
        searchTermMember,
        Expression.Constant(
            wildSearchTerm,
            typeof(string)));
orExpression =
    Expression.Or(leftExpr, rightExpr);

```

As you can see, the logic for the second search term is nearly identical to the first except that the result is assigned to a variable named *rightExpr*. After the method call is generated, *leftExpr*, which holds the first expression, and the new *rightExpr* are combined in an OR operation, via *Expression.Or*, and assigned to *orExpression*. Now, both *leftExpr* and *orExpression* are not null, and *orExpression* is a *BinaryExpression* with *Left* and *Right* properties set to OR expressions on two separate search terms. All other search terms are processed via the *else* clause from [Listing 8-1](#) and repeated next:

```
BinaryExpression tempExpr = orExpression;
```

```

MethodCallExpression rightExpr =
    Expression.Call(
        typeof(SqlMethods)
        .GetMethod(
            "Like",
            new Type[] {
                typeof(string),
                typeof(string) }),
        searchTermMember,
        Expression.Constant(
            wildSearchTerm,
            typeof(string)));
orExpression =
    Expression.Or(tempExpr, rightExpr);

```

Nearly all of the code in this case is the same as the second search term processing, except for the new *BinaryExpression*, *tempExpr*. We need to save the previous *orExpression* into *tempExpr* to continue building the tree. Notice the last statement just shown, which instantiates a new *BinaryExpression*, assigning it to *orExpression*. It assigns *tempExpr*, which was the previous object that *orExpression* referred to, as the left expression. This builds an expression tree where each subsequent search term is added to the left side of the tree.

When there aren't any more search terms, the *foreach* loop ends and the body of the lambda is complete. All that's left is to combine the parameters with the body and return the complete expression tree, as a *LambdaExpression*, to the caller. Here's the code from [Listing 8-1](#) that performs this task:

```

Expression<Func<TSearchObject, bool>> lambdaExpr;

if (orExpression == null)
{
    // only one expression to evaluate
    lambdaExpr =
        Expression.Lambda<Func<TSearchObject, bool>>(
            leftExpr,
            searchTermParam);
}
else
{
    // two or more expressions

```

```

lambdaExpr =
Expression.Lambda<Func<TSearchObject, bool>>(
    orExpression,
    searchTermParam);
}

```

The *lambdaExpr* variable is an *Expression<TDelegate>* that matches the return type of the method. You create a *LambdaExpression* by calling the static *Lambda* method of the *Expression* class, similar to what we did with *Expression.Parameter* and *Expression.Or*. Continuing with the previous logic, if *orExpression* is *null*, then only one search term was processed, and the expression is assigned to *leftExpr*, which becomes the body for *lambdaExpr*. When *orExpression* is not *null*, it will become the body for *lambdaExpr*. The lambda parameter is *searchTermParam*, which was created at the beginning of the algorithm to define the name and type of the lambda parameter. As you saw earlier, *searchTermParam* also serves to build the *MemberParameter* used in the lambda body. You now have an expression tree. [Figure 8-2](#) illustrates the structure and relationship of expressions in the new expression tree.

The expression tree in [Figure 8-2](#) shows each expression that matches the parts of a lambda, including parameter and body. Once the expression tree has been created, the algorithm returns *lambdaExpr* to the caller, who now has a *where* clause with ORs, which is discussed in more depth in the next section.

Executing the Dynamic Query

Executing the *LambdaExpression* is trivial because the *Where* operator takes *Expression<TDelegate>* as a parameter. Earlier, you saw how we passed a lambda expression to *Where*, but you also know that lambda expressions are implicitly convertible to expression trees. Therefore, passing an expression tree to *Where* is no problem because that's the type of the *Where* parameter.

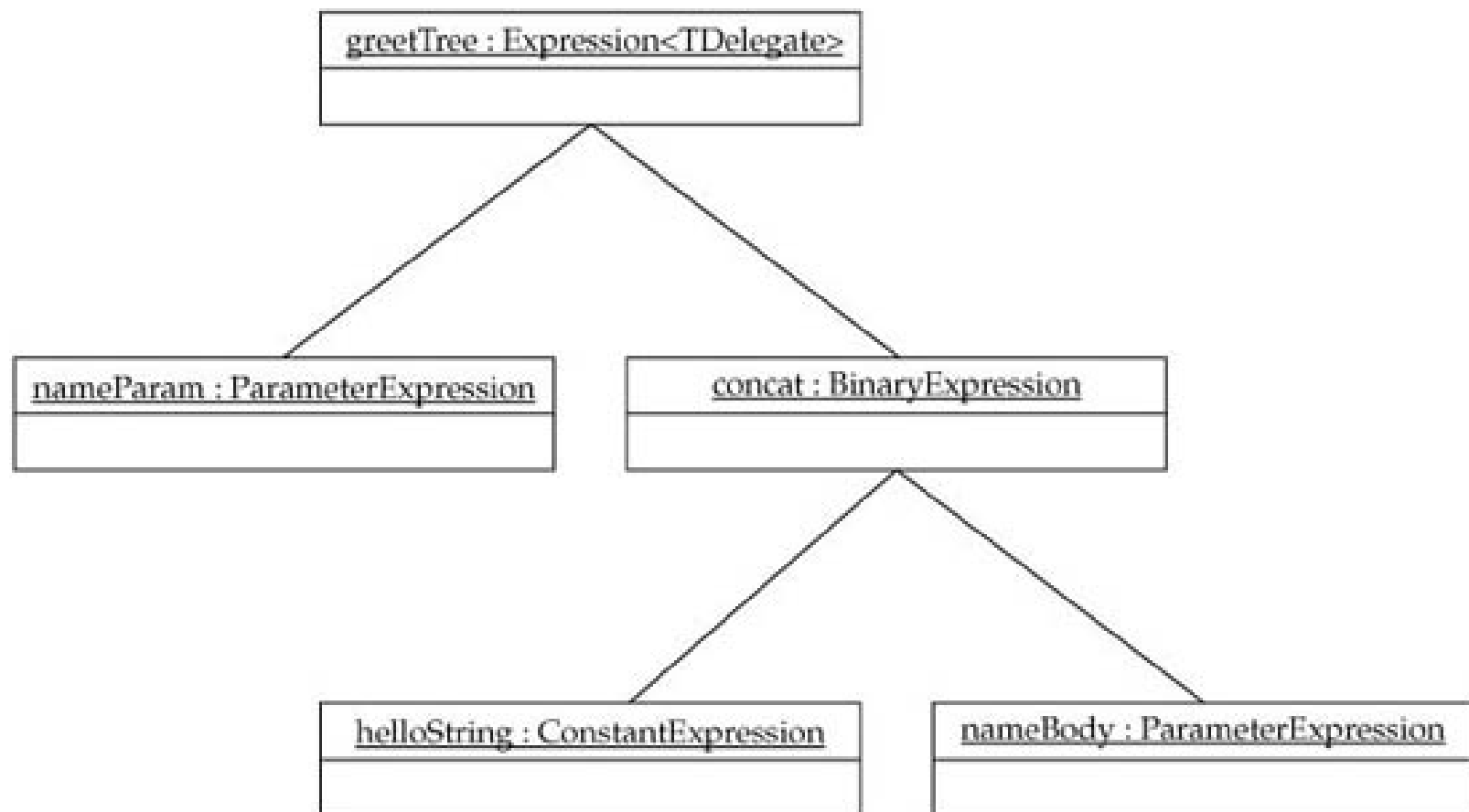


FIGURE 8-2 Object diagram of an expression tree

The following example is nearly the same as the previous example that specified the *Where Or* problem, except that it solves the problem by calling the *CreateOrExpression* method from the previous section, rather than composing *Where* queries:

```
var searchTerms =
    new List<string>
    {
        "bicycle",
        "frame",
        "red"
    };

var ctx = new AdventureWorksDataContext();

ctx.Log = Console.Out;

var products =
    from prod in ctx.Products
    select prod;

var lambdaExpr =
    CreateOrExpression<Chapter_08.Product>(
        "Name", searchTerms);

var prodNames =
    products
    .Where(lambdaExpr)
    .Select(prod => prod.Name);

prodNames.ToList();
```

The *searchTerms*, *DataContext*, and basic search are the same as for the *Where Or* problem statement. What's different is the call to *CreateOrExpression*. The type parameter identifies the *Product* table, the *Name* string parameter specifies the column to query, and the *searchTerm* list is the terms to search for. Because you can specify the entity as a type parameter and the property as a method parameter, you can reuse this method on any LINQ to SQL query.

The result is an *Expression<Func<Product, bool>>*, which is subsequently passed as a parameter to a single *Where* operator. I've used the *Select* method to limit the output to the *Name* property only. You now have a query that ORs terms in the *where* clause, instead of using AND. Here's the *DataContext Log* output:

```
SELECT [t0].[Name]
FROM [SalesLT].[Product] AS [t0]
WHERE ([t0].[Name] LIKE @p0) OR
      ([t0].[Name] LIKE @p1) OR
      ([t0].[Name] LIKE @p2)

-- @p0: Input NVarChar (Size = 9;
      Prec = 0;
      Scale = 0) [%bicycle%]
-- @p1: Input NVarChar (Size = 7;
      Prec = 0;
      Scale = 0) [%frame%]
-- @p2: Input NVarChar (Size = 5; Prec = 0;
      Scale = 0) [%red%]
-- Context: SqlProvider(Sql2005)
```

The preceding clause uses the OR condition, just as we need, producing the results we want.

Summary

You've learned several things about lambda expressions and expression trees. You've seen additional examples of how lambdas can be used, beyond what you saw in [Chapter 1](#). You saw how to use lambdas with a set of *Func* delegates that are commonly used in LINQ.

The discussion of *Func* delegates was important because you use them to build expression trees. You learned how lambdas and expression trees are convertible from one to another. Once you've converted a lambda to an expression tree, you can extract the parts of the expression tree, which are data that can be analyzed. You also saw how to build an expression tree from the ground up, enabling creation of dynamic queries.

All of this was very interesting to look at and perhaps to find new ways to do things, such as dynamic queries, which would be harder or less desirable to do via other means. However, the immediate significance of understanding lambdas and expression trees is in knowing how LINQ works and being more able to solve any problem with advanced knowledge. Now, you're prepared for the next step in understanding how LINQ works and leading you closer to being able to build your own LINQ provider. The next chapter moves you further along this path with a look at extension methods.

CHAPTER 9

Constructing New Code with Extension Methods

As you learned in [Chapter 1](#), extension methods allow you to extend a type through new methods that can be invoked on that type’s instance. While this is a new feature of C#, VB.NET, and potentially other .NET programming languages, extension methods are integral to making LINQ work. Moving forward, you’ll learn how query expressions translate into LINQ extension methods. You’ll also learn how to create your own extension method, seeing the internal workings of how they can be used to build a LINQ provider of your own.

Relating Query Expressions to Extension Methods

Query expressions are syntactic “sugar” disguising their true nature—extension methods. Of course, we want this syntactic sugar because it makes us more productive to have strongly typed clauses that are easy to understand and that are built into the language. Sometimes query expressions are the most desirable way to use LINQ; other times extension methods are easier to use. In the previous chapter, you saw how to use the *Where* method to dynamically build a query, demonstrating how to handle a search function. This dynamic query building capability illustrates an excellent reason to use extension methods instead of query expressions. In the following sections, you’ll learn how query expressions translate into extension methods, which is instrumental toward creating a LINQ provider, in addition to learning about a useful group of extension methods that you’ll want to use.

Matching Extension Methods

Each of the C# query expression clauses has equivalent extension methods that perform the same task. Some of the extension methods have overloads with no equivalent in query syntax. For example, there is a *Select* method overload that passes the current row number to the lambda expression. [Table 9-1](#) shows the C# query expression clauses and matching extension methods.

The extension methods in [Table 9-1](#) are necessary for implementing a LINQ query provider. Clauses in C# query expressions are translated into these methods at compile time.

C# Query Expression Clause	Equivalent Extension Method
from x in entity from z in x.y	SelectMany
select	Select
where	Where
order by	OrderBy
group by	GroupBy
join	Join

TABLE 9-1 Extension Methods Matching Query Expression Clauses

How Query Expressions Translate

C# query expressions compile to extension methods for the LINQ provider identified in the data source to the *from* clause. To demonstrate this, I'll use a LINQ to SQL DataContext for the AdventureWorks database. If you are following along, you can review [Chapter 3](#) for a comprehensive set of instructions on how to create a LINQ to SQL DataContext for the AdventureWorks database. The following code demonstrates how you can view the equivalent extension methods of a query expression:

```
var ctx = new AdventureWorksDataContext();
```

```
var salesOrders =
    from order in ctx.SalesOrderHeaders
    from detail in order.SalesOrderDetails
    select
        new
        {
            detail.Product.Name,
            detail.UnitPrice,
            detail.OrderQty,
            detail.LineTotal
        };

```

```
Console.WriteLine(salesOrders.Expression);
```

As you know from earlier chapters, the nested *from* clauses form a *select many* clause, allowing you to flatten a hierarchical data source, and the *select* clause projects into an anonymous type. This shouldn't be anything new to you by now. What is new is how the preceding query translates into extension methods, which you can view via the *Expression* property of the *IQueryable<T>*, *salesOrders*. The following output demonstrates how query expressions translate into extension methods:

```
Table(SalesOrderHeader)
.SelectMany(
    order => order.SalesOrderDetails,
    (order, detail) =>
        new <>f__AnonymousType0`4(
            Name = detail.Product.Name,
            UnitPrice = detail.UnitPrice,
            OrderQty = detail.OrderQty,
            LineTotal = detail.LineTotal)
)

```

As you can see, the preceding query results in a call to the *SelectMany* extension method, which performs a projection on an anonymous type. You can use the same technique with other query expression clauses to verify the translation as defined in [Table 9-1](#).

Extension Method Composability

The preceding example was relatively simple in that it resulted in a single *SelectMany* call. Many LINQ queries aren't so simple, and you still need an easy way to write them. One way is to nest queries as parameters inside of each other. That works with LINQ to XML functional construction, but that's different and the relationship of clauses isn't necessarily hierarchical. Fortunately, LINQ queries are composable, where the results of one query can flow into another.

You can chain extension methods together, composing the result you need. Here’s an example that demonstrates extension method composability:

```
var ctx = new AdventureWorksDataContext();

ctx.Addresses
    .Where(
        add =>
            add.StateProvince == “Colorado”)
    .Select(
        coAdd =>
            new
            {
                coAdd.PostalCode,
                coAdd.StateProvince
            }
    )
    .ToList()
    .ForEach(
        coAddItem => Console.WriteLine(coAddItem)
    );
```

The preceding example demonstrates how to build an entire query in a single statement. First, it uses *Addresses* as the data source, filters on *Colorado* as the state, projects on *PostalCode* and *StateProvince*, materializes the results into a *List*, and iterates to print each item to the console. Here’s the equivalent query syntax:

```
(from add in ctx.Addresses
 where add.StateProvince == “Colorado”
 select
     new
     {
         add.PostalCode,
         add.StateProvince
     })
.ToList()
.ForEach(
    coAddItem => Console.WriteLine(coAddItem)
);
```

As I mentioned in the introduction to this section, sometimes you want to use extension methods, and other times you would rather use query expressions. The choice often falls into the realm of personal preference. Either approach works.

These were the extension methods that match query expression clauses, but many more extension methods don’t have query expression equivalents. The Appendix has a comprehensive list of all available extension methods, which you can use as a reference to learn about all of the operators available to you.

Building a Custom Extension Method

The C# 3.0 Language Specification states that “the first parameter of a method includes the *this* modifier...” to define what an *extension method* is. It is true that the *this* modifier is what distinguishes the extension method from any other type of method, but other factors also contribute to what can be a valid extension method, including static, extended type specification and allowable types to extend. The following sections will describe how to create a properly formed extension method.

If you had the source code to a method, you could either add a new method directly or, in the case of auto-generated code such as a LINQ to SQL entity, you could add a new method to a partial class. There are times though that you might like to extend a type that you don't have direct access to. Think of all the types in the .NET Framework Class Library, such as *string*, *File*, or *List<T>*, that don't offer the functionality that you wish you had. Now, you can add functionality to those types and more with extension methods.

The example in this section will extend *System.DateTime*. Think of all the times you've needed to ensure that a date was between two other dates. You can do this without too much trouble, but wouldn't it be nice to have a method that did this? Now, you can add a new method to the *DateTime* struct yourself with an extension method. [Listing 9-1](#) shows a custom extension method, named *Between*, that checks to see if one date is between two others.

Listing 9-1 A Custom Extension Method

```
using System;

public static class DateExtensions
{
    public static bool Between(
        this DateTime dateToCheck,
        DateTime earlierDate,
        DateTime laterDate)
    {
        return
            earlierDate <= dateToCheck &&
            dateToCheck < laterDate;
    }
}
```

The *Between* method, just shown, doesn't have any error checking or other code that you would write for a production-ready method, but it does demonstrate the bare guts of what you need to do to write a custom extension method. The parts of [Listing 9-1](#) to pay attention to are the class declaration and method declaration.

The *DateExtensions* class is *static* and non-generic. You'll receive the compiler message "Extension methods must be defined in a non-generic static class" if either of these conditions is not true.

The *Between* method is *static*. If you don't make it *static*, the compiler will give you an error message that states, "Extension methods must be static." If your class is already *static*, you'll also see an error message because you declared an instance member in a *static* class.

The first parameter, *dateToCheck*, has the *this* modifier. Forgetting to use the *this* modifier doesn't generate a compiler error for the *Between* method; it will be treated as just another *static* method. If you have tried to access the *Between* method as if it were an extension method and you forgot to add the *this* modifier to the first parameter, you'll receive a compiler message at the place you are trying to call the method.

The *dateToCheck* parameter, which is type *DateTime*, is a reference to the object instance that *Between* is being called on. Extension methods only work with instances, not on *static* types. Here is an example of how you would call *Between* on a *DateTime* instance:

```
var newYear = new DateTime(2010, 1, 1);
```

```
var earlierDate = new DateTime(2009, 12, 15);  
var laterDate = new DateTime(2010, 1, 15);  
  
var isBetween =  
    new Year.Between(earlierDate, laterDate);  
  
Console.WriteLine(  
    "Between Dec 15th and Jan 15th? " + isBetween);
```

The preceding *newYear* variable is a *DateTime* instance that invokes the *Between* extension method. It is the *newYear* instance that is passed as the first parameter, modified with *this*, to the *Between* method as *dateToCheck* in [Listing 9-1](#).

In practice, an extension method behaves and looks like a normal instance member. The example in [Listing 9-1](#) didn't include a namespace, but you'll need to remember to add the namespace for the extension method in order to use it—just as you would with any other C# type. While working in VS 2008, you can differentiate extension methods from normal instance methods because the extension methods have an icon with a purple down arrow.

Summary

Extension methods are convenient for extending types that you don't already have the code for. More importantly though, extension methods were added to C# to support LINQ. You saw how C# query syntax translates directly to a set of extension methods, demonstrating the magic that happens behind the scenes when your code compiles.

You also learned how to create a custom extension method. This example extended an existing type, but you can also write extension methods for a specific LINQ provider. The next chapter combines the knowledge from lambda expressions and expression trees in [Chapter 8](#) with extension methods in this chapter and shows you how to use interfaces in the System.Linq namespace to build a custom LINQ provider.

CHAPTER 10

Building a Custom LINQ Provider—Introducing LINQ to Twitter

While the new C# language features such as lambda expressions and extension methods are useful in many contexts of their own, the primary motivation for their existence is LINQ. The last couple of chapters covered lambda expressions, expression trees, and extension methods because these capabilities enable an open framework where anyone can build a LINQ provider.

Microsoft ships LINQ to Objects, LINQ to SQL, LINQ to XML, LINQ to Entities, and LINQ to Data Services with the .NET Framework. Additionally, they've made framework tools available for anyone to build a custom LINQ provider. Many third-party providers are already available: LINQ to NHibernate, LINQ to MySQL, and LINQ to Oracle to name a few. Now, you can add your data source, if you have one and would choose to do so, to the growing list of LINQ providers.

To build a custom LINQ provider, you'll need to envision how that provider is used, which shouldn't be a surprise because it uses common LINQ syntax. Before jumping into code, it helps to become familiar with the parts of the LINQ provider development process. After that, you can begin writing your provider code and plugging it into current .NET Framework interfaces and reusing code that is readily available. The following sections will take you through each of these steps, helping you learn how to create a custom LINQ provider yourself.

Introducing LINQ to Twitter

The examples in this chapter are based on a custom LINQ provider named LINQ to Twitter. LINQ to Twitter allows you to perform queries on the Twitter micro-blogging site, which you can find at <http://twitter.com/>. Essentially, Twitter is a free service that allows you to submit short messages that friends will be able to see. You can also see the messages that your friends submit. Twitter has a web service API that anyone can use if they would like to integrate Twitter into their own applications. LINQ to Twitter builds upon the Twitter API, bringing you the common LINQ programming experience that you have with all other LINQ providers. The code for this book will include everything you see here, but updates will also be available via CodePlex at www.codeplex.com/LinqToTwitter.

Before diving into the LINQ provider development process, I'd like to explain how LINQ to Twitter works. Of the many features of the Twitter API, LINQ to Twitter currently supports two types of queries: *public* and *friend*. A *public* query will return the last 20 messages from everyone on Twitter, and a *friend* query will return the last 20 messages from everyone in your friend list. You can perform a *public* query without needing a Twitter account, but the *friend* query requires a Twitter account. Twitter is free and you won't need to pay anything to create an account. You can still make public queries with LINQ to Twitter if you prefer not to create an account.

LINQ to Twitter resides in a DLL assembly named *LinqToTwitter.dll*. To use it, you can create a project that references the *LinqToTwitter.dll* assembly. Alternatively, you can add the *LinqToTwitter* class library project that is part of the [Chapter 10](#) code for this book to your solution, and then create a reference to the *LinqToTwitter* project. All of the *LinqToTwitter* code is part of the *LinqToTwitter* namespace, so you might want to add a *using* declaration to your code for the *LinqToTwitter* namespace. Once you have a reference to the *LinqToTwitter* assembly or project and a *using* declaration to the *LinqToTwitter* namespace, you'll be ready to make LINQ to Twitter queries.

Here's a public query using LINQ to Twitter:

```
var twitterCtx = new TwitterContext(userName, password);
```

```
var type = "Public";
```

```
var tweets =  
    from tweet in twitterCtx.Status  
    where tweet.Type == type  
    select tweet;
```

The *TwitterContext* has a couple features similar to the *DataContext* you use in LINQ to SQL, where it exposes the objects you can query and manages deferred execution. The two parameters to the *TwitterContext* constructor, *userName* and *password*, are optional for a *public* query. Specifying the query type is via a string, and the only two options supported are “Public” for *public* queries and “Friend” for *friend* queries.

The query itself should be no surprise by now. Again, it’s a benefit of LINQ that you can have a common way to query a data source. In the implementation of LINQ to Twitter for this book, the data source is the *Status* property of the *TwitterContext*. In future versions, available via CodePlex, you’ll see other *TwitterContext* data sources for additional Twitter API features. Right now, it’s important to keep the feature set simple, at a minimal level, so that you only need to see what is essential for creating a LINQ provider.

The results of the query are an *IQueryable<Status>*. The *IQueryable<T>* should be familiar because it is the same collection type you get from LINQ to SQL queries. What’s new is the *Status* class, shown here:

```
public class Status  
{  
    public string Type { get; set; }  
    public DateTime CreatedAt { get; set; }  
    public string ID { get; set; }  
    public string Text { get; set; }  
    public string Source { get; set; }  
    public bool Truncated { get; set; }  
    public string InReplyToStatusID { get; set; }  
    public string InReplyToUserID { get; set; }  
    public bool Favorited { get; set; }  
    public User User { get; set; }  
}
```

In the *Status* class, you can see the property for *Type*, which is used to specify the type of status to return, *friend* or *public*. The rest of the values are consistent with the Twitter API, including the *User* type, shown here:

```
public class User  
{  
    public string ID { get; set; }  
    public string Name { get; set; }  
    public string ScreenName { get; set; }  
    public string Location { get; set; }  
    public string Description { get; set; }  
    public string ProfileImageUrl { get; set; }  
    public string URL { get; set; }  
    public bool Protected { get; set; }  
    public int FollowersCount { get; set; }  
}
```

Every time you get a status message, you also receive user information.

Now that you know how to use LINQ to Twitter, we'll start building a custom LINQ provider with an overview of the process and the pieces you need to make it work.

Overview of the LINQ Provider Development Process

Just as with any development process, you'll have some type of feature identification, design, implementation, testing, and deployment. The previous section identified the *feature set*, aka requirements, which also define what will be tested. Deployment will be the user's decision, but that dictates that the solution will be deliverable as a DLL, promoting reuse. What's left is the design and implementation, and this section will describe considerations you can make during your own project design phase(s).

When designing a LINQ provider, you'll need to know about the .NET Framework interfaces that are available, the code that you need for implementing the interfaces, and the information you need from the user for accessing the data source. I'll also share additional information resources to help you gain a broader view of the LINQ provider landscape.

.NET Framework Interfaces

One of the great benefits of the .NET Framework Class Library (FCL) is the ability to reuse existing code and provide interfaces that promote reliable communication between types and that define common ways of doing things. This holds true with LINQ providers, which can use existing FCL interfaces and types, easing the amount of work you must do to create your own provider.

Two of the primary interfaces you can use in a LINQ provider include *IQueryable<T>* and *IQueryProvider*. You also have a *Queryable* class that operates on these interfaces. In a later section of this chapter, I'll go into depth on what these interfaces are and how to implement them. For planning purposes, you should know that you have these types available; they can help you avoid much work. [Table 10-1](#) lists the FCL types used in LINQ to Twitter.

FCL Type	Purpose
IQueryable<T>	Represents a queryable source of data, holding information about the type of an expression, a reference to an expression tree to evaluate, and a reference to a provider that executes the query
IOrderedQueryable<T>	Supports sorting operations
Queryable	Contains extension methods that operate on IQueryable<T>

TABLE 10-1 .NET Framework Class Library Types Supporting Query Providers

There isn't anything that requires you to use existing FCL types, as specified in [Table 10-1](#); you can design and use your own types. For example, LINQ to Objects uses the *Enumerable* class for extension methods, which operate on *IEnumerable<T>* derived types. Therefore, you could create your own extension method library that operates on your own interface types. The point is that there are already

FCL types that you can reuse for multiple scenarios.

LINQ to Twitter uses the FCL types. The actual data source is a web service API, which differs from the LINQ to SQL data source—SQL Server. This is indicative of the fact that the existing FCL types are flexible enough for a wide range of LINQ providers. The rest of the chapter will explain how to use these FCL types, in addition to other code, to implement LINQ to Twitter.

Implementation Types

For each of the interfaces in the previous section, you’ll need to supply implementations. In many cases, most of the implementation code is already available, freeing you to concentrate on the code that translates expression trees to the format you need, executing the request for data and returning the data to the calling query. I’ll identify pieces of the puzzle that already exist and help separate that from code that you must write.

Clearly, you will need to create classes that implement *IQueryable<T>* and *IQueryProvider*. However, LINQ to Twitter implements *IOrderedQueryable<T>*, which derives from *IQueryable<T>*, to provide sorting services if necessary.

In addition to the FCL types, several types are defined in the .NET Framework SDK (VS 2008) documentation for the LINQ to TerraServer sample provider that we’ll reuse. This will further reduce the amount of custom code that must be written for LINQ to Twitter. In a later section, I’ll talk more about the LINQ to TerraServer sample and other resources that allow you to reuse code and see different LINQ provider implementations.

The LINQ to Twitter provider contains several files. Some implement FCL interfaces, some reuse existing code from samples, and other code is written from scratch. [Table 10-2](#) lists all of the files in the *LinqToTwitter* project that comes with this book, outlining the source of the code and the purpose of each file. For the *Source* column, *Documentation* is from the LINQ to TerraServer walkthrough in the .NET Framework documentation, *FCL* is from the .NET Framework Class Library, and *Custom* is custom code for the LINQ to Twitter Implementation.

In addition to the files that are explicitly listed in [Table 10-2](#), LINQ to Twitter relies on the *Queryable* class, which is part of the FCL. *Queryable* already provides dozens of standard query operators (extension methods) that operate on *IQueryable<T>*. Therefore, I didn’t need to reinvent the wheel and do all of that work for LINQ to Twitter.

The big takeaway from this section is being able to identify what code you can reuse and which custom code you must write yourself. Writing a LINQ provider is already a sizable task, and figuring out how to save time can be a benefit.

Data Source Communication

Whenever you’re developing a LINQ provider, you need to consider how you will connect to the data source. Surveying the providers that ship with .NET:

- LINQ to Objects just references the *IEnumerable* collection.
- LINQ to DataSet just uses a *DataSet*.
- LINQ to SQL requires a connection string.

Filename	Source	Purpose

Evaluator.cs	Documentation	Allows partial evaluation of captured variables in query expressions.
ExpressionTreeHelper.cs	Documentation	Utility class for working with expression trees.
ExpressionTreeVisitor.cs	Documentation	Base class for traversing expression tree.
InnernessWhereFinder.cs	Documentation	Locates the <i>where</i> clause in the expression tree so we can extract filter information.
InvalidQueryException.cs	Documentation	Custom exception thrown when illogical conditions exist in expression tree.
ParameterFinder.cs	Custom/Documentation	Code that <i>extracts</i> where clause parameters from the expression tree. These parameters are used to make the Twitter API query.
Status.cs	Custom	<i>Status</i> , class, which holds information returned from Twitter. Shown in previous section on how to use LINQ to Twitter.
TwitterContext.cs	Custom/Documentation	Class that exposes much of LINQ to Twitter functionality to queries. Holds connection credentials, exposes properties for querying, and performs deferred execution.
TwitterQueryable.cs	Custom/ Documentation/FCL	<i>IOrderedQueryable<T></i> , derived class that holds references to expression tree and provider.
TwitterQueryProvider.cs	Custom/ Documentation/FCL	<i>IQueryProvider</i> , derived class that holds part of provider implementation. Delegates execution to <i>TwitterContext</i> .
TypeSystem.cs	Documentation	Reusable class for reflecting on expression tree and extracting underlying type.
User.cs	Custom	Member of <i>Status</i> , includes user information that is returned from a Twitter status request.

TABLE 10-2 LINQ to Twitter Files, Source of Code, and Purpose

- LINQ to XML needs a file or URL.
- LINQ to Entities needs a connection string.

You must specify where the data is coming from, but more involved is the criteria for establishing that connection. In the case of LINQ to SQL and LINQ to Entities, the connection string is more involved because you need to specify the server, database, credentials, and other database-specific information. Additionally, you don’t want to hard-code this information because it changes, meaning that configuration file support is a viable option to consider.

Taking all these data source specification issues into consideration for LINQ to Twitter, there are a few points to make. First, the location of the data source is already set because you must address a Representational State Transfer (REST) web service, based on the Twitter domain name. The base URL, <http://twitter.com/>, won’t change very often, but that isn’t absolute. What if Twitter decided to scale out and offer service over multiple URLs? You could argue whether this is realistic or not, but for the sake of generalizing the discussion and extrapolating the consequences to any potential query provider, it might be a plausible strategy to make this URL a configuration file entry. The LINQ to SQL and LINQ to Entities models using *Properties.Settings* might be a useful approach, because LINQ developers are already familiar with it. The LINQ to Twitter URL is hard-coded, but that’s only a simplification to make the examples easier for you to read, and a real implementation would keep this in the configuration file.

Other parts of the REST URL are more dynamic and would probably be better implemented in code. For example, the REST URL for public queries is http://twitter.com/statuses/public_timeline.xml. LINQ to Twitter’s current implementation only works with status, requiring the statuses part of the URL. However, later implementations will include the ability to send messages directly to friends; those implementations will use a URL of the form http://twitter.com/direct_messages.xml. Therefore, implementing segments of the URL beyond the base URL would probably be better handled in code.

Another consideration is credentials. Twitter requires username and password for all but *public* status queries. So, you need a way to pass those credentials into provider code. You can use configuration file

entries. However, remember that these are essential to security, and you will want to make it easy for an end user of your LINQ provider to use configuration file encryption built into .NET. That is, ASP.NET has a way to encrypt web.config entries on the deployment platform. The *TwitterContext* has a constructor overload for accepting credentials, leaving the decision of how that information should be stored up to the programmer who is coding queries for LINQ to Twitter.

If you have unique connection parameters, such as TCP/IP ports or anything else not covered here, you should consider how that information will be passed to your LINQ provider so that it can connect to the data source.

Additional Information Resources

The primary sources of information for LINQ to Twitter came from Matt Warren's blog and the .NET Framework walkthrough documentation for building the LINQ to TerraServer provider. Here are the URLs so you can find them yourself:

- The Wayward WebLog <http://blogs.msdn.com/mattwar/rss.xml>
- LINQ to TerraServer <http://msdn.microsoft.com/en-us/library/bb546158.aspx>

These samples should give you alternative views of how a LINQ provider can be built. In addition to this, you can navigate in your browser to your favorite search engine and begin a search for LINQ providers. You'll find several open-source implementations, which give you additional examples. Once you've learned about the LINQ to Twitter implementation, covered next, you'll be more prepared to understand the raw source code in other examples.

Implementing LINQ Provider Interfaces

As you learned earlier, two primary interfaces are central to LINQ provider development: *IQueryable<T>* and *IQueryProvider*. Additionally, there is an *IOrderedQueryProvider<T>* you can implement to support sorting. This section will explain what these interfaces do, their relationship to each other, and the essentials of how to implement them for LINQ to Twitter. The following sections of this chapter dig deeper into the LINQ to Twitter implementation, building upon what you learn here.

Understanding the *IQueryable<T>* Interface

The *IQueryable<T>* interface is for types that can be queried. Some of the LINQ providers you are already familiar with implement this interface. For example, the *DataContext* type in LINQ to SQL has properties of type *Table<T>*, where *T*, is an entity that maps to a database table. *Table<T>* implements *IQueryable<T>*. Therefore, all of the entities in LINQ to SQL are *IQueryable<T>*. Similarly, LINQ to Twitter exposes objects that are the data source of queries, which are also *IQueryable<T>*, meaning that you can query these objects. [Figure 10-1](#) shows the FCL interfaces and their relationships.

Not only can you see *IQueryable<T>* in [Figure 10-1](#), but you also can see *IQueryable* (non-generic) and *Queryable*. The *Queryable* class contains all of the extension methods for *IQueryable<T>*. All these types have similar sounding names; they are related, but each has its own purpose. First, let's look at the syntax of *IQueryable<T>*:

```
public interface IQueryable<T> :  
    IQueryable, IEnumerable<T>, IEnumerable  
{  
}  
}
```

As you know, since *IQueryable<T>* derives from *IQueryable*, all *IQueryable<T>* derived classes must also implement *IQueryable* members. Here is the *IQueryable* definition:

```
public interface IQueryable
{
    Type ElementType { get; }
    Expression Expression { get; }
    IQueryProvider Provider { get; }
}
```

When a LINQ provider executes a query, the return type will be represented as the *ElementType* property. *Expression*, is a reference to the expression tree holding the information to be evaluated. The *Provider* property is an *IQueryProvider* that is responsible for evaluating the expression tree, held by *Expression*; executing the query; and returning the result, of the same type as *ElementType*. The next section explains the *IQueryProvider* interface.

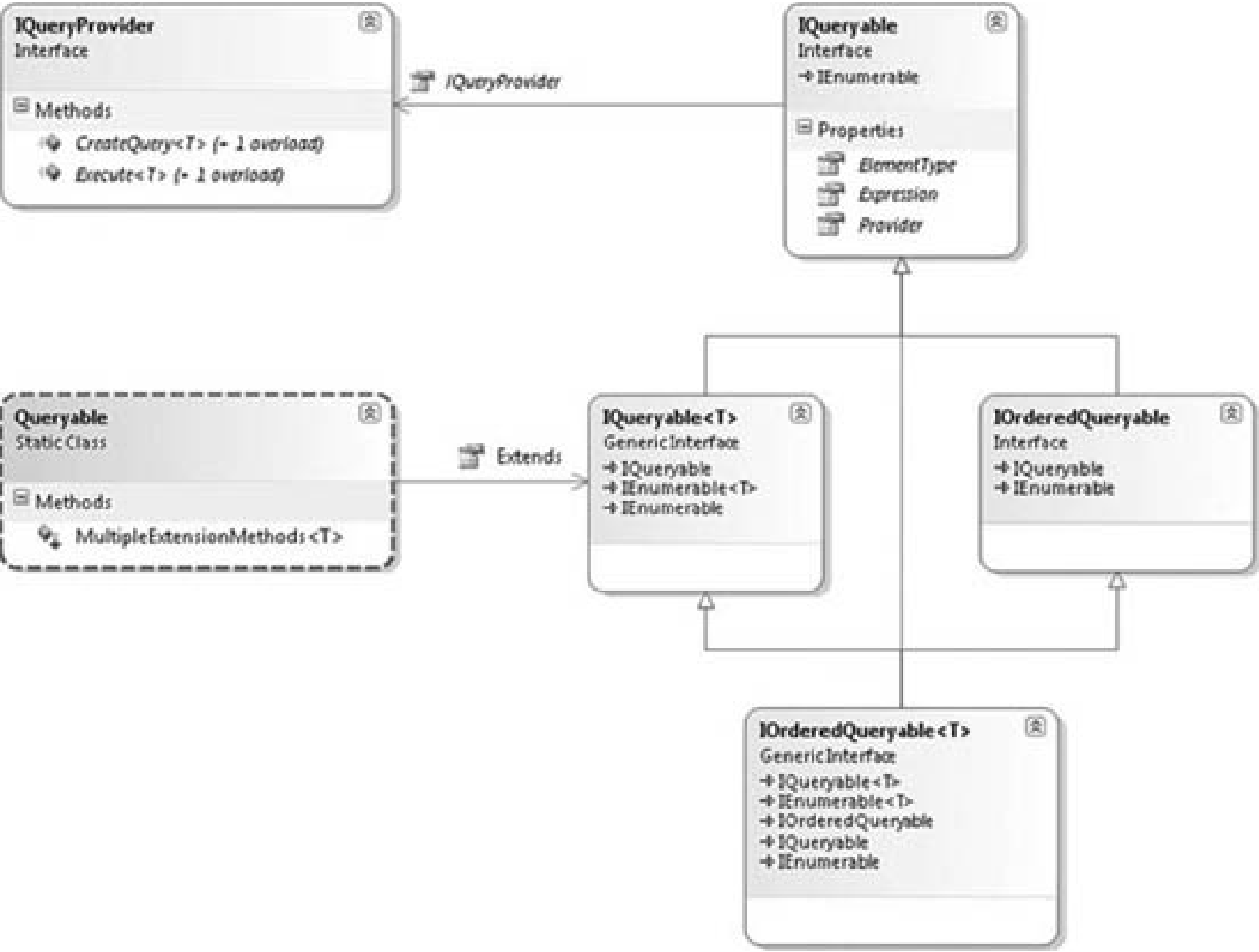


FIGURE 10-1 The .NET Framework Class Library LINQ interfaces

Understanding the *IQueryProvider* Interface

Another object in [Figure 10-1](#) is the *IQueryProvider* interface, referenced by the *Provider* property of

IQueryable. When implementing a LINQ provider, derive a class from *IQueryProvider*. This newly derived class will handle the bulk of the logic required to implement your provider. Of course, you'll build an appropriate object model for handling provider tasks, but it is the *IQueryProvider* derived type that manages all of this. Here's what *IQueryProvider* looks like in code:

```
public interface IQueryProvider
{
    IQueryable<T> CreateQuery<T>(Expression expression);
    IQueryable CreateQuery(Expression expression);

    IQueryable<T> Execute<T>(Expression expression);
    object Execute(Expression expression);
}
```

IQueryProvider has generic and non-generic overloads of *CreateQuery* and *Execute* methods. The purpose of *CreateQuery* is to generate an *IQueryable<T>* for each method of the query. Remember that LINQ uses deferred execution, so *CreateQuery* doesn't perform any queries; it just returns the expression tree in a form that is manageable. The *Execute* method is where the query actually runs. It will be called anytime consuming code performs an action that materializes the query, that is, *foreach*, *ToList*, or *ToArray*.

Understanding *IOrderedQueryable<T>*

Another interface in [Figure 10-1](#) is *IOrderedQueryable<T>*, which derives from *IOrderedQueryable*. The purpose of *IOrderedQueryable<T>* is to support sorting operations, such as *order by*, and so on. Here's the *IOrderedQueryable<T>* specification:

```
public interface IOrderedQueryable<T> :
    IQueryable<T>, IEnumerable<T>,
    IOrderedQueryable, IQueryable, IEnumerable
{
}
```

IOrderedQueryable<T> doesn't have any members, but can be used with extension methods through *IQueryable<T>*, which it derives from.

Next, you'll learn how to implement these interfaces to further the development of the LINQ to Twitter provider.

Implementing *IQueryable<T>*

Most implementations of *IQueryable<T>* or *IOrderedQueryable<T>*, will be similar—they initialize the *IQueryProvider* derived type, *Provider*; the expression tree, *Expression*; and the type of element operated on, *ElementType*. Essentially, the *IQueryable<T>* derived type holds references to the primary objects required for implementing the provider.

Here's an implementation of *IQueryable<T>*, from LINQ to Twitter. This implementation is based upon example code from the .NET Framework documentation, which is also similar to Matt Warren's blog postings, discussed in the previous section. I've customized it by changing identifiers and adding a new constructor. This example can be a template for implementing your own provider:

```
public class TwitterQueryable<T> : IOrderedQueryable<T>
{
```

```

public TwitterQueryable()
{
    Provider = new TwitterQueryProvider();
    Expression = Expression.Constant(this);
}
public TwitterQueryable(TwitterContext context)
: this()
{
    (Provider as TwitterQueryProvider).Context = context;
}

public TwitterQueryable(
    TwitterQueryProvider provider,
    Expression expression)
{
    if (provider == null)
    {
        throw new ArgumentNullException("provider");
    }
    if (expression == null)
    {
        throw new ArgumentNullException("expression");
    }
    if (!typeof(IQueryable<T>).IsAssignableFrom(expression.Type))
    {
        throw new ArgumentOutOfRangeException("expression");
    }

    Provider = provider;
    Expression = expression;
}

public IQueryProvider Provider { get; private set; }
public Expression Expression { get; private set; }

public Type ElementType
{
    get { return typeof(T); }
}

public IEnumerator<T> GetEnumerator()
{
    return (Provider.Execute<IEnumerable<T>>(Expression)).GetEnumerator();
}

IEnumerable IEnumerable.GetEnumerator()
{
    return (Provider.Execute<IEnumerable>(Expression)).GetEnumerator();
}
}

```

In the preceding code, you can see that the *TwitterQueryable* class implements *IOrderedQueryable*, which means that *TwitterQueryable* also implements *IQueryable<T>*. Each constructor initializes the *Provider* and *Expression* properties. One constructor accepts a *TwitterContext*, which holds credential information and query execution logic. It is modeled somewhat after the LINQ to SQL *DataContext*. A later section of this chapter will explain how *TwitterContext* works.

Whenever you materialize the query, one of the *GetEnumerator* overloads executes, delegating to the

matching *GetEnumerator* of the type assigned to *Provider* during constructor initialization. This is where the *IQueryProvider* type, discussed next, gets involved.

Implementing IQueryProvider

The *IQueryProvider* implementation, *TwitterQueryProvider*, supports building the initial expression tree before materialization and executing the query when the materialization occurs. From the perspective of a LINQ provider, materialization is the process of translating an expression tree into a data source–specific form, executing the query on the data source, and translating the data source results into a form understood by the caller. While expression tree building is performed by default code, materialization of that query is where most of your work will be. Here’s the *TwitterQueryProvider* implementation:

```
public class TwitterQueryProvider : IQueryProvider
{
    public TwitterContext Context { get; set; }

    public IQueryable CreateQuery(Expression expression)
    {
        Type elementType = TypeSystem.GetElementType(expression.Type);
        try
        {
            return (IQueryable)Activator.CreateInstance(
                typeof(TwitterQueryable<>)
                    .MakeGenericType(elementType),
                new object[] { this, expression });
        }
        catch (TargetInvocationException tie)
        {
            throw tie.InnerException;
        }
    }

    public IQueryable<TResult> CreateQuery<TResult>(Expression expression)
    {
        return new TwitterQueryable<TResult>(this, expression);
    }

    public object Execute(Expression expression)
    {
        return Context.Execute(expression, false);
    }

    public TResult Execute<TResult>(Expression expression)
    {
        bool IsEnumerable = (typeof(TResult).Name == "IEnumerable`1");

        return (TResult)Context.Execute(expression, IsEnumerable);
    }
}
```

Except for the class identifier, *Context* property, and *Execute* method implementation, all of the preceding code is reusable as is. In fact, Matt Warren added an abstract *Query* class to his blog that implements this class, except for an *abstract Execute* method that you must implement in your own derived class.

The extension methods in the *Queryable* class use the provider reference in *TwitterQueryable* to get a

reference to an instance of *TwitterQueryProvider* just shown. Then the extension method calls *CreateQuery*, passing in an expression tree representation of itself. As you just saw, each *CreateQuery* method returns a *TwitterQueryable*, which is an *IQueryable<T>* instance.

At first, it might sound redundant for a query, which is already *IQueryable<T>*, to pass in an expression tree and return *IQueryable<T>* references when the C# compiler already converts query syntax to extension syntax. However, consider the composition scenario that I showed you in [Chapter 9](#), where you can build your query dynamically with extension methods. Each time the code composes a new extension on a query, it must return a value and assign it to an *IQueryable<T>* (or another interface type, such as *IEnumerable* in the case of LINQ to Objects). Because of deferred execution, the method shouldn't execute until enumerated. Because the *Queryable* class extension methods call *CreateQuery*, they're able to perform this composition at run time, building the expression tree, and enabling deferred execution.

Upon materialization, which can be invoked by a *foreach* loop, *ToList*, or by another method operating on the *IQueryable<T>*, that kicks off enumeration, *GetEnumerator* on *TwitterQueryable* is invoked. *GetEnumerator*, in *TwitterQueryable*, delegates to the provider, referenced by the *Provider* property, which is *TwitterQueryProvider*. The preceding code shows the implementation of *Execute* in *TwitterQueryProvider*.

The preceding *Execute*, method delegates to *TwitterContext*, whose instance is referenced by the *Context* property. The choice to delegate the call to *TwitterContext* is convenient because that's where the connection information is. Regardless of where the actual execution occurs, you still need *TwitterQueryProvider*, because *IQueryable<T>* contains the *Provider* property that refers to *IQueryProvider* (*TwitterQueryProvider* in this case). *TwitterContext* has some similarity to the LINQ to SQL *DataContext*, which provides object management services. If *TwitterContext* should be enhanced to provide more object management services, this design could support that.

Building TwitterContext

The central bits of the LINQ to Twitter provider, where most of the custom work needs to be done, reside in *TwitterContext*. Essential services include connection management, provider access, and query execution. Each of the following sections breaks the *TwitterContext* class into pieces, explaining the secrets at the heart of LINQ to Twitter.

TwitterContext Instantiation

You have a choice of instantiating *TwitterContext* with or without credentials. Here's the top section of *TwitterContext*, showing namespaces, class declaration, credential properties, and constructors:

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Linq.Expressions;
using System.Net;
using System.Xml.Linq;
namespace LinqToTwitter
{
    public class TwitterContext
    {
```



```
public string UserName { get; set; }
public string Password { get; set; }
```

```
public TwitterContext()
{
    UserName = string.Empty;
    Password = string.Empty;
}
```

```
public TwitterContext(string userName, string password)
{
    UserName = userName;
    Password = password;
}
```

As just shown, the *System.Linq*, *System.Linq.Expressions*, and *System.Xml.Linq* namespaces are necessary because *TwitterContext* works extensively with expression trees and LINQ to XML, and requires the use of other LINQ types. When making the connection to Twitter and building streams, *TwitterContext* uses types in the *System.Net* and *System.IO* namespaces. The *System* and *System.Collections.Generic* namespaces are self-explanatory.

Unlike other custom provider types, *TwitterContext* doesn't derive from any other class or implement any interfaces. You might want to add an interface to facilitate mocking up and testing, but there isn't any provider-specific requirement to do so.

As mentioned in the first part of this chapter demonstrating how to use LINQ to Twitter, credentials aren't required for public queries, but they are required for any other query, which is a *friend* query in this implementation. Since *TwitterContext* performs execution, the constructor initializes the *Password* and *UserName* properties for whenever credentials are required to connect to Twitter.

Exposing IQueryable<T>

You must give the user access to *IQueryable<T>* for the provider to work. LINQ to SQL does this through properties that represent database tables and that return an *IQueryable<T>* where *T* is an entity type mapping to a database table. LINQ to Twitter offers a similar user experience via the *Status* property, which returns an *IQueryable<Status>*, shown next:

```
public TwitterQueryable<Status> Status
{
    get
    {
        return new TwitterQueryable<Status>(this);
    }
}
```

This property returns a *TwitterQueryable<Status>*. As you learned earlier, *TwitterQueryable<T>* implements *IQueryable<T>*. After C# converts query syntax to extension methods, the result of the *Status* property is the first *IQueryable<Status>* that is referenced at run time to call *CreateQuery<Status>*.

Calling *CreateQuery<Status>* when executing the query is what happens, but the next section shows you what happens when enumerating the final *IQueryable<Status>*.

Execution—The Bird's Eye View

If you recall from the earlier discussion of *TwitterQueryProvider*, whenever run-time enumeration of an *IQueryable<T>* occurs, *GetEnumerator* in *TwitterQueryable* executes and delegates the call to *Execute* in *TwitterQueryProvider*, which then delegates to *Execute* in *TwitterContext*. Here's the implementation of *Execute*, which delegates its work to other types and methods:

```
internal object Execute(Expression expression, bool IsEnumerable)
{
    Dictionary<string, string> parameters = null;

    var whereFinder = new InnermostWhereFinder();
    var whereExpression =

        whereFinder.GetInnermostWhere(expression);

    if (whereExpression != null)
    {
        var lambdaExpression =
            (LambdaExpression)
            ((UnaryExpression)
            (whereExpression.Arguments[1])).Operand;

        lambdaExpression =
            (LambdaExpression)
            Evaluator.PartialEval(lambdaExpression);

        var paramFinder =
            new ParameterFinder<Status>(
                lambdaExpression.Body,
                new List<string> { "Type" });

        parameters = paramFinder.Parameters;
    }

    var statusList = GetStatusList(parameters);

    var queryableStatusList = statusList.AsQueryable<Status>();

    return queryableStatusList;
}
```

The *InnermostWhereFinder* is one of the classes borrowed from the LINQ to TerraServer example code from the .NET Framework documentation. It traverses the expression tree, passed in as the expression parameter, to find the value of the *where* clause. The *Operand* of the second argument of *whereExpression*, returned from *GetInnermostWhere* contains the *where* clause condition from the original query. For example, if you were doing a *public* query, according to the first section of this chapter that explained how to use LINQ to Twitter, the condition would be the following:

```
tweet => tweet.Type == type
```

The call to *PartialEval* is critical at this point. You see, the preceding lambda body contains a condition for the *type* variable, which is data in the expression tree. To make a Twitter request, we need to know what the real value of *type* is. The call to *PartialEval* will locate *type*, use reflection to obtain a delegate, and perform a call to *DynamicInvoke* to get the run-time value of *type*, which is the string *Public* if this were a *public* query. Finally, *PartialEval* replaces *type* in the lambda expression tree with

its value to produce this:

```
tweet => tweet.Type == "Public"
```

ParameterFinder traverses the lambda body expression tree containing the preceding lambda expression and returns properties with their values as a *Dictionary<string, string>*.

GetStatusList, which is detailed in the next section, makes the Twitter request and returns *Status* objects back to *Execute*, which are then translated back into *IQueryable<Status>* for the caller. Here's the *GetStatusList* implementation:

```
private List<Status> GetStatusList(
    Dictionary<string, string> parameters)
{
    var url = BuildUrl<Status>(parameters);
    var statusList = QueryTwitter(url);

    return statusList;
}
```

The *GetStatusList* method just shown is broken into two pieces: creating the URL, which is the *REST* query to the Twitter web service, and then executing the query and transforming results. The next section shows how to construct the proper URLs.

Transforming the Query

To make the request to Twitter, *TwitterContext*, must build a URL for a well-formed *REST* query. The task of forming this URL revolves around the values passed to the *BuildUrl* method. These are the same values extracted from the lambda body of the *where* clause in the *Execute* method. Right now, the only parameter available for LINQ to Twitter is *Type*. Here's the implementation of *BuildUrl*, which returns the URL used in the Twitter web service request:

```
private string BuildUrl<T>(Dictionary<string, string> parameters)
{
    string url = null;

    if (parameters == null ||
        !parameters.ContainsKey("Type"))
    {
        url =
            "http://twitter.com/statuses/public_timeline.xml";
        return url;
    }

    switch (typeof(T).Name)
    {
        case "Status":
            switch (parameters["Type"])
            {
                case "Public":
                    url =
                        "http://twitter.com/statuses/public_timeline.xml";
                    break;
                case "Friends":
                    url =
                        "http://twitter.com/statuses/friends_timeline.xml";
                    break;
            }
            break;
    }
}
```

```

        default:
            url =
                "http://twitter.com/statuses/public_timeline.xml";
            break;
    }
    break;
default:
    url =
        "http://twitter.com/statuses/public_timeline.xml";
    break;
}

return url;
}

```

As I mentioned earlier, when discussing the design of LINQ to Twitter, I could have saved the base URL in a configuration file, but felt that avoiding the abstraction and indirection would make the code clearer for the book. You would probably want to put the base URL in a configuration file if you were building a web service–based LINQ provider.

If a LINQ to Twitter query doesn’t contain the *Type* parameter, the query defaults to a *public* query. This simplifies the implementation; whether you would want to do this depends on your requirements and development philosophy.

Another simplification was to implement the logic of URL building as a set of nested *switch* statements. The implementation, via the preceding *switch* statement, isn’t too extensive, and I was able to put the code in one place, making it easier to describe for you. In a real implementation, you would want to take a more object-oriented approach. A Strategy pattern might be a good implementation to grow LINQ to Twitter as the Twitter web service API changes over time.

As defined at the beginning of this chapter in the instructions on how to use LINQ to Twitter, the only two features supported are friend and public requests, which are both status requests. Besides status, Twitter supports direct messages, friendship, account, and more categories of queries. This is where that determination is made of how to construct the URL for making the request. The next section shows how you can communicate with Twitter.

Communicating with Twitter

After calling *BuildUrl* from *GetStatusList*, you have a URL to send to Twitter. The Twitter API supports multiple data return formats, such as XML, JSON, and RSS, and LINQ to Twitter uses the XML option. Fortunately, we can use another LINQ provider, LINQ to XML, making the query and subsequent translation into *Status* objects extremely easy. Here’s the *QueryTwitter* method, showing how to do this:

```

private List<Status> QueryTwitter(string url)
{
    var req = HttpWebRequest.Create(url);
    req.Credentials = new NetworkCredential(UserName, Password);
    var resp = req.GetResponse();
    var strm = resp.GetResponseStream();
    var strmRdr = new StreamReader(strm);
    var txtRdr = new StringReader(strmRdr.ReadToEnd());
    var statusXml = XElement.Load(txtRdr);

    var statusList =
        from status in statusXml.Elements("status")

```

```

let dateParts =
    status.Element("created_at").Value.Split(' ')
let createdAtDate =
    DateTime.Parse(
        string.Format("{0} {1} {2} {3} GMT",
            dateParts[1],
            dateParts[2],
            dateParts[5],
            dateParts[3]))
let user = status.Element("user")
select
    new Status
    {
        CreatedAt = createdAtDate,
        Favorited =
            bool.Parse(
                string.IsNullOrEmpty(
                    status.Element("favorited").Value) ?
                    "true" :
                    status.Element("favorited").Value),
        ID = status.Element("id").Value,
        InReplyToStatusID =
            status.Element("in_reply_to_status_id").Value,
        InReplyToUserID =
            status.Element("in_reply_to_user_id").Value,
        Source = status.Element("source").Value,
        Text = status.Element("text").Value,
        Truncated =
            bool.Parse(status.Element("truncated").Value),
        User =
            new User
            {
                Description =
                    user.Element("description").Value,
                FollowersCount = int.Parse(
                    user.Element("followers_count").Value),
                ID = user.Element("id").Value,
                Location = user.Element(
                    "location").Value,
                Name = user.Element("name").Value,
                ProfileImageUrl = user.Element(
                    "profile_image_url").Value,
                Protected = bool.Parse(
                    user.Element("protected").Value),
                ScreenName =
                    user.Element("screen_name").Value,
                URL = user.Element("url").Value
            }
    };

return statusList.ToList();
}
}
}

```

After you build the *HttpRequest*, notice how the *NetworkCredential* uses the *UserName* and

Password properties to add security credentials to the request. Subsequent statements get a stream back from the HTTP request, which is subsequently transformed through various stream types until we finally end up with a *TextReader*, *txtRdr*, that the *Load* method of *XElement* will accept. Getting information from a secured URL into an *XElement* seems like the long way around what should be an easier problem, so it is something you might want to tuck into your Toolbox in case you need it for other tasks.

Using LINQ to XML, the query transforms the document returned from the Twitter request into the *Status* and *User* objects, shown at the beginning of this chapter.

Several parts of this query demonstrate effective implementation of LINQ technology. Notice the effective use of the *let* clauses, splitting a string into the *dateParts* array *range* variable, indexing into *dateParts* to build a formatted string named *createdAtDate*, and then parsing *createdAtDate* in the query projection for the *CreatedAt* property. Also, this query shows the power of LINQ: using LINQ to XML to translate data into an *IEnumerable* collection, translating that collection into a *List<Status>* via LINQ to Objects, and then passing the *List<Status>* back to the caller, who translates it to an *IQueryable<Status>* as the result of a LINQ to Twitter query.

Summary

Now you know how to build a custom LINQ provider through learning how LINQ to Twitter works and is implemented. It's helpful to have a plan up-front about how the LINQ provider will be built, and I gave you some tips and things to consider when doing your own planning.

The LINQ to Twitter implementation uses .NET Framework Class Library types to provide a standard way of interacting with the provider. You learned what these types are and their purpose. Then I showed you how to implement the various interfaces. The most work required for building a custom LINQ provider lies in creating the code to execute queries. You saw how to do this with a custom class named *TwitterContext*, which has similarities with the LINQ to SQL *DataContext* in that it provides data source access, configuration, and execution services.

Most importantly, remember that you don't have to write all of the code yourself. Refer to the sources I provided, such as the expression tree visitor, and reuse all that you can.

One topic that is always a concern in programming is performance. The next chapter addresses this issue by showing you another LINQ provider called Parallel LINQ (PLINQ), which lets you run LINQ to Objects queries on multiple threads.

CHAPTER 11

Designing Applications with LINQ

There are many ways to use LINQ technology, depending on the task you need to accomplish. You can use LINQ in your user interface, business logic, or data layers. You also have important considerations in deciding how specific LINQ providers are used. For example, deferred execution and deferred loading have performance implications that are significant in some scenarios. You need to know where the trade-offs are so you can make smart decisions when designing your applications.

Throughout the sections that follow, you'll see some of these architectural and design issues. You'll also learn the potential pros and cons of various implementations. As you progress, remember that architecture and design is an abstract topic; you'll need to look at your situation and determine whether a specific approach makes sense to you.

Introduction to the ASP.NET LinqDataSource

ASP.NET offers a data control named `LinqDataSource` for interacting with LINQ data sources. You can use it against *IEnumerable<T>* or *IQueryable<T>* data sources. The example in this section will show you the `LinqDataSource` wizard, explaining how to connect `LinqDataSource` to a LINQ to SQL `DataContext` to work with *Customer* records in the AdventureWorksLT database. `LinqDataSource` is attractive to beginners and hobbyists because of its low barrier to entry in getting something running quickly. It could also be used in rapid application development (RAD) scenarios. I'll go through it quickly so you can get the gist of what it does, and then I'll move on to a more maintainable approach in the next section.

The first thing you'll need to do if you're following along is to create a new ASP.NET website and add a LINQ to SQL `DataContext`, named **AdventureWorks**, for the AdventureWorks database. If you're unsure of how to do this, please refer to [Chapter 3](#), which has a complete description of how to work with LINQ to SQL and examples for the AdventureWorksLT database. Make sure you include the *Address*, *CustomerAddress*, and *SalesOrderHeader* tables in your `DataContext`. In fact, you can include all of the tables if you want to make sure you don't forget anything. The following steps will walk you through the process. They'll show you how to add a *ListView* control, add a `LinqDataSource` control to your page, and step through the `LinqDataSource` wizard to configure how to work with the Customer data from AdventureWorksLT.

1. Open an ASP.NET WebForm in the VS 2008 visual designer.
2. Drag and drop a *ListView* control onto the page. [Figure 11-1](#) shows what this looks like.
3. As you can see in [Figure 11-1](#), the *ListView* control has an action list, *ListView Tasks*, with a property named *Choose Data Source*. Click on the drop-down list, and select *New Data Source*, which will display the Data Source Configuration Wizard, shown in [Figure 11-2](#).

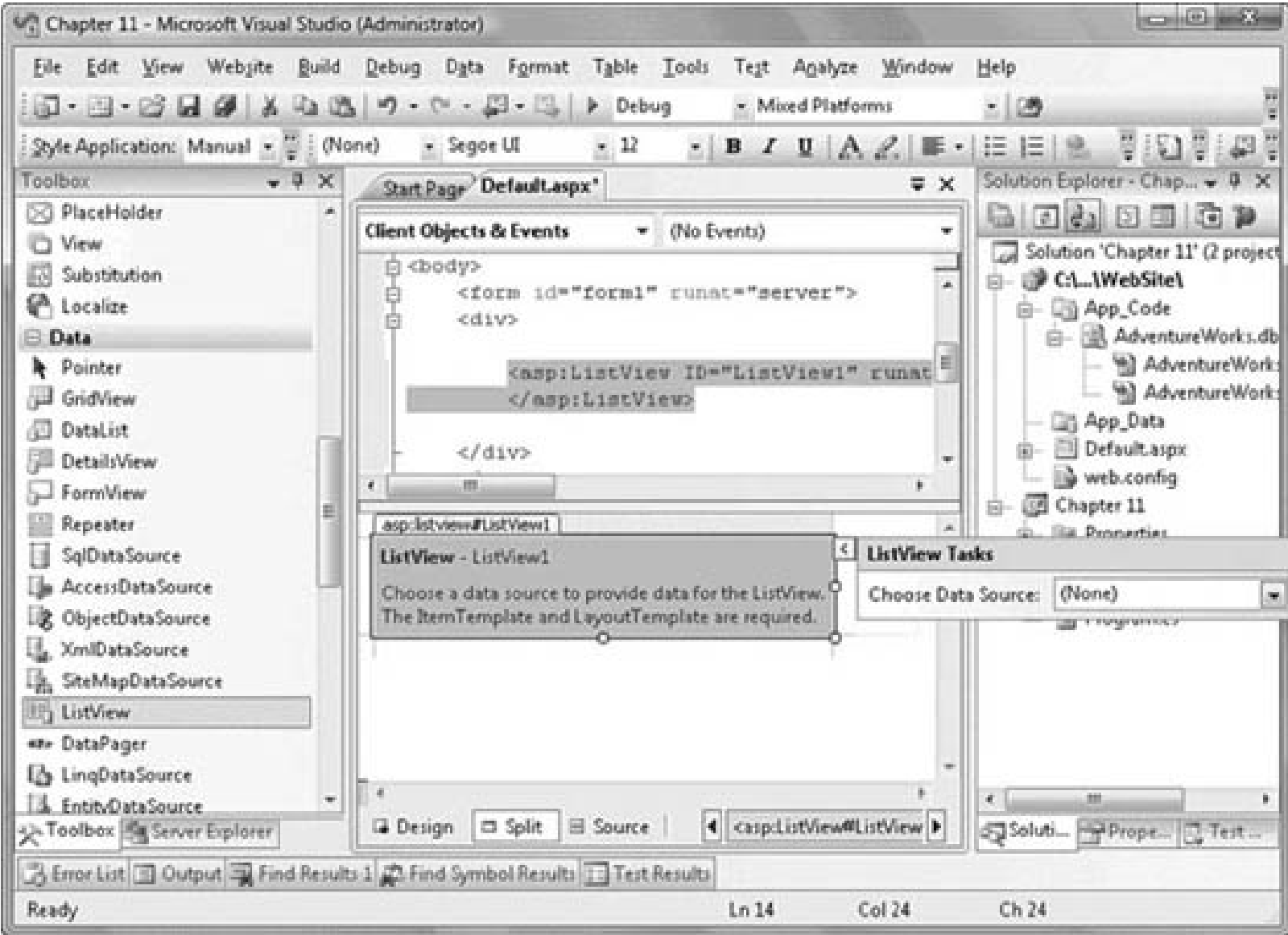


FIGURE 11-1 A ListView control on a WebForm

4. As shown in [Figure 11-2](#), select LINQ, which indicates that you want to use the LinqDataSource, and fill in the text box for Specify An ID For The Data Source with **CustomerLinqDataSource**. Click OK and you'll see the Choose A Context Object page in [Figure 11-3](#).
5. In [Figure 11-3](#), you can see that *AdventureWorksDataContext* is selected as the context object. If you named your LINQ to SQL DataContext with a different name, then select that name. When Show Only DataContext Objects is checked, you'll only see DataContext or DataContext derived objects that are available. If you uncheck Show Only DataContext Objects, you'll see all available objects, allowing you to select *IEnumerable<T>* types. Click the Next button to move to the Configure Data Selection page, shown in [Figure 11-4](#).

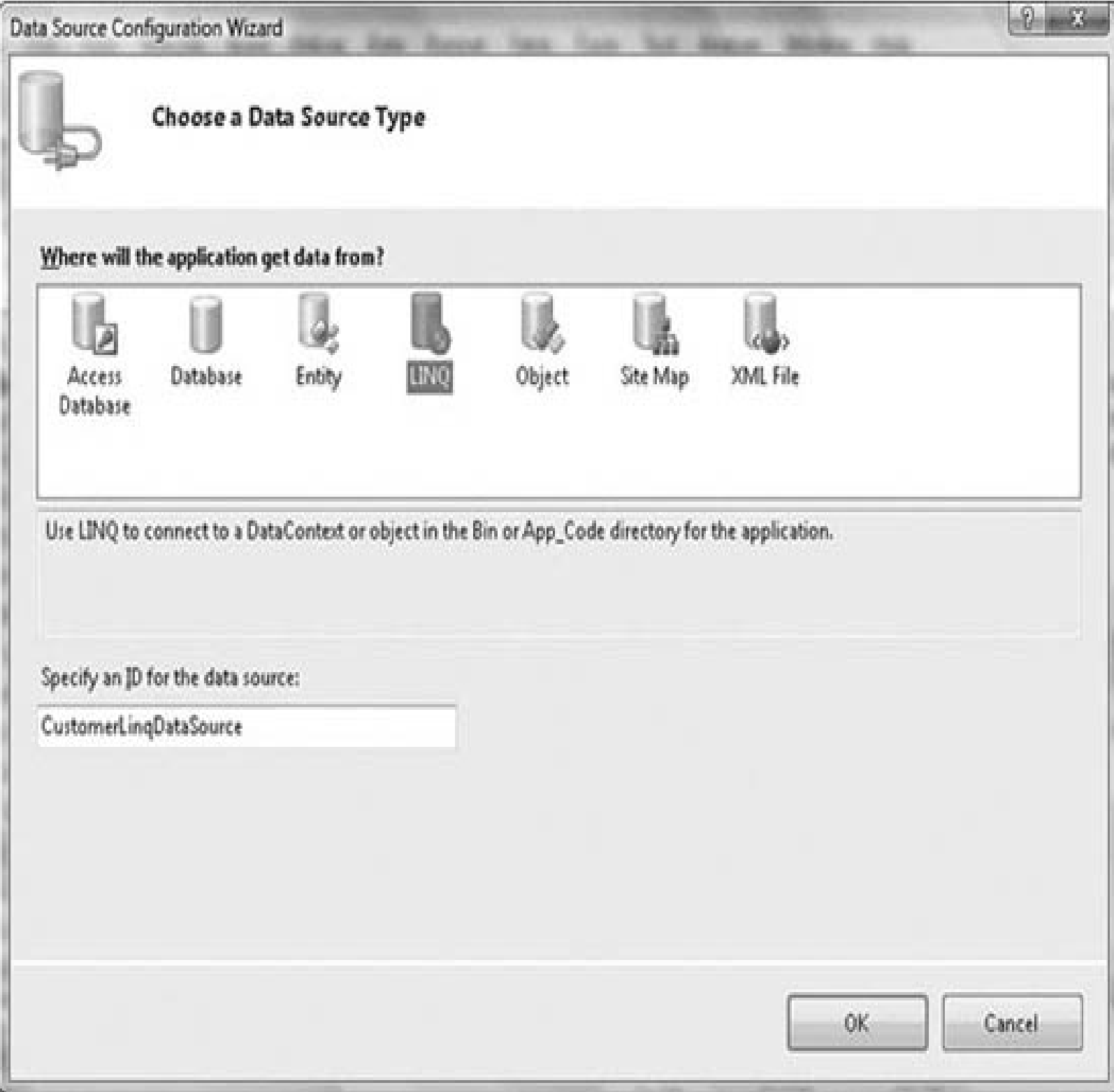


FIGURE 11-2 The Data Source Configuration Wizard

6. Notice that I've selected specific columns in [Figure 11-4](#) that I wanted to display in the *ListView*. Some of the columns appear because of foreign key relationships in the database, surfaced as entity associations in LINQ to SQL. The Where button pops up an editor that allows you to select columns and values for filtering results. The OrderBy button pops up an editor for specifying sorting options. The Advanced button allows you to specify whether the *LinqDataSource* will support insert, update, or delete operations. Because I've selected columns to display, the Advanced button is disabled. If I had selected all columns (the "*" check box), the

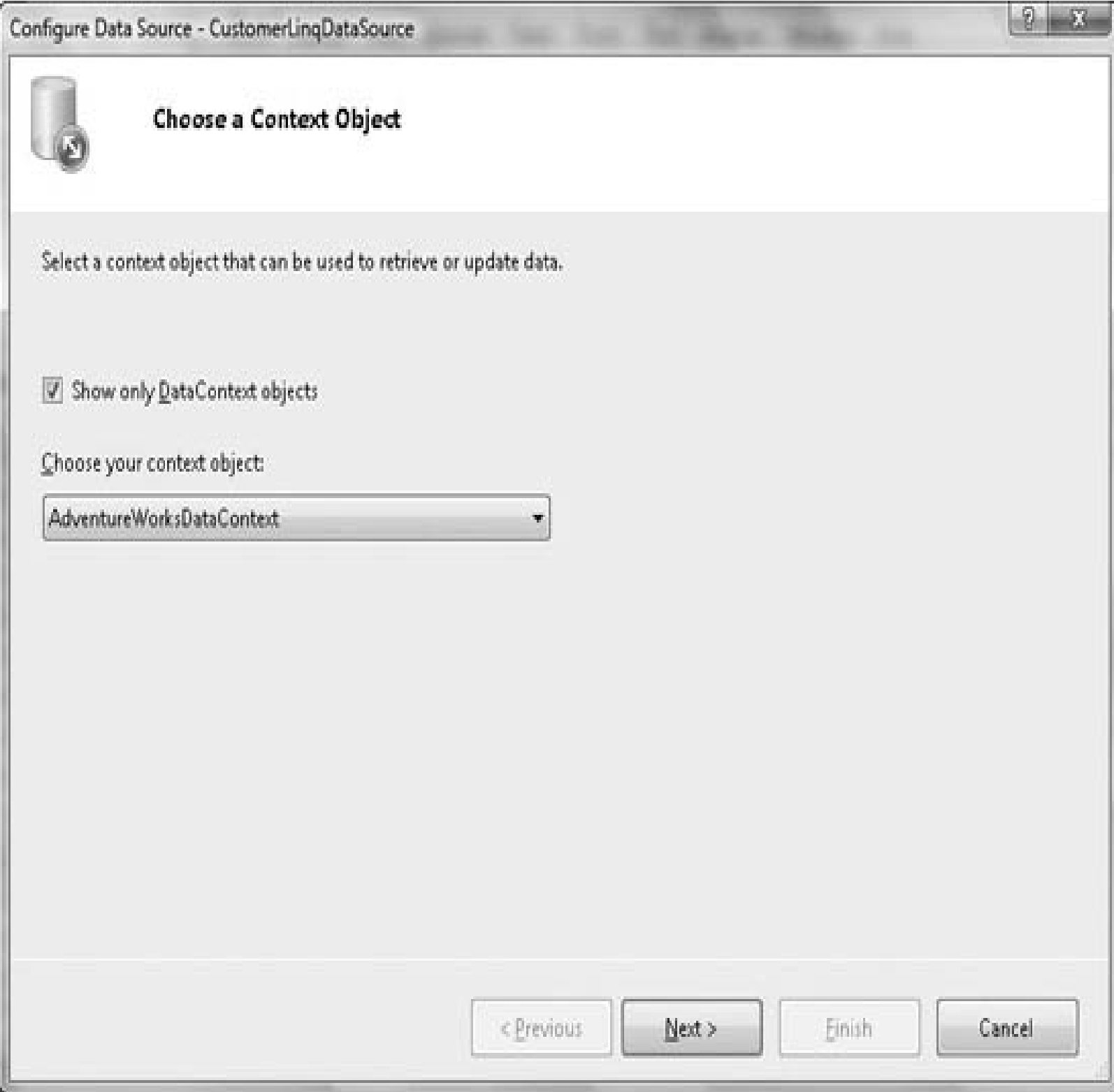


FIGURE 11-3 Choosing a Context Object for the LinqDataSource

Advanced button would be enabled, and Advanced options (insert, update, and delete) would be available. Click the Finish button to see the new LinqDataSource on the WebForm. [Figure 11-5](#) shows what this looks like after opening the *ListView* action list and selecting Configure ListView.

If you fail to select Configure ListView as instructed in Step 6, you'll receive a run-time error message, "An item placeholder must be specified on ListView 'ListView1'."

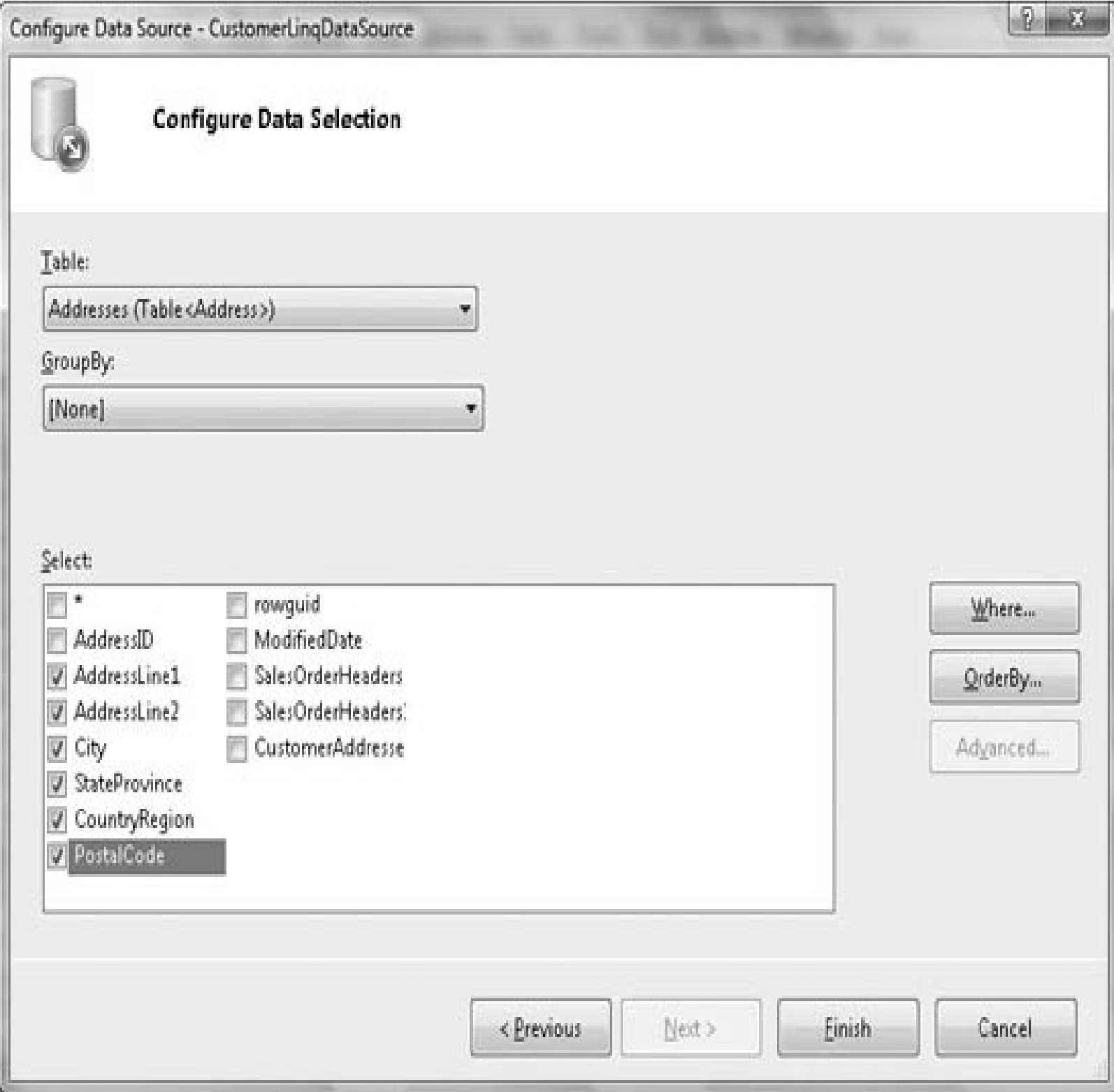


FIGURE 11-4 Configuring a LinqDataSource data selection

Here's the HTML generated for the LinqDataSource:

```
<asp:LinqDataSource ID="CustomerLinqDataSource" runat="server"
  ContextTypeName="AdventureWorksDataContext"
  Select="new (AddressLine1, AddressLine2, City,
    StateProvince, CountryRegion, PostalCode)"
  TableName="Addresses">
</asp:LinqDataSource>
```

The preceding *LinqDataSource* specifies the *ContextTypeName* and *TableName*, which are the *DataContext* and entity name, respectively. It also includes a *Select*, which specifies which properties of

the *Addresses* entity will be read.

This demonstrates how quickly you can use the *LinqDataSource* to display data on a screen. While the *LinqDataSource* could be useful in rapid prototyping scenarios, it quickly becomes cumbersome in n-layered architectures and gains complexity when you need to move beyond elementary scenarios such as error handling and validation. You can now execute this application, and this WebForm will show address data as specified by the *LinqDataSource*.

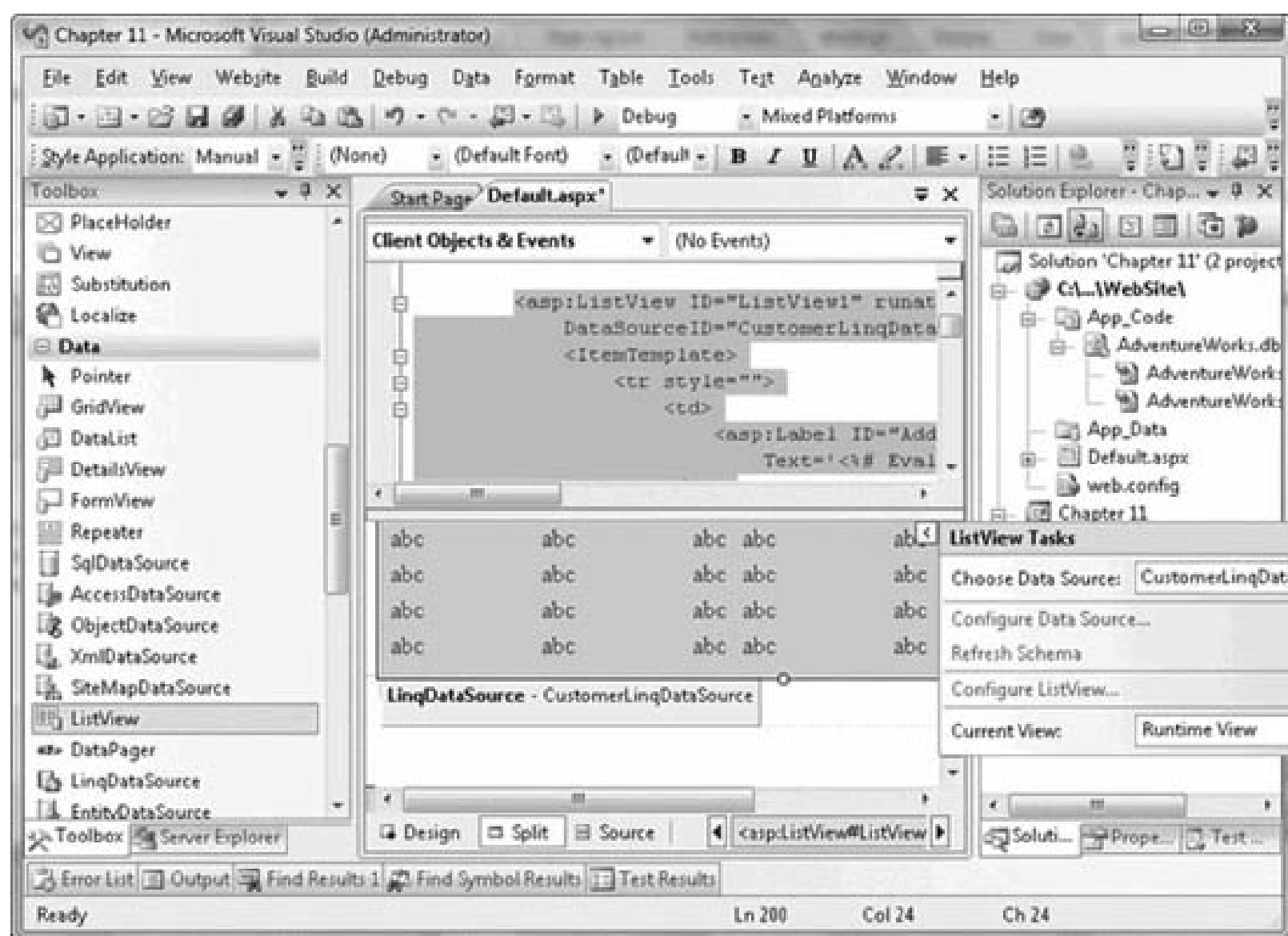


FIGURE 11-5 A ListView configured with a LinqDataSource

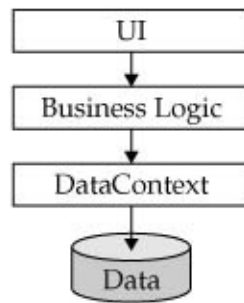
The next section shows you another way to achieve the RAD UI experience while keeping an n-layered architecture.

N-Layer Architecture with LINQ

A typical three-layer architectural model is often used for describing a basic n-layer application. It has a user interface (UI) layer, a business logic layer (BLL), and a data access layer (DAL). The UI only interacts with the user, accepting input and displaying output results in a form that makes sense to the user. The BLL is where you put your business logic, which includes business rules, validation, or data manipulation. The DAL has historically been used to provide a layer of abstraction from the database. The DAL separates all of the physical design-level logic of the database from the rest of the program.

This isolation simplifies the application and makes it more maintainable by compartmentalizing the work to be done. [Figure 11-6](#) helps you visualize the relationship between these layers, but I’ve made a change—it will be from the perspective of LINQ to SQL.

FIGURE 11-6 An n-layer architecture with LINQ to SQL



The n-layer architecture you see in [Figure 11-6](#) shows the three layers that I described previously, except that the DAL is named “DataContext.” In previous versions of .NET, before LINQ, you would write the DAL with ADO.NET, and the DAL would return a business object or a collection of business objects. All you would do is make a method call on the DAL, such as *GetCustomers*, and receive a *List<Customer>* where *Customer* was your business object. However, now the DAL is going to be a DataContext.

By “DataContext,” I mean that you can use the DataContext derived type of LINQ to SQL, *AdventureWorksDataContext*, as your DAL. You don’t have to write all of the ADO.NET code to get objects. Instead, you can use LINQ to SQL, which saves you many hours of work. The following sections cover what code you could put in each layer of your application, using the *Customers* table from AdventureWorks as an example. First, you’ll set up the DataContext, then define business logic, and finally build your user interface.

NOTE I’m using LINQ to SQL in this example, but your DAL could easily be LINQ to Entities,ObjectContext, or any other LINQ provider.

Creating the Data Access Layer (DAL)

When setting up your DAL, all you need to do is set up a LINQ provider. In the previous example, I used LINQ to SQL for creating the DAL, but in this example I’ll use LINQ to Entities. If you need more information about how LINQ to Entities works, you can refer to [Chapter 5](#). Since there are a couple ways to create the LINQ to Entities ObjectContext, similar in purpose to the LINQ to SQL DataContext, I’ll describe a few quick steps to get you started for this demo.

1. Create an ASP.NET website, called **NLayerWebsite**.
2. Right-click on the website, select Add New Item | ADO.NET Entity Data Model, name the new ADO.NET Entity Model **AdventureWorks.edmx**, and click the Add button. You’ll see a message about adding a special file type to the website and that you should add the new ADO.NET Entity Data Model to the App_Code folder, which is where code resides for ASP.NET websites. Click Yes to add the new ADO.NET Entity Data Model to the App_Code folder.
3. On the Entity Data Model Wizard, select Generate From Database, click Next, configure a connection for AdventureWorksLT, click Next, select all database objects, and click Finish. This creates a new ADO.NET Data Model for you. You can refer to [Chapter 5](#) for a more detailed explanation of the steps and figures for each page of the ADO.NET Data Model Wizard.

You have now created a new ADO.NET Data Model, with an ObjectContext derived type named *AdventureWorksLT_DataEntities*. This is your DAL and now you need to build business logic. The actual class names created on your system could be different.

Creating the Business Logic Layer (BLL)

The BLL contains many of the business rules for your application. At a minimum, it interacts with the DAL either to get data and return that data to the UI, or to receive requests from the UI and implement actions with the DAL to fulfill those requests.

The BLL performs validation on input data. While you have sophisticated UI controls for validating information, you should also validate in the BLL. For example, an ASP.NET validation control uses JavaScript to ensure that a required field is filled in. However, if users turn JavaScript off in their browser, you get whatever the users submit, which might be garbage that can crash your program. Therefore, validating input and throwing exceptions are common BLL operations.

The BLL implements any other business rules that define how your application should run. For example, if customers order a product or service and their ZIP code is in another state, you might want to ensure they are charged more for shipping and handling or for travel costs to provide the service.

The BLL for this application won't be very sophisticated. My main goal is to show you how to build the layers of the architecture, considering where LINQ fits in. Therefore, please forgive me for skipping input validation, exception handling, rules management, or any other instrumentation that would be necessary, but know it should be there in a production application.

Creating the Business Object

To create the basic BLL for this application, add a new class, named **CustomerManager**, to the App_Code folder. A message box will appear, telling you that your new class should be put in App_Code if you try to add the new class file to the website. I use the convention of calling my BLL classes *XManager*, where *X* is *Customer* in this case. You might have your own convention, so there isn't anything special about this class name, other than a desire for consistency.

The first thing you should do is decorate the *CustomerManager* class with the *DataObject* attribute, which will make the class discoverable in your UI. I'll describe how this works in the next section when I discuss UI. For now, this is how your new *CustomerManager* class will look:

```
using System;
using System.ComponentModel;
using System.Collections.Generic;
using System.Linq;
using AdventureWorksLT_DataModel;

[DataObject]
public class CustomerManager
{
}
```

To make the example more readable, I removed comments and other extraneous information that will differ from what you see in your IDE. The important part of this example is that *CustomerManager* is decorated with the *DataObject* attribute. You also need a *using* declaration for *System.ComponentModel*.

Adding a Select Method

The first method I'll show you selects records from the ObjectContext. It will be in response to a UI request for *Customers*. For simplicity, I won't perform any filtering or any other operations available

with LINQ. Here's a method that retrieves *Customer* objects:

```
using System;
using System.ComponentModel;
using System.Collections.Generic;
using System.Linq;
using AdventureWorksLT_DataModel;

[DataObject]
public class CustomerManager
{
    [DataObjectMethod(DataObjectMethodType.Select)]
    public List<Customer> GetCustomers()
    {
        using (var ctx = new AdventureWorksLT_DataEntities())
        {
            var customers =
                from cust in ctx.Customer
                select cust;

            return customers.ToList();
        }
    }
}
```

In the preceding example, you can see that we need the *System.Linq* namespace to make the LINQ to Entities query. The *GetCustomers* method returns a list of *Customer* entity objects. The *Customer* entity object is defined in *AdventureWorks.edmx* and resides in the *AdventureWorksLT_DataModel* namespace, which was defined automatically when adding the ADO.NET Entity Data Model to the project.

The *DataObjectMethod* attribute decorates the *GetCustomers* method, specifying (via the *DataObjectMethodType* enum) that it is a *Select* method. The *DataObjectMethodType* enum also has other members for *Fill*, *Insert*, *Update*, and *Delete* in addition to *Select*.

NOTE The *DataObject* and *DataObjectMethod* attributes aren't part of LINQ to Entities. They are part of the .NET Framework and facilitate binding UI data sources to the proper methods, as you'll see in the next section on UI.

Notice that I put the *AdventureWorksLT_DataEntities* ObjectContext in a *using* statement. In a desktop application, this might not be necessary. However, on the Web, you might need all of the scalability you can get. As you know, the *using* statement will automatically call *Dispose* on its parameters. This allows the ObjectContext to immediately release any resources it might be holding onto. I'll cover the implications of this later when discussing deferred loading. Microsoft designed the DataContext and ObjectContext as lightweight objects supporting this scenario, where you wouldn't incur extensive overhead for instantiation and disposal.

The query inside the *GetCustomers* method is just a normal LINQ query, which you've seen before, and consistent with the material in [Chapter 5](#). What I've done here is separate the query from the materialization. I could have returned the query directly and called *ToList* in one statement, like this:

```
return
    (from cust in ctx.Customer
     select cust)
    .ToList();
```

What I've found, though, is that if something goes wrong with your query, you want to debug it to see what is going on. Therefore, I separate the query creation from the materialization so that I can set a

breakpoint and use debugging tools to inspect the code.

Adding an Insert Method

In addition to data retrieval, you'll often need methods in your business object to insert, update, and delete data. The following example shows how to build an *Insert* method for adding a new *Customer* record to the database:

```
[DataObjectMethod(DataObjectMethodType.Insert)]
public int InsertCustomer(Customer cust)
{
    using (var ctx = new AdventureWorksLT_DataEntities())
    {
        cust.ModifiedDate = DateTime.Now;
        cust.PasswordHash = "123";
        cust.PasswordSalt = "456";

        ctx.AddToCustomer(cust);
        ctx.SaveChanges();
        return cust.CustomerID;
    }
}
```

This method demonstrates a standard LINQ to Entities insert operation, which you learned about in [Chapter 5](#). A useful aspect of the method is that it returns the ID of the inserted entity. Whenever the *ObjectContext* (same as with the *DataContext* for LINQ to SQL) inserts a new record, it retrieves its new ID. Therefore, you can grab the ID from the reference to the object that was just submitted. One benefit of writing insert methods to return the ID of the object is that it facilitates unit testing, where you would need the ID of the object just inserted so that you can call a retrieval method with just that ID to get that one object.

I set *ModifiedDate*, *PasswordHash*, and *PasswordSalt* so that I could get this particular example to run. It would be normal to set *ModifiedDate* with the current date and time, but you would have your own specific business logic that determines what *PasswordHash* and *PasswordSalt* should be in your own application. The *DataObjectMethod* attribute is set to *Insert*.

Adding an Update Method

The method that performs an update receives updated data, gets a reference to the existing object, modifies the existing object, and saves the changes. The following update method shows how to update a *Customer* record:

```
[DataObjectMethod(DataObjectMethodType.Update)]
public void UpdateCustomer(Customer updatedCust)
{
    using (var ctx = new AdventureWorksLT_DataEntities())
    {
        var existingCust =
            (from cust in ctx.Customer
             where cust.CustomerID == updatedCust.CustomerID
             select cust)
            .FirstOrDefault();

        existingCust.CompanyName = updatedCust.CompanyName;
        existingCust.EmailAddress = updatedCust.EmailAddress;
```



```

        existingCust.FirstName = updatedCust.FirstName;
        existingCust.LastName = updatedCust.LastName;
        existingCust.MiddleName = updatedCust.MiddleName;
        existingCust.Phone = updatedCust.Phone;

        ctx.SaveChanges();
    }
}

```

Whether you pass in arguments of individual attributes or an entire object from the UI is up to you. The goal is to pass the changes to the DAL. The *DataObjectMethodAttribute* is set to *Update*.

Adding a Delete Method

In addition to select, insert, and update, you’ll often need a delete method. The following example shows how to delete a *Customer* record from the database:

```

[DataObjectMethod(DataObjectMethodType.Delete)]
public void DeleteCustomer(Customer customer)
{
    using (var ctx = new AdventureWorksLT_DataEntities())
    {
        var custToDelete =
            (from cust in ctx.Customer
             where cust.CustomerID == customer.CustomerID
             select cust)
            .FirstOrDefault();

        ctx.DeleteObject(custToDelete);

        ctx.SaveChanges();
    }
}

```

The preceding example shows how to build a method that performs a delete operation. It passes in the ID of the object, retrieves a reference to the object, and then deletes the object. I could have tried to be clever and performed the operation in a single statement, like this:

```

ctx.DeleteObject(
    (from cust in ctx.Customer
     where cust.CustomerID == customer.CustomerID
     select cust)
    .FirstOrDefault());

```

However, it would be harder to debug, so I prefer to separate object retrieval from implementing the action.

You now have a simple BLL. You could add similar objects for the entities your application works with. Of course, this was a simplification that doesn’t preclude design patterns and other ways of organizing your BLL—it only demonstrates how to create a business object that facilitates working with LINQ. You might have your own ideas about how a BLL should be built, and this is one of many places to start; your business logic layer will grow from there.

As I said earlier, the *DataObject* and *DataObjectMethod* attributes decorate BLL members to facilitate communication with UI data source controls. Remember to save and ensure your BLL builds before trying to use BLL objects. The next section will show you how to build a UI that takes advantage of

these attributes.

Creating the User Interface Layer (UI)

Having set up a DAL and BLL, you now have all pieces in place to build a UI that uses the existing code. In fact, you could add multiple WebForms to your UI that use the one BLL *CustomerManager* object that was created in the previous section. We don't need multiple WebForms to demo how to use the *CustomerManager* object, but you should be aware that having a set of business objects such as *CustomerManager* inherently facilitates reuse if you need it. For this UI demo, I'll show you how to use the VS 2008 RAD tools to quickly build a functional WebForm that displays a list of *Customer* objects.

In particular, the UI will contain a *ListView* control that uses an *ObjectDataSource* control for binding to the custom business object, *CustomerManager*. The following steps show you how to do this:

1. Either add a new WebForm to your project or use Default.aspx. Open the WebForm in the visual designer in Design view.
2. Drag and drop a *ListView* control onto the visual designer. You can see the results in [Figure 11-7](#).

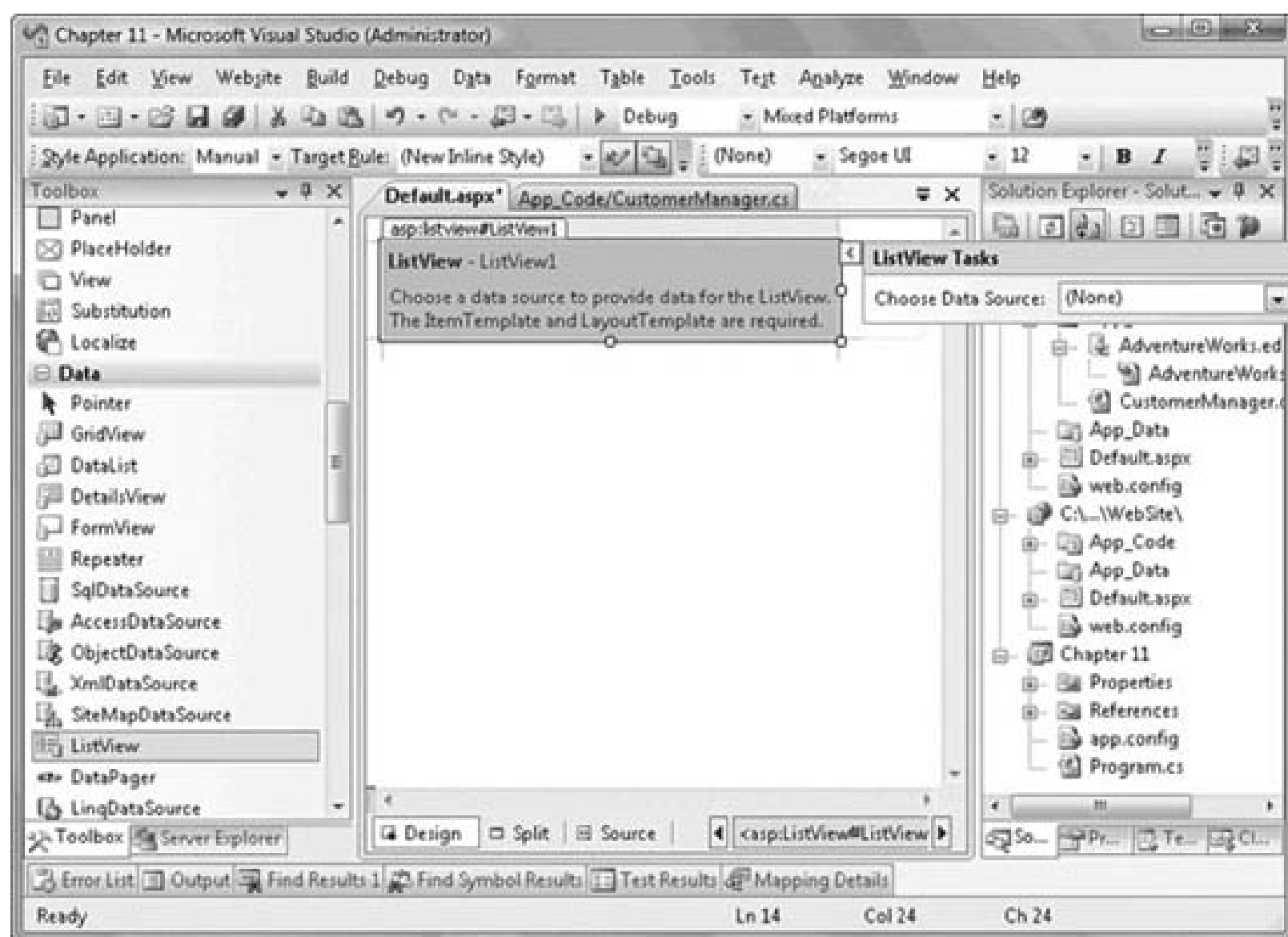


FIGURE 11-7 A ListView control, ready for data binding

3. In the ListView action list, shown in [Figure 11-7](#), select the Choose Data Source property, and then select New Data Source. You'll see the Data Source Configuration Wizard, shown in [Figure 11-8](#).
4. Select Object, which is an *ObjectDataSource*, and change the ID to **CustomerObjectDataSource**. Click OK and you'll see

the Choose A Business Object screen, shown in [Figure 11-9](#).

5. In the Choose Your Business Object drop-down list in [Figure 11-9](#), select *CustomerManager*, which is the custom business object created in the previous section. Notice that Show Only Data Components is checked, meaning that it will only display classes decorated with the *DataObject* attribute. If you recall from the previous section, the *CustomerManager* class is decorated with this attribute. If the box were unchecked, the drop-down list would display every class in the entire application, including referenced assemblies. This is one of the reasons you will want to decorate business objects that work with data with the *DataObject* attribute. Click Next to move to the Define Data Methods screen, shown in [Figure 11-10](#).



FIGURE 11-8 Selecting an ObjectDataSource with the Data Source Configuration Wizard

6. [Figure 11-10](#) shows a Choose A Method drop-down list for each of the SELECT, UPDATE, INSERT, and DELETE tabs. Each of these drop-down lists will only show methods from the object selected in Step 5, *CustomerManager*. The SELECT tab shows that I selected the *GetCustomers* method. Because I checked the Show Only Data Components box in Step 5, the

methods that appear in the drop-down lists for Define Data Methods only include the methods decorated with *DataObjectMethod* attributes. This filters out all other methods that won't be used. If you want a method to appear in the drop-down list, you must cancel the wizard, go back to your class, *CustomerManager* in this case, and decorate that method with a *DataObjectMethod* attribute. Because the *DataObjectMethod* attribute on *GetCustomers* was coded as a *Select* method, it appears in the SELECT tab, but not the others. Similarly, only methods with the *DataObjectMethod* attribute set to *Update*, *Insert*, or *Delete* appear in the UPDATE, INSERT, or DELETE tabs, respectively. Similar to the *DataObject* attribute, the *DataObjectMethod* attribute filters the methods and their purpose to save you time in running this wizard. In addition to the SELECT tab, select the appropriate methods for the UPDATE, INSERT, and DELETE tabs too. Click the Finish button.

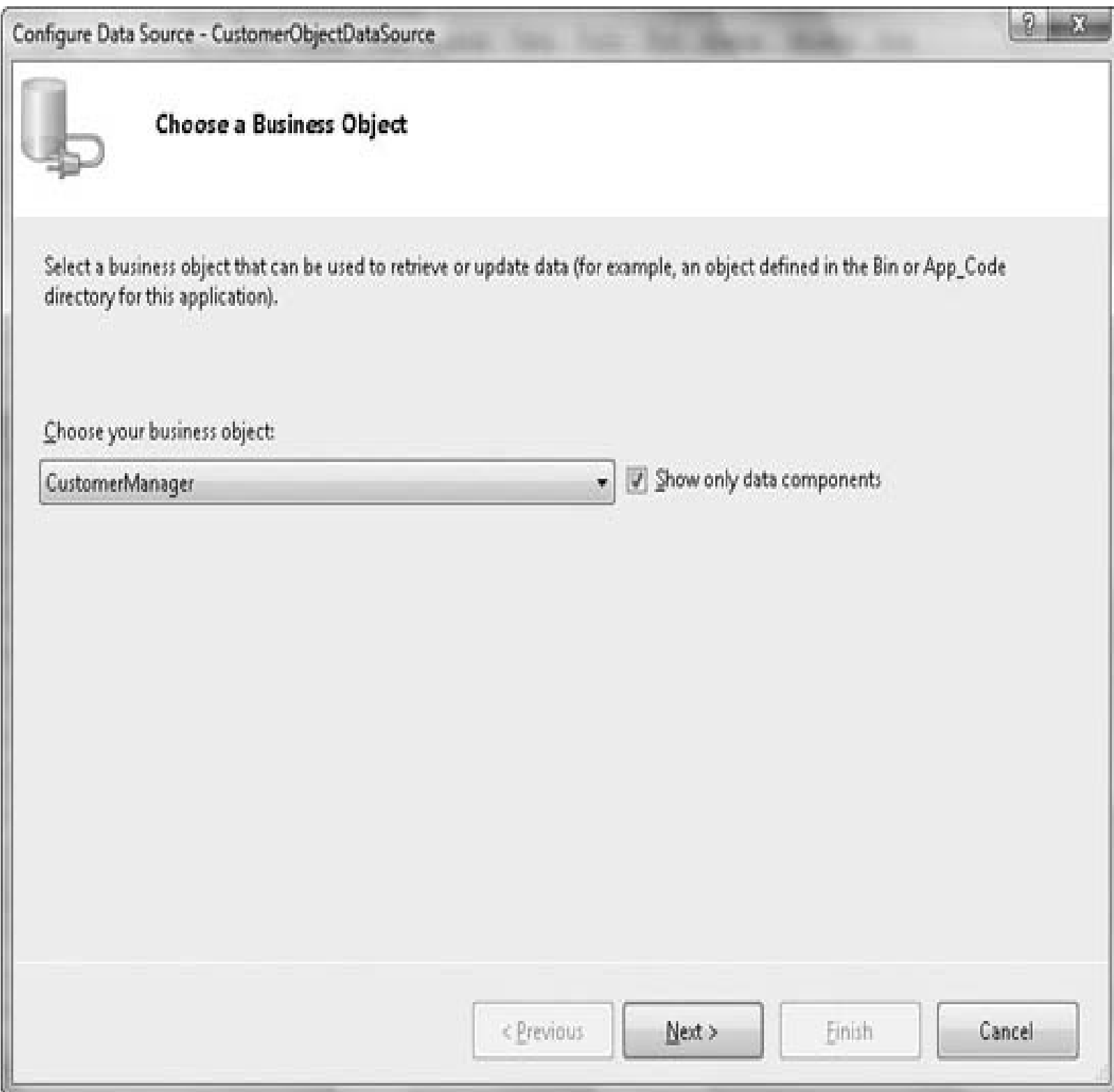


FIGURE 11-9 Choosing a business object for the ObjectDataSource

7. Anytime you bind an ObjectDataSource to a *ListView* control, you should open the *ListView* action list and select Configure

When configuring the *ListView*, you have several options in layout and initial coloring. While the detailed operation of a *ListView* is out of scope for this book, the important settings you should make are to choose the Enable Editing, Enable Inserting, and Enable Deleting check boxes. This gives you built-in support for the data manipulation methods added to the business object, *CustomerManager*. [Figure 11-12](#) shows the results in VS 2008, and [Figure 11-13](#) shows what the application looks like when it runs.

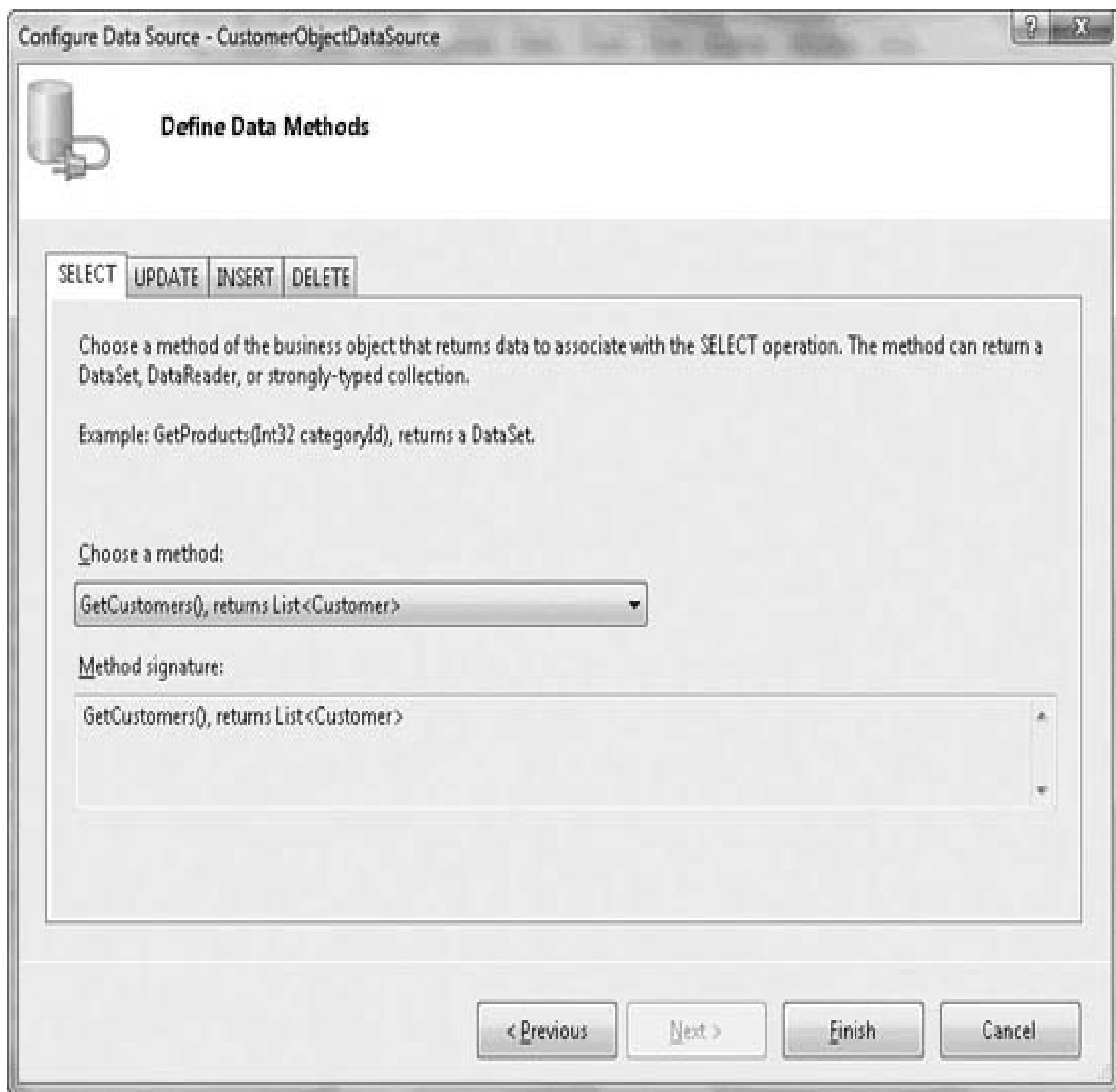


FIGURE 11-10 Selecting data object methods for the ObjectDataSource

Here's the code for the ObjectDataSource shown in [Figure 11-12](#):

```

<asp:ObjectDataSource ID="CustomerObjectDataSource" runat="server"
    DataObjectTypeName="AdventureWorksLT_DataModel.Customer"
    TypeName="CustomerManager"
    OldValuesParameterFormatString="original_{0}"
    SelectMethod="GetCustomers"
    InsertMethod="InsertCustomer"
    UpdateMethod="UpdateCustomer"
    DeleteMethod="DeleteCustomer">
</asp:ObjectDataSource>

```

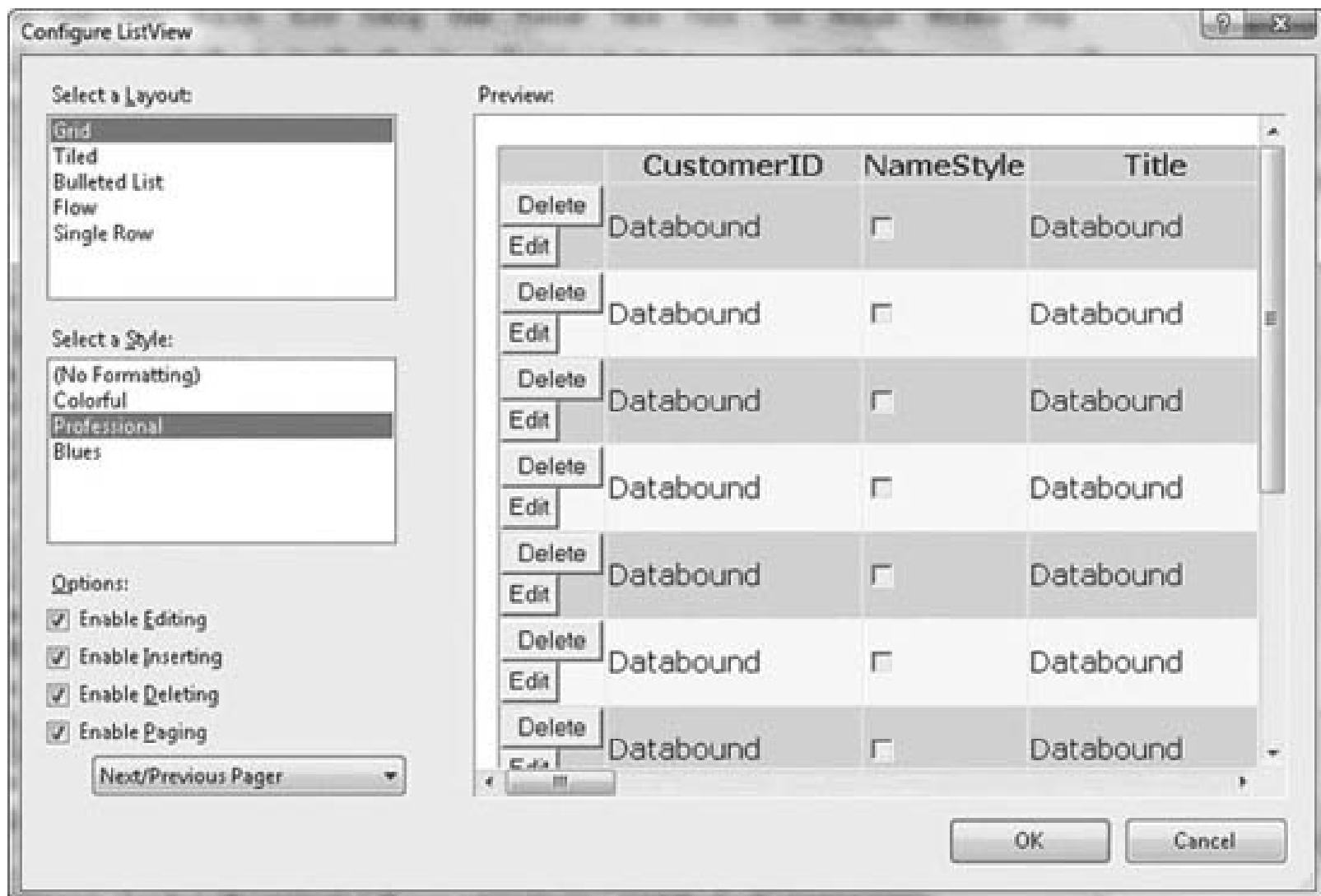


FIGURE 11-11 Configuring a ListView after binding to an ObjectDataSource

You can see the fully qualified name of the LINQ to Entities Customer object specified in *DataObjectTypeName*. The *SelectMethod*, *InsertMethod*, *UpdateMethod*, and *DeleteMethod* attributes have the method names that were specified in the Data Source Configuration Wizard. The .NET Framework documentation has more details on all of the options available in the ObjectDataSource. I just wanted to show you an effective alternative to the LinqDataSource when building n-layer applications using LINQ and needed to describe enough of the ObjectDataSource control to share the benefits and trade-offs with you.

You should go to Source view on your WebForm and comment-out the following columns for each template in the *ListView* control: *PasswordHash*, *PasswordSalt*, *rowguid*,

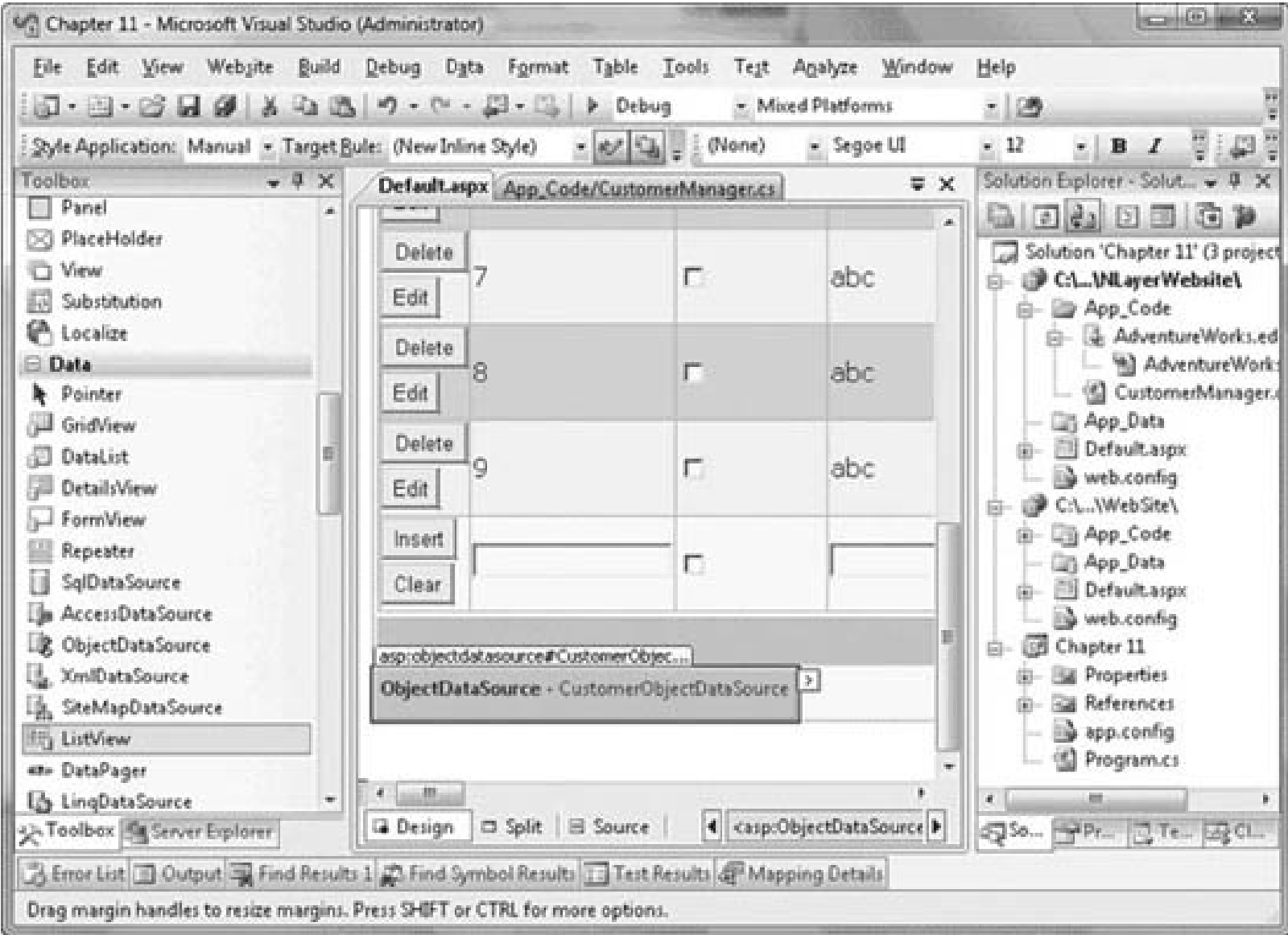


FIGURE 11-12 A configured ObjectDataSource in VS 2008

ModifiedDate, *CustomerAddress*, *SalesOrderHeader*, *EntityState*, and *EntityKey*. If you receive an exception message telling you that *EntityState* is *ReadOnly*, then you know you forgot to do this step. [Figure 11-13](#) shows the running application.

In just a few steps, you are able to quickly bind your UI to business objects. This is possible through using the ObjectDataSource and adding the right attributes to your business objects. Actually, the attributes on the business objects are optional, but they can help. Contrast this with the LinqDataSource shown earlier, and you can see that the ObjectDataSource delivers the speed of RAD UI development with the maintainability and reusability of n-layer architecture and design.

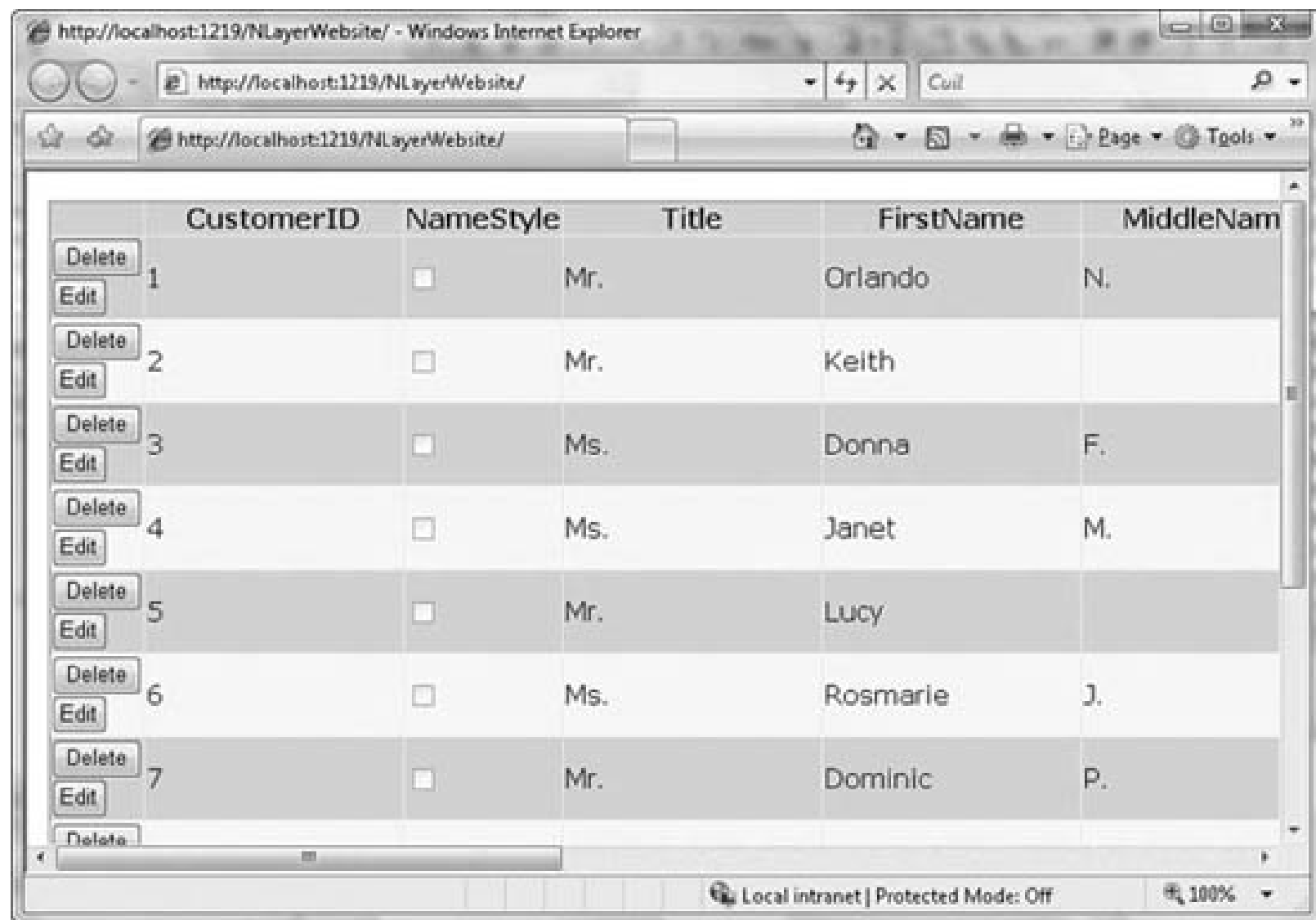


FIGURE 11-13 A WebForm running with an n-layer architecture, using LINQ to Entities

Handling Data Concurrency

When performing data operations, you have two types of data concurrency (aka locking): optimistic and pessimistic. With *pessimistic* concurrency you explicitly lock database records and don't allow another program to access them until you are done. Pessimistic concurrency can slow down a system, making applications wait for locks on data to be released if another application already has the lock. Of course, pessimistic concurrency is required in some situations, such as with financial transactions. On the other hand, *optimistic* concurrency allows many processes to work with the same data records simultaneously. This makes the system more efficient because there is less waiting to access records. In data entry situations, where the likelihood of conflicts in reading and writing to data is expected to be rare, optimistic concurrency could be a viable data locking strategy.

You can implement either optimistic or pessimistic concurrency with LINQ, as explained in the following sections. The examples in the following sections require a console application with a LINQ to SQL DataContext. Ensure you've included the *Address*, *Customer*, and *SalesOrderHeader* tables in the DataContext.

Implementing Pessimistic Concurrency with LINQ

A *TransactionScope* class in the .NET Framework allows you to put any code in a transaction, including

LINQ. You can use the *TransactionScope* class to implement pessimistic concurrency in LINQ transactions. To do so, you must add a project reference to the *.NET System.Transactions.dll* assembly. Additionally, you will need to add a *using* declaration to your code for the *System.Transactions* namespace. The following example shows how to perform pessimistic concurrency with a LINQ transaction that requests sales orders:

```
var txOptions =
    new TransactionOptions
    {
        IsolationLevel = IsolationLevel.Serializable
    };

List<SalesOrderHeader> orders = null;

using (var txScope = new TransactionScope(
    TransactionScopeOption.Required, txOptions))
{
    using (var ctx = new AdventureWorksDataContext())
    {
        orders =
            (from order in ctx.SalesOrderHeaders
             select order)
            .ToList();
    }
}

foreach (var order in orders)
{
    Console.WriteLine(
        "Number: " + order.SalesOrderNumber);
}
```

The preceding *TransactionScope* constructor uses two parameters: a *TransactionScopeOption* enum and a *TransactionOptions* class instance. *TransactionScopeOption* allows you to choose between *Required*, *RequiresNew*, and *Suppress*. The *TransactionOptions* instance has its *IsolationLevel* property set to *Serializable*. Other options of the *IsolationLevel* enum include *Chaos*, *ReadCommitted*, *ReadUncommitted*, *RepeatableRead*, *SnapShot*, and *Unspecified*. All of the settings correspond to applicable transaction settings that SQL Server supports, and you can review SQL Server Books On-Line or *Microsoft SQL Server 2008: A Beginner's Guide, Fourth Edition* (McGraw-Hill Professional, 2008).

Optimistic Concurrency

Unlike the pessimistic concurrency scenario with *TransactionScope*, LINQ to SQL has built-in support for optimistic concurrency. First, I'll show you a scenario that causes an optimistic concurrency exception. Then I'll describe the features of LINQ to SQL that help you handle optimistic concurrency.

When Optimistic Concurrency Conflicts Happen

An optimistic concurrency exception can occur whenever two people or programs try to perform overlapping update operations on the same record. Here's the scenario: User 1 reads a record from the database. User 2 reads the same record. User 2 modifies the record and saves it. Note that User 1 has not saved yet and that User 2 has already read and saved the same record that User 1 is looking at right now. Then User 1 saves the record—or I should say that User 1 *tries* to save the record because User 1 will

receive an exception when he or she tries to save the record. The problem is that the record has already been changed by User 2. Here's this same scenario enacted in code:

```
var user1Ctx = new AdventureWorksDataContext();
var user2Ctx = new AdventureWorksDataContext();
// user 1 reads
var user1AddressCopy =
    (from address in user1Ctx.Addresses
     where address.AddressID == 32
     select address)
    .Single();
// user 2 reads
var user2AddressCopy =
    (from address in user2Ctx.Addresses
     where address.AddressID == 32
     select address)
    .Single();
// user 2 modifies
user2AddressCopy.AddressLine2 = "26910 Indela Street";
// user 2 saves
user2Ctx.SubmitChanges();
// user 1 modifies
user1AddressCopy.AddressLine1 = "26910 Indela Avenue";
// user 1 saves
user1Ctx.SubmitChanges();
```

The preceding code results in a *ChangeConflictException* being raised with a message of "Row not found or changed." Each user is represented by a separate DataContext. You can see that User 2 modifies and updates after User 1 initially reads and before User 1 saves. The following sections describe features of LINQ to SQL to help you manage optimistic concurrency more effectively.

If you run the preceding example, including the code in the following sections, remember to change the street addresses on each run; otherwise there won't be a conflict with database data, and you won't see the results of a concurrency exception.

Conflict Detection via Properties

By default, LINQ to SQL will check every property of an object for changes when performing updates. This is managed through the *UpdateCheck* attribute of each entity. You can change this value to either *Always*, *WhenChanged*, or *Never*, indicating whether the property will participate in optimistic concurrency checks. To change this value, you can open the visual designer in VS 2008, select the property on the entity you want to change, open the Properties window, and modify the *UpdateCheck* property.

Specifying when a ConcurrencyConflictException Can Occur

You can specify that a *ConcurrencyConflictException* can occur either on the first conflict or after all updates have been processed. To do so, add a *ConflictMode* enum parameter to the call to *SubmitChanges* set to either *FailOnFirstConflict* or *ContinueOnConflict*. Here's an example that will process all records and then throw one exception after the update. The code requires a *using* declaration for the *System.Data.Linq* namespace:

```
try
{
```

```

        user1Ctx.SubmitChanges(ConflictMode.ContinueOnConflict);
    }
    catch (ChangeConflictException ccex)
    {
        foreach (var conflict in user1Ctx.ChangeConflicts)
        {
            var address = conflict.Object as Address;
            var metaTable = user1Ctx.Mapping.GetTable(typeof(Address));

            Console.WriteLine("Table: " + metaTable.TableName);

            foreach (var member in conflict.MemberConflicts)
            {
                Console.WriteLine("Member Current: " + member.CurrentValue);
                Console.WriteLine("Member Original: " + member.OriginalValue);
                Console.WriteLine("Member Database: " + member.DatabaseValue);
            }
        }
        Console.WriteLine(ccex.ToString());
    }
}

```

The preceding example shows you how to extract information from change conflict exceptions. Notice how the example iterates over the *ChangeConflicts*, which is a member of the *DataContext* and is populated when a *ChangeConflictException* occurs. You can extract information from each *ChangeConflict* and drill down into *MemberConflict* items to examine the actual properties (database fields) that are in conflict.

Resolving Change Conflicts

Continuing with the previous example, where each *ObjectChangeConflict* is processed, this section will show you how to resolve conflicts. Both *ObjectChangeConflict* and *MemberChangeConflict* have *Resolve* methods that accept a *RefreshMode* enum with three values: *KeepChanges*, *KeepCurrentValues*, and *OverwriteCurrentValues*. *KeepChanges* means that current values are kept, but other values are updated with database values. *KeepCurrentValues* will replace original values with database values and update the rest of the values with database values. *OverwriteCurrentValues* replaces all values with what is in the database. Here's an example that updates the *DataContext* object with the new values from the database:

```

try
{
    user1Ctx.SubmitChanges(ConflictMode.ContinueOnConflict);
}
catch (ChangeConflictException ccex)
{
    foreach (var conflict in user1Ctx.ChangeConflicts)
    {
        conflict.Resolve(RefreshMode.OverwriteCurrentValues);
    }
}

```

Calling the preceding *Resolve* updates the object in memory with changes that User 2 made in the database. You can also call *Resolve* on individual *MemberChangeConflict* instances, which you learned how to access in the previous section too.

Remember to change the street addresses, *Address1* and *Address2*, on each run; otherwise there won't

be a conflict with database data, and you won't see the results of a concurrency exception.

Working with Deferred Execution

Deferred execution is the feature of LINQ that delays executing a query until you actually make a request for the data. The primary benefit in deferred execution is that you can build a query in multiple statements without incurring the overhead of requests being executed on every statement.

To see an example of the problems that would be created without deferred execution, consider the scenario where you are building a search function. You need to have the basic query, multiple compositions to define search criteria, and then materialize the query. By “materialization” I mean that the query will actually execute. All of the composition statements prior to materialization do not execute the query. Here's an example, using LINQ to SQL, that takes a salesperson and a company type and performs a search on customers to return contact names of customers who meet the salesperson and company type criteria:

```
var salesPerson = "jillian0";
var companyType = "Bike";
using (var ctx = new AdventureWorksDataContext())
{
    // construct basic query
    var selectedCustomers =
        from cust in ctx.Customers
        select cust;

    // compose with sales person criteria
    selectedCustomers =
        selectedCustomers
        .Where(
            cust =>
                cust.SalesPerson.Contains(salesPerson));

    // compose with customer type criteria
    selectedCustomers =
        selectedCustomers
        .Where(
            cust =>
                cust.CompanyName.Contains(companyType));

    // materialize results
    selectedCustomers
        .ToList()
        .ForEach(
            cust =>
                Console.WriteLine(
                    "Contact: " +
                    cust.FirstName +
                    " " +
                    cust.LastName));
}
```

The preceding example uses a couple *Where* extension methods to filter query results. As you learned in [Chapter 9](#), extension methods are composable. Therefore, building a query like this, piece by piece, is quite easy. In the end, you get a single query to execute that contains all of the composed elements. The preceding example simply composes a couple *Where* methods, one after the other. However, in a real search scenario, you would have *if* statements and other conditional logic that would intelligently choose which filters were composed. Therefore, this offers a lot of power with the simplicity of LINQ.

Getting back to the subject of deferred execution, neither of the preceding compositional statements with the *Where* methods causes the query to execute. However, as soon as the *ToList* in the last statement runs, the query executes. The *ToList* materializes (executes) the query.

Imagine what would have happened without deferred execution. Four statements are in the algorithm, and you definitely would not want all four of the statements to execute the query, which would be very inefficient. Several LINQ extension methods invoke materialization, including *ToList*, *ToDictionary*, *ToLookup*, and *ToArray*. The Appendix describes each of these methods.

Another feature you need to consider in algorithm design is called *deferred loading*, which you'll learn about in the next section.

Working with Deferred Loading

Much of the work you do will include working with entities that have associations. For example, *SalesOrderHeader* entities can have *SalesOrderDetail* entities, which is a one-to-many relationship where the *SalesOrderDetail* table in the database has a foreign key to the *SalesOrderHeader* table. In the object-oriented perspective, *SalesOrderHeader* has a collection of *SalesOrderDetail*. Whenever you only need to work with *SalesOrderHeader* objects, you only need to think about the deferred execution features of your LINQ queries. However, as soon as you start traversing the associations where you need to retrieve information related to *SalesOrderHeader* such as *SalesOrderDetail*, another LINQ feature emerges that you must design your algorithms for—deferred loading.

Deferred loading is the LINQ feature for ensuring that related entities are not loaded until they are referenced. This is related to deferred execution in that the query doesn't execute until materialized, except that deferred loading is specific to associations with an entity that is already loaded.

By default when a query executes, only the entity that is requested will load. If you subsequently request an associated entity, then another query will be automatically executed to load the associated entity.

What Is Good about Deferred Loading?

Deferred loading is good for efficiency, resource utilization, and scalability because it only loads what is needed. Imagine what would happen if the scenario were turned around, where associated child entities loaded with their parent. More specifically, consider that *SalesOrderHeader* has associations with *SalesOrderDetail*. For every *SalesOrderHeader* loaded, the *SalesOrderDetail* would be loaded too. Maybe that isn't too bad if your *SalesOrderHeader* list is filtered so you don't grab every *SalesOrderHeader* in the database. Even in that case, consider that *SalesOrderDetail* has an association with *Product*, and *Product* has associations with *ProductModel* and *ProductCategory*. I could go on, but hopefully you get the picture. A single query could potentially try to bring a large portion of the database into memory, which would be catastrophic. Therefore, deferred loading is good because it lands on the side of safety when determining when records should be loaded.

What Could Be Problematic about Deferred Loading?

While deferred loading is designed to help prevent you from inadvertently loading a large portion of the database, at times it can cause inefficiencies. For example, what if you did want to pull all *SalesOrderDetail* records for a specific set of *SalesOrderHeader* entities? What would happen is that

you would make the *SalesOrderHeader* request and get all of the *SalesOrderHeader* objects, but not associated *SalesOrderDetail* objects. Then, every time you tried to access a *SalesOrderDetail*, a separate request would execute. Here's an example, using LINQ to SQL, that demonstrates this scenario:

```
using (var ctx = new AdventureWorksDataContext())
{
    ctx.Log = Console.Out;

    var salesOrders =
        from salesOrd in ctx.SalesOrderHeaders
        select
            new
            {
                salesOrd.SalesOrderNumber,
                salesOrd.TotalDue,
                salesOrd.SalesOrderDetails
            };
    foreach (var salesOrd in salesOrders.Take(3))
    {
        Console.WriteLine(
            "#: " + salesOrd.SalesOrderNumber +
            "Total: " + salesOrd.TotalDue);

        foreach (var salesDetail in salesOrd.SalesOrderDetails)
        {
            Console.WriteLine(
                "\tLine Total: " + salesDetail.LineTotal);
        }
    }
}
```

The preceding example performs a query on *SalesOrderHeader*. The outer *foreach* loop uses the *Take* operator to retrieve only the first three records. Notice that the inner *foreach* loop requests *SalesOrderDetail* records. What happens in terms of execution is that the outer *foreach* loop causes a query to return all of the *SalesOrderHeader* records in a single request, which is intended. However, the inner *foreach* loop will perform a load on the *SalesOrderDetail* for the current *SalesOrderHeader* being referenced, *salesOrd*. This results in a new query to retrieve the *SalesOrderDetail* records for each *SalesOrderHeader*. At the top of the listing, *Console.Out* is being assigned to the *Log* property of the LINQ to SQL context, producing the following output that demonstrates what I just explained:

```
SELECT TOP (3) [t0].[SalesOrderNumber],
               [t0].[TotalDue], [t0].[SalesOrderID]
FROM [SalesLT].[SalesOrderHeader] AS [t0]
-- Context: SqlProvider(Sql2005)
   Model: AttributedMetaModel Build: 3.5.30729.1
SELECT [t0].[SalesOrderID], [t0].[SalesOrderDetailID],
       [t0].[OrderQty], [t0].[ProductID], [t0].[UnitPrice],
       [t0].[UnitPriceDiscount], [t0].[LineTotal],
       [t0].[rowguid], [t0].[ModifiedDate]
FROM [SalesLT].[SalesOrderDetail] AS [t0]
WHERE [t0].[SalesOrderID] = @x1
-- @x1: Input Int (Size = 0; Prec = 0; Scale = 0) [71774]
-- Context: SqlProvider(Sql2005)
   Model: AttributedMetaModel Build: 3.5.30729.1
#: SO71774Total: 972.7850
   Line Total: 356.898000
   Line Total: 356.898000
```

```

SELECT [t0].[SalesOrderID], [t0].[SalesOrderDetailID],
       [t0].[OrderQty], [t0].[ProductID], [t0].[UnitPrice],
       [t0].[UnitPriceDiscount], [t0].[LineTotal],
       [t0].[rowguid], [t0].[ModifiedDate]
FROM [SalesLT].[SalesOrderDetail] AS [t0]
WHERE [t0].[SalesOrderID] = @x1
-- @x1: Input Int (Size = 0; Prec = 0; Scale = 0) [71776]
-- Context: SqlProvider(Sql2005)
    Model: AttributedMetaModel Build: 3.5.30729.1

```

#: SO71776Total: 87.0851

Line Total: 63.900000

```

SELECT [t0].[SalesOrderID], [t0].[SalesOrderDetailID],
       [t0].[OrderQty], [t0].[ProductID], [t0].[UnitPrice],
       [t0].[UnitPriceDiscount], [t0].[LineTotal],
       [t0].[rowguid], [t0].[ModifiedDate]
FROM [SalesLT].[SalesOrderDetail] AS [t0]
WHERE [t0].[SalesOrderID] = @x1
-- @x1: Input Int (Size = 0; Prec = 0; Scale = 0) [71780]
-- Context: SqlProvider(Sql2005)
    Model: AttributedMetaModel Build: 3.5.30729.1

```

#: SO71780Total: 42452.6519

Line Total: 873.816000

Line Total: 923.388000

At the top of this output, you can see the *SalesOrderHeader* being executed. There is only one query for *SalesOrderHeader*. Next, observe that the next query is for *SalesOrderDetail*, which occurred in the inner *foreach* loop. The *@x1* parameter is set to the ID of the current *SalesOrderHeader*, 71774. Then you can see the output from the *Console.WriteLine* statements. After that, you see another *SalesOrderDetail* query where *@x1* is set to 71776, and the output from the *foreach* loop prints again. Finally, you see the third call to *SalesOrderDetail*, which is consistent with the proven pattern that there will be a query for *SalesOrderDetail* for every *SalesOrderHeader* in the list.

Now, multiply that behavior by the number of requests for *SalesOrderHeaders* that you might see with a web application. The amount of traffic being generated to retrieve *SalesOrderDetail* records has a significant negative impact on the scalability of an application, especially in web and networked environments. The next section shows you how to resolve a problem where deferred loading impacts the scalability of your application.

Resolving Deferred Loading Problems

Deferred loading is the default *DataContext* behavior. However, you have control over how loading occurs by loading all objects immediately, loading a subset of objects immediately, or filtering associated objects. The following sections cover each of these capabilities.

Loading All Objects Immediately

If you know that you want all objects to be loaded immediately, you can turn off deferred loading. This means that all child objects will be loaded when the parent object is loaded. To implement immediate loading of all objects, you can add the following statement after instantiating the *DataContext*:

```
ctx.DeferredLoadingEnabled = false;
```

Remember to set *DeferredLoadingEnabled* to *false* prior to executing any queries. Be careful with this

option, because you could find yourself in an undesirable scenario as described in the previous section on “What Could Be Problematic about Deferred Loading?”

An important distinction to remember is that deferred loading and deferred execution are two different concepts. Deferred *execution* means that the top-level data is not requested until it is materialized by a *foreach* loop, a call to *ToList*, or any other method of materializing a deferred query. On the other hand, deferred *loading* means that associated child-level data isn’t loaded until it is requested. In other words, in deferred loading, a query can materialize and retrieve the initial data, but related data isn’t loaded until it is explicitly requested.

Immediate Loading of Selected Objects

Taking a more granular approach, you can implement immediate loading of specific child objects. The following example shows how to perform immediate loading of *SalesOrderDetail* records. It nearly duplicates the previous example, showing deferred loading, except that it configures *DataLoadOptions* for the *DataContext* and materializes the query to a list. Remember to add the *System.Data.Linq* namespace for the *DataLoadOptions* type:

```
using (var ctx = new AdventureWorksDataContext())
{
    DataLoadOptions loadOptions = new DataLoadOptions();
    loadOptions.LoadWith<SalesOrderHeader>(  
        salesOrd => salesOrd.SalesOrderDetails);
    ctx.LoadOptions = loadOptions;
    ctx.Log = Console.Out;
    var salesOrders =
        from salesOrd in ctx.SalesOrderHeaders
        select
            new
            {
                salesOrd.SalesOrderNumber,
                salesOrd.TotalDue,
                salesOrd.SalesOrderDetails
            };
    foreach (var salesOrd in salesOrders.Take(3).ToList())
    {
        Console.WriteLine(
            "#: " + salesOrd.SalesOrderNumber +
            ", Total: " + salesOrd.TotalDue);
        foreach (var salesDetail in salesOrd.SalesOrderDetails)
        {
            Console.WriteLine(
                "\tLine Total: " + salesDetail.LineTotal);
        }
    }
}
```

In the preceding example, the immediate loading code is shown in bold. You need to instantiate a *DataLoadOptions* object first. Then you use the *LoadWith* generic method, specifying the parent object type and then using a lambda to specify which child type to load. You can call *LoadWith* for each child type you want immediate loading for. Additionally, you can use *LoadWith* to immediately load children of children. You could place the following statement directly below the *LoadWith<SalesOrderHeader>* (...) statement from the previous section:

```
loadOptions.LoadWith<SalesOrderDetail>(  
    salesDetail => salesDetail.Children  
);
```



```
salesOrdDetail => salesOrdDetail.Product);
```

Before you're done, assign the *DataLoadOptions* instance to the *LoadOptions* property of the *DataContext*. You can't modify the load options once load options has been assigned to the *DataContext*, and you would receive an exception if you tried. Also, notice that I materialized *salesOrders* in the outer *foreach* loop with a call to *ToList*. This is required because immediate loading won't work unless you completely materialize the query. The following output is the same whether you set *DeferredLoading* enabled to *false* or use the *LoadOptions* approach:

```
SELECT TOP (3) [t0].[SalesOrderNumber],
      [t0].[TotalDue], [t0].[SalesOrderID]
FROM [SalesLT].[SalesOrderHeader] AS [t0]
-- Context: SqlProvider(Sql2005)
Model: AttributedMetaModel Build: 3.5.30729.1
```

```
SELECT [t0].[SalesOrderID], [t0].[SalesOrderDetailID],
      [t0].[OrderQty], [t0].[ProductID], [t0].[UnitPrice],
      [t0].[UnitPriceDiscount], [t0].[LineTotal],
      [t0].[rowguid], [t0].[ModifiedDate],
      [t1].[ProductID] AS [ProductID2], [t1].[Name],
      [t1].[ProductNumber], [t1].[Color], [t1].[StandardCost],
      [t1].[ListPrice], [t1].[Size], [t1].[Weight],
      [t1].[ProductCategoryID], [t1].[ProductModelID],
      [t1].[SellStartDate], [t1].[SellEndDate],
      [t1].[DiscontinuedDate], [t1].[ThumbNailPhoto],
      [t1].[ThumbnailPhotoFileName], [t1].[rowguid] AS [rowguid2],
      [t1].[ModifiedDate] AS [ModifiedDate2]
FROM [SalesLT].[SalesOrderDetail] AS [t0]
INNER JOIN [SalesLT].[Product] AS [t1]
  ON [t1].[ProductID] = [t0].[ProductID]
WHERE [t0].[SalesOrderID] = @x1
-- @x1: Input Int (Size = 0; Prec = 0; Scale = 0) [71774]
-- Context: SqlProvider(Sql2005)
Model: AttributedMetaModel Build: 3.5.30729.1
```

```
SELECT [t0].[SalesOrderID], [t0].[SalesOrderDetailID],
      [t0].[OrderQty], [t0].[ProductID], [t0].[UnitPrice],
      [t0].[UnitPriceDiscount], [t0].[LineTotal],
      [t0].[rowguid], [t0].[ModifiedDate],
      [t1].[ProductID] AS [ProductID2], [t1].[Name],
      [t1].[ProductNumber], [t1].[Color], [t1].[StandardCost],
      [t1].[ListPrice], [t1].[Size], [t1].[Weight],
      [t1].[ProductCategoryID], [t1].[ProductModelID],
      [t1].[SellStartDate], [t1].[SellEndDate],
      [t1].[DiscontinuedDate], [t1].[ThumbNailPhoto],
      [t1].[ThumbnailPhotoFileName], [t1].[rowguid] AS [rowguid2],
      [t1].[ModifiedDate] AS [ModifiedDate2]
FROM [SalesLT].[SalesOrderDetail] AS [t0]
INNER JOIN [SalesLT].[Product] AS [t1]
  ON [t1].[ProductID] = [t0].[ProductID]
WHERE [t0].[SalesOrderID] = @x1
-- @x1: Input Int (Size = 0; Prec = 0; Scale = 0) [71776]
-- Context: SqlProvider(Sql2005)
Model: AttributedMetaModel Build: 3.5.30729.1
```

```
SELECT [t0].[SalesOrderID], [t0].[SalesOrderDetailID],
      [t0].[OrderQty], [t0].[ProductID], [t0].[UnitPrice],
      [t0].[UnitPriceDiscount], [t0].[LineTotal], [t0].[rowguid],
```

```

[t0].[ModifiedDate], [t1].[ProductID] AS [ProductID2],
[t1].[Name], [t1].[ProductNumber], [t1].[Color],
[t1].[StandardCost], [t1].[ListPrice], [t1].[Size],
[t1].[Weight], [t1].[ProductCategoryID],
[t1].[ProductModelID], [t1].[SellStartDate],
[t1].[SellEndDate], [t1].[DiscontinuedDate],
[t1].[ThumbNailPhoto], [t1].[ThumbNailPhotoFileName],
[t1].[rowguid] AS [rowguid2],
[t1].[ModifiedDate] AS [ModifiedDate2]
FROM [SalesLT].[SalesOrderDetail] AS [t0]
INNER JOIN [SalesLT].[Product] AS [t1]
ON [t1].[ProductID] = [t0].[ProductID]
WHERE [t0].[SalesOrderID] = @x1
-- @x1: Input Int (Size = 0; Prec = 0; Scale = 0) [71780]
-- Context: SqlProvider(Sql2005)
Model: AttributedMetaModel Build: 3.5.30729.1

```

```

#: SO71774, Total: 972.7850
    Line Total: 356.898000
    Line Total: 356.898000
#: SO71776, Total: 87.0851 Line Total: 63.900000
#: SO71780, Total: 42452.6519
    Line Total: 873.816000
    Line Total: 923.388000

```

This time, the output shows that all queries occur at one time, and then you get the output. The queries ran as soon as the query materialized with the call to *ToList*.

Filtering Child Objects

Sometimes you don't want every child object in an association to load. For the initial reason that I gave for immediate loading of too many objects, you have the ability to load only those child objects that you really need.

To filter which child objects are loaded, you can use the *AssociateWith* method of the *DataLoadOptions*. The following example shows how to use *AssociateWith* to filter only *SalesOrderDetail* records whose total is less than 400:

```

DataLoadOptions loadOptions = new DataLoadOptions();
loadOptions.AssociateWith<SalesOrderHeader>(<
    salesOrd =>
        salesOrd.SalesOrderDetails.Where(
            detail => detail.LineTotal < 400));
ctx.LoadOptions = loadOptions;

```

The preceding code replaces the *LoadWith* code in the previous example. The difference is that it uses an *AssociateWith* method instead of a *LoadWith*. The *AssociateWith* type parameter is a *SalesOrderHeader*, the parent, and the lambda references *SalesOrderDetails*, the child collection. The *Where* method specifies the filter to use when retrieving child references. Similar to *LoadWith*, *AssociateWith* causes immediate loading of child records, but filters the results as demonstrated in the following output:

```

SELECT TOP (3) [t0].[SalesOrderNumber],
[t0].[TotalDue], [t0].[SalesOrderID]
FROM [SalesLT].[SalesOrderHeader] AS [t0]
-- Context: SqlProvider(Sql2005)

```

Model: AttributedMetaModel Build: 3.5.30729.1

```
SELECT [t0].[SalesOrderID], [t0].[SalesOrderDetailID],
       [t0].[OrderQty], [t0].[ProductID], [t0].[UnitPrice],
       [t0].[UnitPriceDiscount], [t0].[LineTotal],
       [t0].[rowguid], [t0].[ModifiedDate]
FROM [SalesLT].[SalesOrderDetail] AS [t0]
WHERE ([t0].[LineTotal] < @p0) AND ([t0].[SalesOrderID] = @x1)
-- @p0: Input Decimal (Size = 0; Prec = 38; Scale = 6) [400]
-- @x1: Input Int (Size = 0; Prec = 0; Scale = 0) [71774]
-- Context: SqlProvider(Sql2005)
```

Model: AttributedMetaModel Build: 3.5.30729.1

```
SELECT [t0].[SalesOrderID], [t0].[SalesOrderDetailID],
       [t0].[OrderQty], [t0].[ProductID], [t0].[UnitPrice],
       [t0].[UnitPriceDiscount], [t0].[LineTotal],
       [t0].[rowguid], [t0].[ModifiedDate]
FROM [SalesLT].[SalesOrderDetail] AS [t0]
WHERE ([t0].[LineTotal] < @p0) AND ([t0].[SalesOrderID] = @x1)
-- @p0: Input Decimal (Size = 0; Prec = 38; Scale = 6) [400]
-- @x1: Input Int (Size = 0; Prec = 0; Scale = 0) [71776]
-- Context: SqlProvider(Sql2005)
```

Model: AttributedMetaModel Build: 3.5.30729.1

```
SELECT [t0].[SalesOrderID], [t0].[SalesOrderDetailID],
       [t0].[OrderQty], [t0].[ProductID], [t0].[UnitPrice],
       [t0].[UnitPriceDiscount], [t0].[LineTotal],
       [t0].[rowguid], [t0].[ModifiedDate]
FROM [SalesLT].[SalesOrderDetail] AS [t0]
WHERE ([t0].[LineTotal] < @p0) AND ([t0].[SalesOrderID] = @x1)
-- @p0: Input Decimal (Size = 0; Prec = 38; Scale = 6) [400]
-- @x1: Input Int (Size = 0; Prec = 0; Scale = 0) [71780]
-- Context: SqlProvider(Sql2005)
```

Model: AttributedMetaModel Build: 3.5.30729.1

```
#: SO71774, Total: 972.7850
   Line Total: 356.898000
   Line Total: 356.898000
#: SO71776, Total: 87.0851
   Line Total: 63.900000
#: SO71780, Total: 42452.6519
   Line Total: 323.994000
   Line Total: 149.874000
```

I removed several *Line Total* entries from the third set of output to shorten the listing, but you can see that the output conforms to the filter. Also, notice that the *where* clause in the queries includes the same filter specified in the lambda to the *Where* method in the *AssociateWith* method call.

Immediate Loading and the Using Statement

Going back to a point I made earlier in the chapter, you might want to implement the *using* statement for greater scalability because it disposes the DataContext. However, that raises a problem when you're working with child collections and deferred loading that I'll need to explain a workaround for.

When implementing your query in a business object with a *using* statement, the collection of objects returns to the caller, typically UI layer code, and the *using* statement disposes the DataContext. If your UI layer subsequently attempts to access a child object, you'll receive an exception. The exception is caused

by the fact that a deferred load is occurring but the DataContext has already closed.

You can solve this problem with immediate loading. For example, if you are showing orders with *SalesOrderHeader* and need to also display line items with *SalesOrderDetail*, you should perform an immediate load of *SalesOrderDetail* at the time you make the query in your business object. This way, you won't receive an exception, because the child objects are already loaded when you pass the collection from your business logic back to your UI.

Modularized Composition and Deferred Execution

Sometimes you have reusable code that is a LINQ query and you would like to compose that query in any number of other queries. To do so, you'll want to use the same DataContext. To accomplish this goal, you can create a separate method, modularizing the query. You would pass in the DataContext from the calling method, and the method would return a result.

One scenario would be if you receive information about a customer, such as first name and last name, and needed to get the customer's ID for a subsequent query. Perhaps you can imagine a situation where this code is reusable for other queries that need to get a customer ID. The following code demonstrates how to get a sales order for a customer with only the customer's name as input:

```
var firstName = "David";
var lastName = "Hodgson";

using (var ctx = new AdventureWorksDataContext())
{
    var custID = GetCustomerID(ctx, firstName, lastName);

    var orders =
        from order in ctx.SalesOrderHeaders
        where order.CustomerID == custID
        select order;

    foreach (var order in orders)
    {
        Console.WriteLine(
            "Order #: " + order.SalesOrderNumber +
            ", Customer: " + order.Customer.FirstName +
            " " + order.Customer.LastName);
    }
}
```

This example hard-codes *firstName* and *lastName*, simulating input parameters. The call to *GetCustomerID* is the modularized query that I was referring to. The arguments passed are the DataContext, *firstName*, and *lastName*, and it returns the customer's ID. The query subsequently uses this ID to get the requested sales order. Here's the code for *GetCustomerID*:

```
private static int GetCustomerID(
    AdventureWorksDataContext ctx,
    string firstName,
    string lastName)
{
    var custID =
        (from cust in ctx.Customers
         where cust.FirstName == firstName &&
             cust.LastName == lastName
```

```

        select cust.CustomerID)
        .FirstOrDefault();

    return custID;
}

```

This code is the same as any other LINQ to SQL query, except that it uses the *DataContext* passed in by the caller. The benefits are that you can reuse the code, and you can potentially reuse *DataContext* objects that are already loaded. There are many *ifs* in this scenario, and you'll have to evaluate it to see if it works for you, but I've found it handy on occasion.

You could also implement the algorithm in a fully deferred mode. The following example shows how to get a customer reference that only materializes when the caller chooses a fully deferred mode query:

```

var firstName = "David";
var lastName = "Hodgson";

using (var ctx = new AdventureWorksDataContext())
{
    ctx.Log = Console.Out;

    var customer = GetCustomerDeferred(ctx, firstName, lastName);

    var orders =
        from order in ctx.SalesOrderHeaders
        where order.CustomerID == customer.Single().CustomerID
        select order;

    foreach (var order in orders)
    {
        Console.WriteLine(
            "Order #: " + order.SalesOrderNumber +
            ", Customer: " + order.Customer.FirstName +
            " " + order.Customer.LastName);
    }
}

```

Notice that the call to *GetCustomerDeferred* results in an *IQueryable<Customer>* as demonstrated by the *where* clause of the query that invokes the *Single* operator on *customer* to reference the *CustomerID*. The *GetCustomerDeferred* method is implemented as follows:

```

private static IQueryable<Customer> GetCustomerDeferred(
    AdventureWorksDataContext ctx,
    string firstName,
    string lastName)
{
    var customer =
        from cust in ctx.Customers
        where cust.FirstName == firstName &&
            cust.LastName == lastName
        select cust;

    return customer;
}

```

The important thing to notice is that there are no calls of *ToList* and the like to cause materialization of the query—that is left to the caller.

Summary

A few discussions concerned issues of architecture and design that you'll need at one time or another when doing LINQ programming. Most of the scenarios centered around database operations, which are commonly used. Be careful about trying to use RAD tools such as LinqDataSource in an enterprise environment when you really need to implement an n-layer system for greater maintainability. In many cases, you can achieve UI RAD with ObjectDataSource and still keep your n-layer architecture.

This chapter proposed an n-layer architecture that assumed a LINQ to SQL DataContext or LINQ to EntitiesObjectContext as your DAL. The approach doesn't necessarily advocate this one architecture in all scenarios, but does illuminate an effective way to achieve an n-layer architecture compromise that can be effective and save you a lot of time.

You also learned how to manage data concurrency conflicts, both pessimistic and optimistic, where LINQ provides built-in support for optimistic concurrency management. A couple other important design issues surfaced: deferred execution and deferred loading. You learned how both of these features help you with performance in most cases, but also have hidden potential to hurt performance in other scenarios. The sections that covered deferred execution and deferred loading provided ways to handle these potential problems effectively.

CHAPTER 12

Concurrent Programming with Parallel LINQ (PLINQ)

As has been the norm for many years, hardware and software advances continue at a steady pace. More recently, hardware design has shifted in the direction of multicore CPUs and multiprocessing systems. This change is dramatic for software because it also leads us in the direction of a paradigm shift in how we develop algorithms. Much code we write today is single-threaded, and our performance gains come in the form of faster hardware or efficient single-threaded algorithms. Moving into the future, we need to embrace a concurrent programming paradigm, where multiple threads of execution operate on systems with multicore and multiprocessor hardware.

While Windows and .NET have supported concurrent programming with multithreading APIs for many years, it wasn't as easy to implement as it could be. Because of the advances in multicore and multiprocessing hardware, Microsoft is building software that will make it easier for programmers to write concurrent software. LINQ is particularly important in concurrency because of its declarative style of development; you tell LINQ what to do, rather than how to do it. Building upon this, Microsoft is introducing a new API for concurrent programming called the *Parallel Extensions to the .NET Framework*.

Germane to the subject of this book, Parallel Extensions to the Microsoft Framework includes Parallel LINQ (PLINQ)—a LINQ extension that allows you to run queries on multiple threads. This opens the potential to let your programs run faster and make better use of computing resources. The following sections will outline system requirements for setting up PLINQ, show you how to use PLINQ to speed up your applications, and demonstrate how to manage exceptions that might occur during a PLINQ query.

System Requirements and Setup for PLINQ

At the time of this writing, PLINQ is available as a download from the MSDN Parallel Computing Developer Center at <http://msdn.microsoft.com/en-us/concurrency/default.aspx>. It's part of the Parallel Extensions to the .NET Framework package. Although there is more functionality than just PLINQ in the Parallel Extensions to the .NET Framework, I'll refer to the whole package as "PLINQ." You'll need Windows Server 2003, Windows Vista, or Windows XP as your operating system. You'll also need the .NET Framework 3.5 and VS 2008 is recommended.

The PLINQ setup program adds the assembly, *System.Threading.dll*, to the Global Assembly Cache (GAC) and makes the assembly's reference available via the VS 2008 References window. You can also find the PLINQ assembly in the installation directory, which is some version of Parallel Extensions, depending on the version you install, under the Program Files folder.

TIP When testing an application written in PLINQ on multiple machines, you aren't required to install the whole Parallel Extensions to the .NET Framework package on each machine. Instead, you can copy just *System.Threading.dll*, which is just another .NET 3.5 assembly, to the same folder where your executable assembly resides, and let the normal .NET assembly binding process find it.

Whenever you build an application with PLINQ, you'll need to add a project reference to the *System.Threading.dll* assembly. If you're using VS 2008, you should right-click on the project, select Add Reference, and pick *System.Threading.dll* from the list of assemblies on the .NET tab. If you're using another build tool or doing a quick compile on the command line, you should add the reference to the *System.Threading.dll* assembly in the installation directory.

In your code, remember to add a *using* declaration for the *System.Threading* namespace. After setup, you'll be ready to code, which you can learn about in the next section.

Running PLINQ Queries

PLINQ extends LINQ to Objects and LINQ to XML, allowing queries to execute on multiple threads. With PLINQ, you get the benefit of declarative code with common LINQ syntax and the additional benefit of speeding up queries. To use PLINQ, you'll need to know different ways to run queries. The sections that follow examine PLINQ types and show you how to parallelize an existing LINQ to Objects query and how to preserve ordering of a query.

Examining PLINQ Types

As with other query providers, PLINQ has a static class that implements all of its extension methods that extend *IEnumerable<T>*, *ParallelEnumerable*. PLINQ has an additional class for *IParallelEnumerable* extension methods named *ParallelQuery*. Just as with other LINQ providers, your learning curve is minimal. You can also use C# query syntax combined with PLINQ extension methods. The Appendix lists available extension methods with examples of their usage.

ParallelEnumerable and *ParallelQuery* are types in the *System.Linq* namespace, which you should add to your code with a *using* declaration. Don't forget a *using* declaration for the *System.Threading* namespace either, which contains the *AggregateException* class used in the last section of this chapter on exception handling.

Parallelizing a Query

The example I'll use in this chapter executes a method that adds CPU load to the system. This simulates a typical scenario where you might want to use PLINQ—splitting up long-running tasks among multiple threads to take advantage of either multicore or multiprocessor hardware. Here's the *LongProcess* method, designed to simulate a heavy algorithm:

```
private static int LongProcess(int val)
{
    for (int i = 0; i < 1000; i++)
    {
        Math.Sqrt(val);
    }

    return (int)Math.Sqrt(val);
}
```

The results of this algorithm are not important, but the purpose is. If you have an algorithm that takes time or puts a heavy load on a CPU, such as the preceding one, you could have a good candidate for concurrency. It could be a mathematical calculation, manipulation of string data, or a parsing routine. There are many tasks you can perform on elements of a collection using LINQ. Here's a *List* of *int* that examples in this chapter will operate on:

```
private static List<int> m_input =
    Enumerable
        .Range(0, byte.MaxValue)
        .ToList();
```


As you learned in [Chapter 2](#), *Enumerable* is the static class that implements *IEnumerable<T>* extension methods. *Range* is an extension method that returns a list of numbers from the value of its first parameter to the value of the second parameter. *Range* gives you a quick way to build a collection of values with a small amount of code to support code examples or testing.

With a method in place for adding CPU load and a collection to query, the LINQ query for the demos in this chapter is shown next:

```
var result =  
    from val1 in m_input  
    from val2 in m_input  
    select LongProcess(val1 + val2);
```

This query projects on each element of the input collection, running each value through the *LongProcess* method. To create even more load, the query implements a *select many* operation, effectively creating a cross-join of *m_input* with itself.

Implementing PLINQ is as easy as composing the data source with the PLINQ *AsParallel* method. Here's how you could modify the preceding query:

```
result =  
    from val1 in m_input.AsParallel()  
    from val2 in m_input  
    select LongProcess(val1 + val2);
```

All the preceding query does is append *AsParallel()* to the first *from* clause data source, *m_input*. Now, PLINQ will perform the query using multiple threads. If you put some timer code around each algorithm, you'll be able to see the difference in speed between queries. A later section of this chapter will show you a more extensive example with timers.

Is this not cool?! But let's inject a dose of reality to balance expectations of what speed increases you really get with PLINQ.

Two Threads Are Not Always Better than One

At this point, you might be thinking that this is great—you can simply add *AsParallel* to every query, and your code will run faster—but this is not true. PLINQ can speed up your code on multicore and/or multiprocessor CPUs, but will have the opposite effect on single-processor, single-core CPUs. That's right; PLINQ will slow down your code if it doesn't run on hardware that can take advantage of it.

The rationale behind the single-core/single-processor slow down is that it takes overhead to launch new threads. Since the single-core processor can only perform one operation at a time, the overhead adds to the total amount of time to run the query. However, with multicore/multiprocessors, the threads really do run in parallel; the overhead of creating the threads still exists, but you still see the speed up from the threads running simultaneously.

Further discussion will assume that you are aware of single-core/single-processor performance behavior, without my needing to qualify each statement I make.

Although a multicore/multiprocessor machine will run the algorithm faster, you still won't see time savings as a division of cores and/or processors. For example, if you have one processor with two cores, PLINQ has two threads to run the query with. If the query runs for two seconds sequentially, you might be mistakenly looking for a speed increase of only one second on two threads. This won't happen because

there is overhead involved with managing multiple threads, meaning that you might see the query take more than one second with two threads. Amdahl's Law defines this phenomenon and explains why two threads will never run twice as fast, because you must add the overhead of thread management and some operations that must run sequentially to the time it takes to complete the work.

Related to overhead, the next section describes some scenarios affecting where you place the *AsParallel* method.

Choosing What to Parallelize

The previous example uses *AsParallel* on the first data source only, which I'll refer to as the *outer loop*. The second *from* clause, which I'll call the *inner loop*, doesn't call *AsParallel* on its data source. You might think that adding *AsParallel* to both the outer loop and the inner loop might help, as shown in the following example, but it really doesn't help:

```
var result =  
    from val1 in m_input.AsParallel()  
    from val2 in m_input.AsParallel()  
    select LongProcess(val1 + val2);
```

This query runs slower than the one with *AsParallel* on the outer loop only. Think about Amdahl's law again. There are N threads that can run on the system, but *AsParallel* causes the query to run with even more threads. When N threads have been created and are running, the rest of the threads are overhead, which slows down the query.

Another feature of *AsParallel* is that you can manage the number of threads, which is covered next.

Setting the Degree of Parallelism

AsParallel has an overload that takes an *int* parameter named *degreeOfParallelization*. You can set this value and tweak it to find the best settings that work on your system. Here's an example of how to use the *degreeOfParallelism* parameter to set the number of threads to 2:

```
var result =  
    from val1 in m_input.AsParallel(2)  
    from val2 in m_input  
    select LongProcess(val1 + val2);
```

You can set the number of threads as needed to get the best performance from this one query.

TIP Rather than hard-code the value, you could also create a configuration file setting and then use the configuration file entry to run the number of threads; this would let you customize performance for different machines without a rebuild of the application.

The next section looks at how *AsParallel* handles result ordering.

Ordering PLINQ Results

One of the principles of multithreaded programming is that you should never rely on the execution order of parallel threads. On one run, each thread will execute in a specific order, and subsequent runs could all result in different execution orders. This principle holds true, even if you use the *orderby* clause in a PLINQ query. To overcome this problem, you can compose your data source with the *AsOrdered* method. Here's an example that preserves the ordering of a query:

```
(from val in
    m_input
    .AsParallel()
    .AsOrdered()
orderby val descending
select LongProcess(val))
.ToList()
.ForEach(
    valOut => Console.WriteLine(valOut)
);
```

You first call *AsParallel* on the data source and then call *AsOrdered* as just shown. The results will be ordered according to the *orderby* clause. Be aware that *AsOrdered* will slow down your query because of the extra work it must do to overcome the issue of unpredictable ordering produced by *AsParallel*. Regardless, *AsOrdered* should still be faster than sequential.

The following section discusses a couple more PLINQ methods related to *AsParallel* and *AsOrdered*.

Additional PLINQ Methods

Both *AsParallel* and *AsOrdered* have opposite methods named *AsSequential* and *AsUnordered*. *AsSequential* will ensure that a query runs sequentially, and *AsUnordered* will ensure that ordering does not occur.

You might be wondering why, if you coded a query with *AsParallel* and *AsOrdered*, you would use *AsSequential* or *AsUnordered*. After all, sequential is the default query concurrency mode, and parallel queries are naturally unordered.

To answer this, think back to [Chapter 9](#), where you learned that extension methods were composable. You can have multiple branches of logic that determine how a query is composed. Perhaps one part of the algorithm figures out that the query should run in parallel, but another part of the algorithm finds a condition that requires the query to be sequential. In that case, making the query sequential via *AsSequential* could undo an earlier decision. Here's an example that simulates this scenario:

```
var parList =
    from val1 in m_input.AsParallel()
    from val2 in m_input
    select LongProcess(val1 + val2);

var sequentialQuery = parList.AsSequential();
```

The first statement, assigning a parallel PLINQ query to *parList*, could happen earlier in an algorithm. Later, the statement removing parallelism from the query, assigning the new query to *sequentialQuery*, might be invoked because of some logic.

In another scenario, supported in [Chapter 11](#), you learned that deferred execution allows you to modularize queries. What if you had part of a query built in one method that might have ordered query results with *AsOrdered* and then discovered in the calling algorithm that the query shouldn't be ordered? In that case, you could use *AsUnordered* to undo the earlier ordering. The following example simulates this scenario:

```
var ordList =
    from val in
        m_input
```

```
.AsParallel()  
.AsOrdered()  
orderby val descending  
select LongProcess(val);
```

```
var unorderedQuery = ordList.AsUnordered();
```

In the first statement, *ordList* becomes an ordered query. Later, as the result of logic that determines the result shouldn't be ordered, the second statement removes ordering.

Both the *AsSequential* and *AsUnordered* give you extra flexibility in managing the parallelism in your application if you should ever need it. Next, we'll look at the actual time it takes to run different PLINQ queries.

PLINQ Performance Testing

When you're doing performance testing, there are often multiple factors to consider. For example, what type of CPU does the test machine have? How does the test machine compare with the production system? Is there anything in the query or algorithm that could differ between machines? How much does other hardware on the machine affect what is being measured? The list can go on, but the point is that a single example in a book such as this will never be as accurate as your own assessment of these conditions and testing in your own environment. That said, I can show you a couple techniques for testing and some generalizations for evaluating different query techniques. Here's an example of performance testing, followed by some guidance on hardware considerations.

Analyzing Performance

Listing 12-1 is a program that tests multiple types of queries. It iterates through a list of delegates to various methods, each with their own PLINQ implementations, and measures the amount of time in seconds that each method takes. To minimize the impact of CLR overhead from loading, Just In Time (JIT) compilation, and the like, I ran each method 10 times. My goal was not to be absolutely precise, but to draw some generalizations on performance and give you a quick-and-dirty example of how you might put a test like this together yourself.

Listing 12-1 Performance Test of Multiple PLINQ Queries

```
using System;  
using System.Collections.Generic;  
using System.Diagnostics;  
using System.Linq;  
using System.Threading;  
  
class Program  
{  
    static void Main()  
    {  
        Console.WriteLine("\nTest Various Queries:\n");  
  
        new List<Func<List<int>>>>  
        {  
            RunSequential,  
            RunDoubleLoopParallel,  
            RunOuterLoopParallel,  
        }
```

```

        RunInnerLoopParallel
    }
    .ForEach(
        func =>
        {
            var stopWatch = new Stopwatch() ;
            stopWatch.Start();

            func();

            stopWatch.Stop();

            Console.WriteLine(
                func.Method.Name + " : " +
                stopWatch.Elapsed.TotalMilliseconds) ;
        }
    );

    Console.ReadKey();
}

private static List<int> RunSequential()
{
    var result =
        from val1 in m_input
        from val2 in m_input
        select LongProcess(val1 + val2);

    return result.ToList();
}

private static List<int> RunOuterLoopParallel()
{
    var result =
        from val1 in m_input.AsParallel()
        from val2 in m_input
        select LongProcess(val1 + val2);
    return result.ToList();
}

private static List<int> RunDoubleLoopParallel()
{
    var result =
        from val1 in m_input.AsParallel()
        from val2 in m_input.AsParallel()
        select LongProcess(val1 + val2);

    return result.ToList();
}

private static List<int> RunInnerLoopParallel()
{
    var result =
        from val1 in m_input
        from val2 in m_input.AsParallel()
        select LongProcess(val1 + val2);
    return result.ToList();
}
}

```

The *Main* method in [Listing 12-1](#) iterates through a list of delegates, timing the invocation of each of the methods referred to by the delegates. You've seen all of the preceding examples, except for the query in the *RunInnerLoopParallel*, which adds *AsParallel* to the inner loop only. The timer is a *StopWatch* class, which is helpful because it enables more precision than calculating *Timespan* instances from the *DateTime* class.

I ran this code with three different CPU configurations: single-core/single-processor, dual-core/single-processor, and dual-core/dual-processor machines. I also ran the program several times on each machine. The results differed between runs, but were generally consistent for each configuration. Here are the results on the single-core/single-processor configuration:

```
RunSequential: 305.895
RunDoubleLoopParallel: 265.9555
RunOuterLoopParallel: 261.8321
RunInnerLoopParallel: 270.95
```

You can see that the PLINQ queries, with *Parallel* suffix, are a little faster than the sequential query, with *Sequential* suffix. However, you'll often see better performance with a sequential query on a single-core/single-processor machine. That's because of the overhead associated with running multiple threads and only a single core/processor to do the thread switching. Next, here are the results of the dual-core/single-processor system:

```
RunSequential: 303.5567
RunDoubleLoopParallel: 167.9749
RunOuterLoopParallel: 166.9775
RunInnerLoopParallel: 297.1983
```

In this example, PLINQ generally performed better than the sequential query. The *RunOuterLoopParallel* performed better than the other queries, and *RunInnerLoopParallel* shows that there was more overhead than normal when trying to parallelize the inner loop. You'll occasionally see one algorithm run slightly faster than another when the results are close. That's the nature of multithreaded programming because a number of factors affect which threads run, when a thread runs, and which processor a thread is assigned to. Overall, the differences can be striking, especially comparing well-designed PLINQ queries with sequential queries, such as the fact that *RunDoubleLoopParallel* and *RunOuterLoopParallel* run almost twice as fast as *RunSequential*. Here are the results from the dual-core/dual-processor system:

```
RunSequential: 251.5928
RunDoubleLoopParallel: 126.905
RunOuterLoopParallel: 124.316
RunInnerLoopParallel: 257.5836
```

The results here are consistent with the dual-core/single-processor example. In this case, the performance of *RunOuterLoopParallel* was twice as good as that of *RunSequential*. As you can see, different CPUs can affect the performance of your application. The next section discusses hardware more and introduces issues you'll need to think about.

Considering the Impact of Hardware

The difference in milliseconds for each result, in the previous section, is mostly due to the fact that the processors on each machine were of different classes and clock speeds. For example, the single-

core/single-processor CPU was an Intel Pentium 4 with a 2.80-GHz clock, the dual-core/single-processor CPU was an Intel Core 2 Duo with 1.4 GHz clock, and the multicore/ multiprocessor machine had two Intel Xeon CPUs with 2.20 GHz clocks. While the PLINQ API makes it easy to write concurrent code, remember that you must understand the hardware that your application is running on. Consideration of hardware is part of the paradigm shift I referred to at the beginning of this chapter, which is something you’ve rarely had to consider with high-level, single-threaded, object-oriented software development.

Besides the CPU, other hardware on your system can affect performance of a query. Consider the following example, which performs a sequential and parallel query on all files in a list of subdirectories. Because of the *Directories* class, you’ll need to add a *using* declaration for *System.IO* in the following example, and *using System.Diagnostics* is required for the *StopWatch*:

```
string parentDir =
    Environment.GetFolderPath(
        Environment.SpecialFolder.System);
var stopWatch1 = new Stopwatch();
stopWatch1.Start();

var sequentialFiles =
    (from dir in Directory.GetDirectories(parentDir)
     from file in Directory.GetFiles(dir)
     select file)
    .ToList();

    stopWatch1.Stop();

Console.WriteLine(
    "Sequential File Query: " +
    stopWatch1.Elapsed.TotalMilliseconds);

var stopWatch2 = new Stopwatch();
stopWatch2.Start();

var parallelFiles =
    (from dir in Directory.GetDirectories(parentDir)
     from file in Directory.GetFiles(dir)
     select file)
    .ToList();

stopWatch2.Stop();

Console.WriteLine(
    "Parallel File Query: " +
    stopWatch2.Elapsed.TotalMilliseconds);
```

Notice that the query calls *GetDirectories* in the outer loop to get a list of directories from the file system and then calls *GetFiles* in the inner loop to get the file list for each directory. Since the query uses the *System* folder, this is a lot of file I/O. Here are the results from a dual-core/single-processor machine:

```
Sequential File Query: 42.0062
Parallel File Query: 44.1543
```

At first, it might seem logical to be confused about the lack of difference in performance between sequential and parallel queries. However, you’ll see that there is reason for this. Before I reveal that reason, here’s another example on the dual-core/multiprocessor system with the same algorithm:

These results are a little surprising because they show a sequential query running faster than a parallel query—on a machine with two dual-core processors! If you recall from my opening discussion, multiple factors are in play when testing a multithreaded system. In this case, the reason for the upside-down results is because of the type of work the query is doing. More specifically, the file I/O is so excessive that any gains from parallelization go unnoticed. In some cases, the parallel query will be slightly faster than the sequential query, but the results are not consistent because they depend on factors that are more associated with disk I/O than the query itself.

The lesson to be learned here is that hardware conditions matter. You should test your system and challenge your assumptions (often based on the sequential paradigm) before making design decisions for concurrent code resulting in algorithms that don't perform the way you expect.

Handling PLINQ Exceptions

A PLINQ query can encounter exceptions on any number of threads at any time. Unlike with a normal sequential exception, you might not see the exception immediately. When an exception occurs in one thread, PLINQ will not launch any more threads and will return when all running threads at the time of the exception have completed. This means that you could receive multiple exceptions. This section will explain what happens to those exceptions and how you can handle them.

PLINQ uses an *AggregateException* in the *System.Threading* namespace to collect all of the exceptions that occur in a query's threads. *AggregateException* has a collection named *InnerExceptions* of all of the exceptions thrown during the query. This gives you access to all of the information for each exception that you would normally have, including stack traces. The following example shows you how to handle *AggregateException* for a query that throws multiple exceptions:

```
var annualSales =  
    new List<decimal>  
    {  
        10000,  
        20000,  
        30000  
    };  
  
int salesPeriod = 0;  
  
try  
{  
    var monthlyAverages =  
        (from sale in annualSales.AsParallel(2)  
         select  
             new  
             {  
                 Amount = sale,  
                 AvgMonth = sale / salesPeriod  
             })  
        .ToList();  
}  
catch (AggregateException ae)  
{  
    foreach (var ex in ae.InnerExceptions)  
    {
```



```
        Console.WriteLine($"{ex}");
    };
}
```

The preceding example intentionally causes *DivideByZeroException* to be thrown. The *catch* handler for *AggregateException* iterates through each member of the *InnerExceptions* collection to process each exception. Notice the plural on *InnerExceptions*, which can be confusing because *Exception*, the base class of all .NET Exceptions, has an *InnerException* property, which is singular. That means that *AggregateException* has both *InnerException* and *InnerExceptions* properties. While *InnerExceptions* contains all of the exceptions thrown during the query, *InnerException* is *null*. You might notice this difference if you use *InnerException* in a *foreach* loop and the code won't compile, meaning that you should have used *InnerExceptions* instead.

The preceding example uses the *degreeOfParallelization* parameter of *AsParallel* to illustrate the behavior of PLINQ exception handling. If you set it to 2, only two threads will be running at a time. Therefore, the preceding example never gets to the third value of the *annualSales* list, and you'll only see two exceptions in *InnerExceptions*, proving that PLINQ stops all other threads on an exception, but threads that are already running will continue.

Summary

PLINQ lets you run LINQ to Objects queries in parallel. You saw how to set up PLINQ and what its system requirements are.

You learned how to implement a PLINQ query with *AsParallel*. As you progressed, you encountered situations that impacted the performance of a query. The chapter also covered the *AsOrdered* extension method, enabling you to maintain ordering when queries run on multiple threads.

Since performance is an important reason why you would use PLINQ, you were introduced to some techniques for testing PLINQ performance. Remember that the results of a query will not always be what you expect, and it's important to consider issues that were discussed in this chapter.

For a reliable system, you need to be able to handle exceptions. The last section showed you how to handle exceptions and how PLINQ behaves when one or more exceptions are thrown during a query.

Congratulations! You've now reached the end of this book and have many tools in your engineering backpack for working with LINQ. I wish all the best of luck to you.

—Joe Mayo

APPENDIX

Standard Query Operator Reference

The mechanism behind C# and VB.NET language query syntax is extension methods. The query syntax clauses cover a subset of a set of extension methods called the *Standard Query Operators*. Each LINQ provider is based on a static class that implements extension methods for each of the Standard Query Operators. That is, LINQ to Objects Standard Query Operators are extension methods in the *Enumerable* class, and LINQ to SQL Standard Query Operators are extension methods in the *Queryable* class. This appendix contains examples for all of the Standard Query Operators.

Setting Up the Examples

All of the examples in this appendix are based on the same set of source data. This will make it easier to concentrate on the example itself, rather than on the infrastructure to make it work. The following sections describe the setup for LINQ to Objects and LINQ to SQL.

Setting Up LINQ to Objects

The data source for LINQ to Objects examples is made of a set of classes and collections. One of the primary classes is the *Reviewer* class, which represents a person who creates written reviews on products, shown next:

```
public class Reviewer :
    IEqualityComparer<Reviewer>,
    IComparer<string>
{
    public string Name { get; set; }
    public string Specialty { get; set; }
    public List<ProductReview> Reviews { get; set; }

    public bool Equals(Reviewer x, Reviewer y)
    {
        return x.Name == y.Name;
    }

    public int GetHashCode(Reviewer obj)
    {
        return obj.Name.GetHashCode();
    }

    public int Compare(string x, string y)
    {
        return x.ToLower().CompareTo(y.ToLower());
    }
}
```

The *Reviewer* class implements the *IEqualityComparer<Reviewer>* and *IComparer<string>* interfaces, which are instrumental in showing examples of Standard Query Operators with overloads with these interfaces as parameter types. Another helper class, *ReviewerSpecialtyComparer*, implements *IEqualityComparer<string>* for queries that only want to compare strings that are keys:

```
public class ReviewerSpecialtyComparer :
    IEqualityComparer<string>
```

```

{
    #region IEqualityComparer<string> Members

    public bool Equals(string x, string y)
    {
        return x == y;
    }

    public int GetHashCode(string obj)
    {
        return obj.GetHashCode();
    }

    #endregion
}

```

Another class is *ReviewerKey*, which is used in *GroupBy* operations to show how to group a composite key that is based on a custom type:

```

public class ReviewerKey :
    IEqualityComparer<ReviewerKey>
{
    public string Specialty { get; set; }
    public int Year { get; set; }

    public bool Equals(ReviewerKey x, ReviewerKey y)
    {
        return
            x.Specialty == y.Specialty &&
            x.Year == y.Year;
    }

    public int GetHashCode(ReviewerKey obj)
    {
        return
            (obj.Specialty + obj.Year)
            .ToString()
            .GetHashCode();
    }
}

```

The *ReviewerKey* class implements *IEqualityComparer<ReviewerKey>*, which will demonstrate how to work with custom composite keys in queries that you need to run with custom equality implementations.

Based on the preceding classes and the .NET *List<T>* collection class, the following objects contain the data for the LINQ to Objects examples in the Appendix query:

```

var joe = new Reviewer {
    Name = "Joe",
    Specialty = "Computers",
    Reviews = new List<ProductReview>() };
var tony = new Reviewer {
    Name = "Tony",
    Specialty = "Food",
    Reviews = new List<ProductReview>() };
var may = new Reviewer {
    Name = "May",
    Specialty = "Food",

```

```

    Reviews = new List<ProductReview>() };
var joleen = new Reviewer {
    Name = "Joleen",
    Specialty = "Media",
    Reviews = new List<ProductReview>() };
var jennifer = new Reviewer {
    Name = "Jennifer",
    Specialty = "Literature",
    Reviews = new List<ProductReview>() };

var productReviewers =
    new List<Reviewer>
    {
        joe, tony, may, jennifer
    };

var productPurchasers =
    new List<Reviewer>
    {
        tony, may, joleen
    };

var productReviews =
    new List<ProductReview>
    {
        new ProductReview {
            Reviewer = joe,
            Details = "Great" },
        new ProductReview {
            Reviewer = tony,
            Details = "Wunderbar" },
        new ProductReview {
            Reviewer = tony,
            Details = "Xho Merng Bla" },
        new ProductReview {
            Reviewer = may,
            Details = "No Comprende" }
    };

joe.Reviews.Add(productReviews[0]);
tony.Reviews.Add(productReviews[1]);
tony.Reviews.Add(productReviews[2]);
may.Reviews.Add(productReviews[3]);

```

You'll notice that the objects and collections in the preceding example contain relationships supporting advanced query scenarios. That is, *productReviews* contain references to *Reviewer* instances, establishing a hierarchical relationship.

Unlike with LINQ to Objects, data for LINQ to SQL comes from a SQL Server database.

Setting Up LINQ to SQL

To run the LINQ to SQL examples, you'll need to set up a DataContext as described in [Chapter 3](#). You should not try to use the DataContext from the code that accompanies the source code for this book without changing the connection string to refer to your database. The database I used was AdventureWorksLT, as described in [Chapter 3](#). You'll need to add all of the tables from AdventureWorksLT to your DataContext.

In addition to the tables from AdventureWorksLT, I’ve added a new entity, *Status5Orders*, to the DataContext that you’ll need to make. *Status5Orders* is an entity that derives from the *StandardOrderHeader* entity, and the *Cast* and *OfType* Standard Query Operator examples use *Status5Orders*. Here are instructions for adding *Status5Orders* to your DataContext, which is consistent with the entity inheritance topic in [Chapter 3](#):

1. Open your *.dbml file so you can see the design surface with all of your entities.
2. Open the Toolbox. You can press CTRL-W, X if necessary.
3. Drag and drop a *Class* control from the Toolbox to the *.dbml design surface. It’s best to position the new class near the *SalesOrderHeader* class to make creating the inheritance relationship easier.
4. Rename the new class as **Status5Order**. You can double-click the class name on the new object and retype the name.
5. Add the inheritance relationship, where *Status5Order* derives from *SalesOrderHeader*. You can click on the *Inheritance* control in the Toolbox, click on *Status5Order* on the *.dbml design surface, and then click on *SalesOrderHeader* on the *.dbml design surface. You should see an inheritance relationship on the *.dbml design surface.
6. Select the inheritance relationship on the *.dbml design surface, and open the Properties window. You can press CTRL-W, P if necessary.
7. Set the *Base Class Discriminator* value to 0, *Derived Class Discriminator* to 5, *Discriminator Property* to Status, and set *Inheritance Default* to SalesOrderHeader.

The *Status5Order* class should now be set up to derive from *SalesOrderHeader*. This is all you need to do to get the examples to run. Of course, in a real application, you would probably have multiple derived entities with different discriminators and multiple properties. However, the focus here is on seeing how the Standard Query Operators can be used, so the data source infrastructure only needs to be minimal.

Now that you have data sources set up, you can use any of the Standard Query Operator examples demonstrated in the rest of this appendix.

Organization of the Examples

The examples in this appendix are organized alphabetically. This is so you can use it as a reference. If you know the operator you’re interested in, you can find it rather quickly. On the other hand, Microsoft has organized the Standard Query Operators logically, into functional groups. I present these groupings here, which you can use if you aren’t quite sure of which operator you need but know what you want to do. You can then find an operator that meets your needs, and quickly find examples for that operator in the alphabetical reference. The following tables categorize and describe the Standard Query Operators.

Aggregate Operators

Aggregate	Enables creating custom aggregates
Average	Average of sequence values
Count	Number of items—return type int
LongCount	Larger counts—return type long
Max	Largest number in sequence

Min	Smallest number in sequence
Sum	Sequence numbers added together

Concatenation Operator

Concat	Concatenates two sequences
--------	----------------------------

Conversion Operators

AsEnumerable	Converts an IQueryable<T> sequence to an IEnumerable<T> sequence
AsQueryable	Converts an IEnumerable<T> sequence to an IQueryable<T> sequence
Cast	Converts non-generic sequence to a type
OfType	Extracts type of objects from a sequence
ToArray	Converts sequence to an Array
ToDictionary	Converts sequence to a Dictionary
ToList	Converts sequence to a List
ToLookup	Converts sequence to Lookup collection

Element Operators

ElementAt	Returns item at position
ElementAtOrDefault	Returns item at position or default for type
First	Returns first item in sequence
FirstOrDefault	Returns first item in sequence or default of type
Last	Returns last item in sequence
LastOrDefault	Returns last item in sequence or default of type
Single	Returns one item
SingleOrDefault	Returns one item or default of type

Equality Operator

SequenceEqual	True if sequences are equal, or false
---------------	---------------------------------------

Filter Operators

OfType	Returns items of type
Where	Applies filter to sequence

Generation Operators

DefaultIfEmpty	Provides default values in left joins
Empty	Creates an empty sequence
Range	Creates a sequence with values in range
Repeat	Creates a sequence with a repeated value

Grouping Operators

GroupBy	Groups data
ToLookup	Creates a Lookup collection

Join Operators

Join	Joins two sequences on equal keys
GroupJoin	Joins and groups sequences on equal keys

Partitioning Operators

Skip	Ignores items in a sequence number of times
SkipWhile	Ignores items in a sequence while a specified condition is true
Take	Includes items in a sequence number of times
TakeWhile	Includes items in a sequence while specified condition is true

Projection Operators

Select	Defines custom projection for each item in a sequence
SelectMany	Flattens hierarchy of objects

Quantifier Operators

All	All items satisfy condition
Any	At least one item satisfies condition
Contains	Element is in sequence

Set Operators

Distinct	Ensures each item in a sequence is unique
Except	Returns all items but a specified group
Intersect	Returns common items of two sequences
Union	Returns all items from two sequences

Sorting Operators

OrderBy	Sorts by specified property
OrderByDescending	Sorts by specified property in reverse
ThenBy	Subsequent property to sort by
ThenByDescending	Subsequent property to sort in reverse
Reverse	Rewrites a sequence in reverse order

Alphabetical Standard Query Operator Reference

The following list of items is the set of valid Standard Query Operators, listed in alphabetical order. You'll see the signature of the extension method for each operator, parameter descriptions, supporting LINQ providers, and an example of usage.

Aggregate

Description Allows you to create custom aggregates. To save time, you should check for existing aggregates such as *Average*, *Count*, and *Sum* before writing your own.

Signature 1

```
public static TSource Aggregate<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, TSource, TSource> func)
```

Parameters

source	Sequence to operate on
func	Custom operation

Example

```
var ints = new List<int> { 1, 2, 3 };
var sum = ints.Aggregate(
    (acc, curr) => acc + curr);
```

Provider LINQ to Objects

Signature 2

```
public static TAccumulate Aggregate<TSource, TAccumulate>(
    this IEnumerable<TSource> source,
    TAccumulate seed,
    Func<TAccumulate, TSource, TAccumulate> func)
```

Parameters

source	Sequence to operate on
seed	Value to begin at
func	Custom operation

Example

```
var ints = new List<int> { 1, 2, 3 };
var seedSum = ints.Aggregate(4,
    (acc, curr) => acc + curr);
```

Provider LINQ to Objects

Signature 3

```
public static TResult Aggregate<TSource, TAccumulate, TResult>(
    this IEnumerable<TSource> source,
    TAccumulate seed,
    Func<TAccumulate, TSource, TAccumulate> func,
```

Func<TAccumulate, TResult> resultSelector)

Parameters

source	Sequence to operate on
seed	Value to begin at
func	Custom operation
resultSelector	Projections on results

Example

```
var ints = new List<int> { 1, 2, 3 };  
var modSum = ints.Aggregate(4,  
    (acc, curr) => acc + curr,  
    acc => acc % 3);
```

Provider LINQ to Objects

All

Description Confirms that all items in a sequence match the specified condition.

Signature

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

Parameters

source	Sequence to operate on
predicate	Condition to check

Example

```
var hasAddress =  
    ctx.Customers.All(  
        cust => cust.EmailAddress != null  
    );
```

Providers LINQ to Objects, LINQ to SQL

Any

Description Confirms that at least one item in the sequence matches the specified condition.

Signature 1

```
public static bool Any<TSource>(  
    this IQueryable<TSource> source)
```

Parameter

source	Sequence to operate on
--------	------------------------

Example

```
var hasCusts = ctx.Customers.Any();
```

Providers LINQ to Objects, LINQ to SQL

Signature 2

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

Parameters

source	Sequence to operate on
predicate	Condition to check

Example

```
var noEmail = ctx.Customers.Any(  
    cust => cust.EmailAddress == null);
```

Providers LINQ to Objects, LINQ to SQL

AsEnumerable

Description Converts *IQueryable<T>* sequence to *IEnumerable<T>*.

Signature

```
public static IEnumerable<TSource> AsEnumerable<TSource>(  
    this IEnumerable<TSource> source)
```

Parameter

source	Sequence to operate on
--------	------------------------

Example

```
var enumCusts = ctx.Customers.AsEnumerable();
```

Providers LINQ to Objects, LINQ to SQL

AsQueryable

Description Converts *IEnumerable<T>* sequence to *IQueryable<T>*.

Signature

```
public static IQueryable AsQueryable(  
    this IEnumerable source)
```

Parameter

source	Sequence to operate on
--------	------------------------

Example

```
var queryCusts = enumCusts.AsQueryable();
```

Providers LINQ to Objects, LINQ to SQL

Average

Description Computes the average of a sequence. Overloaded in both signatures for nonnullable and nullable *decimal*, *double* *int*, *float*, *long*.

Signature 1

```
public static Nullable<double> Average(  
    this IQueryable<Nullable<int>> source)
```

Parameter

source	Sequence to operate on
--------	------------------------

Example

```
var avg =  
    (from order in ctx.SalesOrderHeaders  
     select order.TotalDue)  
     .Average();
```

Providers LINQ to Objects, LINQ to SQL

Signature 2

```
public static double Average<TSource>(  
    this IQueryable<TSource> source,  
    Expression<Func<TSource, int>> selector)
```

Parameters

source	Sequence to operate on
selector	Specifies which property to calculate

Example

```
var avgSelect =
    ctx.SalesOrderHeaders.Average(
        order => order.TotalDue
    );
```

Providers LINQ to Objects, LINQ to SQL

Cast

Description Converts collection to specified type. Useful for converting non-generic collections to generic collections that can be queried via LINQ.

Signature

```
public static IQueryable<TResult> Cast<TResult>(
    this IQueryable source)
```

Parameter

source	Sequence to operate on
--------	------------------------

Example

```
var stat5 = ctx.SalesOrderHeaders.Cast<Status5Order>();
```

Providers LINQ to Objects, LINQ to SQL

Concat

Description Concatenates one sequence to another.

Signature

```
public static IQueryable<TSource> Concat<TSource>(
    this IQueryable<TSource> source1,
    IEnumerable<TSource> source2)
```

Parameters

source1	First sequence
source2	Concatenated to end of first sequence

Example

```
var ordCat =
    ctx.SalesOrderHeaders.Concat(
        ctx.SalesOrderHeaders
    );
```

Providers LINQ to Objects, LINQ to SQL

Contains

Description Tells whether an item is part of another sequence.

Signature 1

```
public static bool Contains<TSource>(
    this IQueryable<TSource> source,
    TSource item)
```

Parameters

source	Sequence to operate on
item	Item to check

Example

```
var custToSearch =
    ctx.Customers
        .Where(cust => cust.CustomerID == 1)
        .Single();

var hasCust1 = ctx.Customers.Contains(custToSearch);
```

Providers LINQ to Objects, LINQ to SQL

Signature 2

```
public static bool Contains<TSource>(
    this IEnumerable<TSource> source,
    TSource value,
    IEqualityComparer<TSource> comparer)
```

Parameters

source	Sequence to operate on
value	Item to check
comparer	Custom compare object

Example

```
var reviewer = productReviews[1];

var foundReviewer =
    productReviews.Contains(reviewer, new Reviewer());
```

or

```
var foundReviewer =
    productReviews.Contains(reviewer, reviewer);
```

Provider LINQ to Objects

Count

Description Number of items in a sequence.

Signature 1

```
public static int Count<TSource>(  
    this IQueryable<TSource> source)
```

Parameter

source	Sequence to operate on
--------	------------------------

Example

```
var custCount = ctx.Customers.Count();
```

Providers LINQ to Objects, LINQ to SQL

Signature 2

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

Parameters

source	Sequence to operate on
predicate	Filters items to include in count

Example

```
var salesCount =  
    ctx.SalesOrderHeaders.Count(  
        order => order.TotalDue > 300m  
    );
```

Providers LINQ to Objects, LINQ to SQL

DefaultIfEmpty

Description Supports left *join* by providing default data for the type of each column when record on left side of *join* doesn't have matching record on right side of *join*.

Signature 1

```
public static IQueryable<TSource> DefaultIfEmpty<TSource>(  
    this IQueryable<TSource> source)
```

Parameter

source	Sequence to operate on
--------	------------------------

Example

```
var custOrders =
    from cust in ctx.Customers
    join ord in ctx.SalesOrderHeaders
    on cust.CustomerID equals ord.CustomerID
    into custOrdContinuation
    from custOrd in custOrdContinuation.DefaultIfEmpty()
    select
        new
        {
            cust.LastName,
            custOrd.SalesOrderNumber
        };
```

Providers LINQ to Objects, LINQ to SQL

Signature 2

```
public static IEnumerable<TSource> DefaultIfEmpty<TSource>(
    this IEnumerable<TSource> source,
    TSource defaultValue)
```

Parameters

source	Sequence to operate on
defaultValue	Value to use as default

Example

```
var defaultReview =
    new ProductReview
    {
        Details = "Review not available"
    };
var reviewReviewersJoined =
    from pRevr in productReviews
    join pRev in productReviews
    on pRevr equals pRev.Reviewer
    into pRevRevrContinuation
    from pRevRevr in
        pRevRevrContinuation
        .DefaultIfEmpty(defaultReview)
    select
        new
        {
            Name = pRevr.Name,
            Specialty = pRevr.Specialty,
            Details = pRevRevr.Details
        };
```

Provider LINQ to Objects

Distinct

Description Ensures sequence results in unique items.

Signature 1

```
public static IQueryable<TSource> Distinct<TSource>(  
    this IQueryable<TSource> source)
```

Parameter

source	Sequence to operate on
--------	------------------------

Example

```
var orderDetailProducts =  
    from ordDetail in ctx.SalesOrderDetails  
    join product in ctx.Products  
    on ordDetail.ProductID equals product.ProductID  
    into prodSoldContinuation  
    from prodSold in prodSoldContinuation.DefaultIfEmpty()  
    select prodSold.Name;  
  
var ordDetails = orderDetailProducts.ToList();  
var productsSold = orderDetailProducts.Distinct();
```

Providers LINQ to Objects, LINQ to SQL

Signature 2

```
public static IQueryable<TSource> Distinct<TSource>(  
    this IQueryable<TSource> source)
```

Parameter

source	Sequence to operate on
--------	------------------------

Example

```
var dupNames =  
    new List<Reviewer>  
    {  
        joe, may, joe, tony, may  
    };  
  
var uniqueNames =  
    dupNames.Distinct(new Reviewer());
```

Provider LINQ to Objects

ElementAt

Description Selects item at specified position in sequence.

Signature

```
public static TSource ElementAt<TSource>(
    this IEnumerable<TSource> source,
    int index)
```

Parameters

source	Sequence to operate on
index	Position in sequence to find item

Example

```
var reviewerElementAt1 = productReviewers.ElementAt;
```

Provider LINQ to Objects

ElementAtOrDefault

Description Selects item at specified position in sequence or returns a default value for the type if the index doesn’t fall into the range of the sequence.

Signature

```
public static TSource ElementAtOrDefault<TSource>(
    this IEnumerable<TSource> source,
    int index)
```

Parameters

source	Sequence to operate on
index	Position in sequence to find item

Example

```
var reviewerElementAt2 =
    productReviewers.ElementAtOrDefault;
```

Provider LINQ to Objects

Empty

Description Returns an *IEnumerable<T>* with no items.

Signature

```
public static IEnumerable<TResult> Empty<TResult>()
```

Parameter None

Example

```
var emptyReviewers = Enumerable.Empty<Reviewer>();
```

Provider LINQ to Objects

Except

Description Specifies items to exclude from sequence.

Signature 1

```
public static IEnumerable<TSource> Except<TSource>(
    this IEnumerable<TSource> first,
    IEnumerable<TSource> second)
```

Parameters

first	Sequence to operate on
second	Sequence to leave out

Example

```
var ordersToExclude =
    from order in ctx.SalesOrderHeaders
    where order.TotalDue < 100.00m
    select order;
```

```
var ordersToInclude =
    ctx.SalesOrderHeaders.Except(
        ordersToExclude.AsEnumerable());
```

Providers LINQ to Objects, LINQ to SQL

Signature 2

```
public static IEnumerable<TSource> Except<TSource>(
    this IEnumerable<TSource> first,
    IEnumerable<TSource> second,
    IEqualityComparer<TSource> comparer)
```

Parameters

first	Sequence to operate on
second	Sequence to leave out
comparer	Custom comparer object

Example

```
var reviewersToExclude =
    new List<Reviewer> { productReviewers[0] };
```

```
var reviewersToInclude =
    productReviewers.Except(
        reviewersToExclude,
        new Reviewer());
```

Provider LINQ to Objects

First

Description Selects first item in sequence.

Signature 1

```
public static TSource First<TSource>(
    this IQueryable<TSource> source)
```

Parameter

source	Sequence to operate on
--------	------------------------

Example

```
var firstAddress = ctx.Addresses.First();
```

Providers LINQ to Objects, LINQ to SQL

Signature 2

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

Parameters

source	Sequence to operate on
predicate	Filters sequence

Example

```
var firstCity =
    ctx.Addresses.First(
        addr => addr.City == "Denver");
```

Providers LINQ to Objects, LINQ to SQL

FirstOrDefault

Description Selects first item in sequence or returns a default value for the type when sequence contains no values.

Signature 1

```
public static TSource FirstOrDefault<TSource>(
```

this IQueryable<TSource> source)

Parameter

source	Sequence to operate on
--------	------------------------

Example

```
var noAddress = emptyAddresses.FirstOrDefault();
```

Providers LINQ to Objects, LINQ to SQL

Signature 2

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

Parameters

source	Sequence to operate on
predicate	Filters sequence

Example

```
var noCountry =
    ctx.Addresses.FirstOrDefault(
        addr => addr.CountryRegion == "UK");
```

Providers LINQ to Objects, LINQ to SQL

GroupBy

Description Provides grouping of data from a sequence. See Chapters 2 and 3 for examples of handling a *group by* clause, which returns an *IGrouping<TKey, TSource>*.

Signature 1

```
public static IQueryable<IGrouping<TKey, TSource>> GroupBy<TSource, TKey>(
    this IQueryable<TSource> source,
    Expression<Func<TSource, TKey>> keySelector)
```

Parameters

source	Sequence to operate on
keySelector	Key to group by

Example

```
var salesDetails =
```

```
ctx.SalesOrderDetails.GroupBy(  
    key => key.Product.Name);
```

Provider LINQ to Objects, LINQ to SQL

Signature 2

```
public static IEnumerable<IGrouping<TKey, TSource>> GroupBy<TSource, TKey>(  
    this IEnumerable<TSource> source,  
    Func<TSource, TKey> keySelector,  
    IEqualityComparer<TKey> comparer)
```

Parameters

source	Sequence to operate on
keySelector	Key to group by
comparer	Custom comparer object

Example

```
var groupedReviewerKeys =  
    productReviewers  
    .GroupBy(  
        gpReviewer =>  
            new ReviewerKey  
            {  
                Specialty = gpReviewer.Specialty,  
                Year = 2009  
            },  
        new ReviewerKey());
```

Provider LINQ to Objects

Signature 3

```
public static IQueryable<IGrouping<TKey, TElement>> GroupBy<TSource, TKey,  
TElement>(  
    this IQueryable<TSource> source,  
    Expression<Func<TSource, TKey>> keySelector,  
    Expression<Func<TSource, TElement>> elementSelector)
```

Parameters

source	Sequence to operate on
keySelector	Key to group by
elementSelector	Custom projection over results

Example

```
var salesDetailsSummary =
    ctx.SalesOrderDetails
        .GroupBy<SalesOrderDetail, string, DetailSummary> (
            key => key.Product.Name,
            val =>
                new DetailSummary
                {
                    LineTotal = val.LineTotal,
                    Quantity = val.OrderQty
                });
```

Providers LINQ to Objects, LINQ to SQL

Signature 4

```
public static IQueryable<TResult> GroupBy<TSource, TKey, TResult>(
    this IQueryable<TSource> source,
    Expression<Func<TSource, TKey>>> keySelector,
    Expression<Func<TKey, IEnumerable<TSource>, TResult>>> resultSelector)
```

Parameters

source	Sequence to operate on
keySelector	Key to group by
resultSelector	Custom summarization of each group

Example

```
var salesDetailKeyAndGroupGrouping =
    ctx.SalesOrderDetails
        .GroupBy(
            key => key.Product.Name,
            (key, group) =>
                new DetailSummary
                {
                    GroupName = key,
                    LineTotal =
                        group.Average(
                            detail => detail.LineTotal),
                    Quantity =
                        (int)group.Average(
                            detail => detail.OrderQty)
                });
```

Providers LINQ to Objects, LINQ to SQL

Signature 5

```
public static IEnumerable<IGrouping<TKey, TElement>> GroupBy<TSource, TKey,
TElement>(
    this IEnumerable<TSource> source,
    Func<TSource, TKey> keySelector,
    Func<TSource, TElement> elementSelector,
    IEqualityComparer<TKey> comparer)
```

Parameters

source	Sequence to operate on
keySelector	Key to group by
elementSelector	Custom projection over results
comparer	Custom comparer object

Example

```
var groupedReviewers =
    productReviewers
    .GroupBy(
        gpReviewer =>
            new ReviewerKey
            {
                Specialty = gpReviewer.Specialty,
                Year = 2009
            },
        gpReviewer =>
            new Reviewer
            {
                Specialty = gpReviewer.Specialty,
                Name = gpReviewer.Name
            },
        new ReviewerKey());
```

Provider LINQ to Objects

Signature 6

```
public static IQueryable<TResult> GroupBy<TSource, TKey, TElement, TResult>(
    this IQueryable<TSource> source,
    Expression<Func<TSource, TKey>> keySelector,
    Expression<Func<TSource, TElement>> elementSelector,
    Expression<Func<TKey, IEnumerable<TElement>, TResult>> resultSelector)
```

Parameters

source	Sequence to operate on
keySelector	Key to group by
elementSelector	Custom projection over group
resultSelector	Custom summary of each group

Example

```
var salesDetailGroupSummary =
```



```
ctx.SalesOrderDetails
.GroupBy(
    detail =>
        detail.LineTotal <= 100m ?
            “Low”:
                detail.LineTotal > 100m &&
                detail.LineTotal < 500m ?
                    “Medium”:
                        “High”,
    detail =>
        new
        {
            LineTotal = detail.LineTotal,
            Quantity = detail.OrderQty
        },
    (key, group) =>
        new DetailSummary
        {
            GroupName = key,
            LineTotal =
                group.Average(
                    summary => summary.LineTotal),
            Quantity =
                (int)group.Average(
                    summary => summary.Quantity)
        }
);
```

Providers LINQ to Objects, LINQ to SQL

Signature 7

```
public static IEnumerable<TResult> GroupBy<TSource, TKey, TResult>(
    this IEnumerable<TSource> source,
    Func<TSource, TKey> keySelector,
    Func<TKey, IEnumerable<TSource>, TResult> resultSelector,
    IEqualityComparer<TKey> comparer)
```

Parameters

source	Sequence to operate on
keySelector	Key to group by
resultSelector	Custom summarization of group
comparer	Custom comparer object

Example

```
var groupedReviewerKeySummary =
    productReviewers
    .GroupBy(
        gpReviewer =>
            new ReviewerKey
            {
```

```

        Specialty = gpReviewer.Specialty,
        Year = 2009
    },
    (key, group) =>
        new
        {
            GroupName = key.Specialty + “ ” + key.Year,
            NameCount = group.Count().ToString()
        },
    new ReviewerKey());

```

Provider LINQ to Objects

Signature 8

```

public static IEnumerable<TResult> GroupBy<TSource, TKey, TElement,
TResult>(
    this IEnumerable<TSource> source,
    Func<TSource, TKey> keySelector,
    Func<TSource, TElement> elementSelector,
    Func<TKey, IEnumerable<TElement>, TResult> resultSelector,
    IEqualityComparer<TKey> comparer)

```

Parameters

source	Sequence to operate on
keySelector	Key to group by
elementSelector	Custom projection over results
resultSelector	Custom summarization of group
comparer	Custom comparer object

Example

```

var groupedReviewerKeyCustomizedSummary =
    productReviewers
    .GroupBy(
        gpReviewer =>
            new ReviewerKey
            {
                Specialty =
                    gpReviewer.Specialty == “Food” ?
                    “Cuisine”:
                    “Technology”,
                Year = 2009
            },
        gpReviewer => gpReviewer.Specialty,
        (key, group) =>
            new
            {
                GroupName = key.Specialty + ” ” + key.Year,
                NameCount = group.Count().ToString()
            }
    );

```

```
    },  
    new ReviewerKey());
```

Provider LINQ to Objects

GroupJoin

Description Joins two sequences and allows you to group them.

Signature 1

```
public static IQueryable<TResult> GroupJoin<TOuter, TInner, TKey, TResult>(
    this IQueryable<TOuter> outer,
    IEnumerable<TInner> inner,
    Expression<Func<TOuter, TKey>> outerKeySelector,
    Expression<Func<TInner, TKey>> innerKeySelector,
    Expression<Func<TOuter, IEnumerable<TInner>, TResult>> resultSelector)
```

Parameters

outer	First group to join
inner	Second group to join with first
outerKeySelector	Key for first group
innerKeySelector	Key for second group to match first group key
resultSelector	Defines grouping

Example

```
var reviewsForSpecialty =  
    productReviews  
    .GroupJoin(  
        productReviews,  
        reviewerKey => reviewerKey,  
        reviewKey => reviewKey.Reviewer,  
        (reviewerKey, allReviews) =>  
            new  
            {  
                Name = reviewerKey.Name,  
                Reviews = allReviews  
            },  
        new Reviewer()  
    );
```

Providers LINQ to Objects, LINQ to SQL

Signature 2

```
public static IEnumerable<TResult> GroupJoin<TOuter, TInner, TKey, TResult>(
    this IEnumerable<TOuter> outer,
    IEnumerable<TInner> inner,
```

Func<TOuter, TKey> outerKeySelector,
Func<TInner, TKey> innerKeySelector,
Func<TOuter, IEnumerable<TInner>, TResult> resultSelector,
IEqualityComparer<TKey> comparer)

Parameters

outer	First group to join
inner	Second group to join with first
outerKeySelector	Key for first group
innerKeySelector	Key for second group to match first group key
resultSelector	Defines grouping
comparer	Custom comparer object

Example

```
var reviewsForSpecialty =
    productReviewers
    .GroupJoin(
        productReviews,
        reviewerKey => reviewerKey,
        reviewKey => reviewKey.Reviewer,
        (reviewerKey, allReviews) =>
            new
            {
                Name = reviewerKey.Name,
                Reviews = allReviews
            },
        new Reviewer()
    );
```

Provider LINQ to Objects

Intersect

Description Results in a sequence of common items from two other sequences.

Signature 1

```
public static IQueryable<TSource> Intersect<TSource>(
    this IQueryable<TSource> source1,
    IEnumerable<TSource> source2)
```

Parameters

source1	First sequence

source2	Second sequence
---------	-----------------

Example

```
var bikePurchasers =
    from prod in ctx.Products
    where prod.ProductNumber.StartsWith("BK")
    from details in prod.SalesOrderDetails
    select details.SalesOrderHeader.Customer;

var helmetPurchasers =
    from prod in ctx.Products
    where prod.ProductNumber.StartsWith("HL")
    from details in prod.SalesOrderDetails
    select details.SalesOrderHeader.Customer;

var bikeAndHelmetPurchasers =
    bikePurchasers
    .Intersect(
        helmetPurchasers);
```

Providers LINQ to Objects, LINQ to SQL

Signature 2

```
public static IEnumerable<TSource> Intersect<TSource>(
    this IEnumerable<TSource> first,
    IEnumerable<TSource> second,
    IEqualityComparer<TSource> comparer)
```

Parameters

source1	First sequence
source2	Second sequence
Comparer	Custom comparer object

Example

```
var reviewersAndPurchasers =
    productReviewers
    .Intersect(
        productPurchasers,
        new Reviewer()
    );
```

Provider LINQ to Objects

Join

Description Joins two sequences together.

Signature 1

```
public static IQueryable<TResult> Join<TOuter, TInner, TKey, TResult>(  
    this IQueryable<TOuter> outer,  
    IEnumerable<TInner> inner,  
    Expression<Func<TOuter, TKey>> outerKeySelector,  
    Expression<Func<TInner, TKey>> innerKeySelector,  
    Expression<Func<TOuter, TInner, TResult>> resultSelector)
```

Parameters

outer	First group to join
inner	Second group to join with first
outerKeySelector	Key for first group
innerKeySelector	Key for second group to match first group key
resultSelector	Custom projection over results

Example

```
var ordersAndDetails =  
    ctx.SalesOrderHeaders  
    .Join(  
        ctx.SalesOrderDetails,  
        order => order,  
        detail => detail.SalesOrderHeader,  
        (order, details) =>  
            new  
            {  
                OrderNumber = order.SalesOrderNumber,  
                ProductName = details.Product.Name,  
                Amount = details.LineTotal  
            }  
    );
```

Providers LINQ to Objects, LINQ to SQL

Signature 2

```
public static IEnumerable<TResult> Join<TOuter, TInner, TKey, TResult>(  
    this IEnumerable<TOuter> outer,  
    IEnumerable<TInner> inner,  
    Func<TOuter, TKey> outerKeySelector,  
    Func<TInner, TKey> innerKeySelector,  
    Func<TOuter, TInner, TResult> resultSelector,  
    IEqualityComparer<TKey> comparer)
```

Parameters

outer	First group to join
inner	Second group to join with first

outerKeySelector	Key for first group
innerKeySelector	Key for second group to match first group key
resultSelector	Custom projection over results
comparer	Custom comparer object

Example

```

var reviewersPurchasersJoined =
    productReviews
        .Join(
            productReviews,
            prodReviewer => prodReviewer,
            reviewItem => reviewItem.Reviewer,
            (prodReviewer, reviewItem) =>
                new
                {
                    Reviewer = prodReviewer.Name,
                    Details = reviewItem.Details
                },
            new Reviewer()
        );

```

Provider LINQ to Objects

Last

Description Selects the last item in a sequence.

Signature 1

```

public static TSource Last<TSource>(
    this IEnumerable<TSource> source)

```

Parameter

source	Sequence to operate on
--------	------------------------

Example

```

var lastReviewer = productReviews.Last();

```

Provider LINQ to Objects

Signature 2

```

public static TSource Last<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, bool> predicate)

```

Parameters

source	Sequence to operate on
predicate	Filters sequence results

Example

```
var lastFoodReviewer =  
    productReviewers  
    .Last(  
        prodRev => prodRev.Specialty == "Food"  
    );
```

Provider LINQ to Objects

LastOrDefault

Description Selects the last item in a sequence or returns a default value for the type when sequence is empty.

Signature 1

```
public static TSource LastOrDefault<TSource>(  
    this IEnumerable<TSource> source)
```

Parameter

source	Sequence to operate on
--------	------------------------

Example

```
var lastReviewerDefault = productReviewers.LastOrDefault();
```

Provider LINQ to Objects

Signature 2

```
public static TSource LastOrDefault<TSource>(  
    this IEnumerable<TSource> source,  
    Func<TSource, bool> predicate)
```

Parameters

source	Sequence to operate on
predicate	Filters sequence results

Example

```
var lastFoodReviewerDefault =  
    productReviewers  
    .LastOrDefault(  
        prodRev => prodRev.Specialty == "Psychology"
```


);

Provider LINQ to Objects

LongCount

Description Number of items in a sequence. Result is 64-bit *long* value.

Signature 1

```
public static long LongCount<TSource>(  
    this IQueryable<TSource> source)
```

Parameter

source	Sequence to operate on
--------	------------------------

Example

```
var orderCount =  
    ctx.SalesOrderHeaders.LongCount();
```

Providers LINQ to Objects, LINQ to SQL

Signature 2

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

Parameters

source	Sequence to operate on
predicate	Filters items to include in count

Example

```
var orderOver5000Count =  
    ctx.SalesOrderHeaders.LongCount(  
        ord => ord.TotalDue > 5000m  
    );
```

Providers LINQ to Objects, LINQ to SQL

Max

Description Returns largest number in sequence.

Signature 1

```
public static TSource Max<TSource>(  
    this IQueryable<TSource> source)
```

Parameter

source	Sequence to operate on
--------	------------------------

Example

```
var maxOrder =  
    (from ord in ctx.SalesOrderHeaders  
     select ord.TotalDue)  
    .Max();
```

Providers LINQ to Objects, LINQ to SQL

Signature 2

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

Parameters

source	Sequence to operate on
selector	Selects property to use

Example

```
var maxOrderFiltered =  
    ctx.SalesOrderHeaders.Max(  
        ord => ord.TotalDue  
    );
```

Providers LINQ to Objects, LINQ to SQL

Min

Description Returns smallest number in sequence.

Signature 1

```
public static TSource Min<TSource>(  
    this IQueryable<TSource> source)
```

Parameter

source	Sequence to operate on
--------	------------------------

Example

```
var minOrder =  
    (from ord in ctx.SalesOrderHeaders  
     select ord.TotalDue)
```

```
.Min();
```

Providers LINQ to Objects, LINQ to SQL

Signature 2

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

Parameters

source	Sequence to operate on
selector	Selects property to use

Example

```
var minOrderFiltered =
    ctx.SalesOrderHeaders.Min(
        ord => ord.TotalDue
    );
```

Providers LINQ to Objects, LINQ to SQL

OfType

Description Selects items from sequence of specified type.

Signature

```
public static IQueryable<TResult> OfType<TResult>(
    this IQueryable source)
```

Parameter

source	Sequence to operate on
--------	------------------------

Example

```
var status5OrderTypes =
    ctx.SalesOrderHeaders.OfType<Status5Order>();
```

Providers LINQ to Objects, LINQ to SQL

OrderBy

Description Sorts sequence in ascending order.

Signature 1

```
public static IOrderedQueryable<TSource> OrderBy<TSource, TKey>(
    this IQueryable<TSource> source,
    Expression<Func<TSource, TKey>> keySelector)
```

Parameters

source	Sequence to operate on
keySelector	Key to sort on

Example

```
var countries =
    ctx.Addresses.OrderBy(
        addr => addr.CountryRegion
    );
```

Providers LINQ to Objects, LINQ to SQL

Signature 2

```
public static IEnumerable<TSource> OrderBy<TSource, TKey>(
    this IEnumerable<TSource> source,
    Func<TSource, TKey> keySelector,
    IComparer<TKey> comparer)
```

Parameters

source	Sequence to operate on
keySelector	Key to sort on
comparer	Custom comparer object

Example

```
var orderedReviewers =
    productReviewers.OrderBy(
        pRev => pRev.Specialty,
        new Reviewer()
    );
```

Provider LINQ to Objects

OrderByDescending

Description Sorts sequence in descending order.

Signature 1

```
public static IQueryable<TSource> OrderByDescending<TSource, TKey>(
    this IQueryable<TSource> source,
    Expression<Func<TSource, TKey>> keySelector)
```

Parameters

--	--

source	Sequence to operate on
keySelector	Key to sort on

Example

```
var countriesDescending =
    ctx.Addresses.OrderByDescending(
        addr => addr.CountryRegion
    );
```

Providers LINQ to Objects, LINQ to SQL

Signature 2

```
public static IEnumerable<TSource> OrderByDescending<TSource, TKey>(
    this IEnumerable<TSource> source,
    Func<TSource, TKey> keySelector,
    IComparer<TKey> comparer)
```

Parameters

source	Sequence to operate on
keySelector	Key to sort on
comparer	Custom comparer object

Example

```
var orderedReviewersDescending =
    productReviewers.OrderByDescending(
        pRev => pRev.Specialty,
        new Reviewer()
    );
```

Provider LINQ to Objects

Range

Description Creates a sequence that includes *int* values starting from a specified value for a specified count.

Signature

```
public static IEnumerable<int> Range(
    int start,
    int count)
```

Parameters

start	Number to start at
-------	--------------------

count	Number of items to add
-------	------------------------

Example

```
var rangeOfSeven = Enumerable.Range(1, 7);
```

Provider LINQ to Objects

Repeat

Description Duplicates an item a specified number of times.

Signature

```
public static IEnumerable<TResult> Repeat<TResult>(
    TResult element,
    int count)
```

Parameters

element	Item to duplicate
count	Number of times to duplicate

Example

```
var genericReviewers =
    Enumerable.Repeat(
        new Reviewer
        {
            Name = "Unassigned",
            Specialty = "To Be Determined"
        },
        5
    );
```

Provider LINQ to Objects

Reverse

Description Rewrites a sequence in reverse order.

Signature

```
public static IEnumerable<TSource> Reverse<TSource>(
    this IEnumerable<TSource> source)
```

Parameter

source	Sequence to operate on
--------	------------------------

Example

```
productReviewers.Reverse();
```

Provider LINQ to Objects

Select

Description Allows you to perform a custom projection over a sequence.

Signature 1

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

Parameters

source	Sequence to operate on
selector	Custom projection

Example

```
var selectedStateProvince =
    ctx.Addresses.Select(
        addr => addr.StateProvince
    );
```

Providers LINQ to Objects, LINQ to SQL

Signature 2

```
public static IEnumerable<TResult> Select<TSource, TResult>(
    this IEnumerable<TSource> source,
    Func<TSource, int, TResult> selector)
```

Parameters

source	Sequence to operate on
selector	Custom projection with a row ID

Example

```
var prodRevWithLineNumbers =
    productReviewers.Select(
        (pRev, row) =>
            new
            {
                Row = row + 1,
                Name = pRev.Name
            }
    );
```

Provider LINQ to Objects

SelectMany

Description Flattens a hierarchy.

Signature 1

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

Parameters

source	Parent collection
selector	Child collection

Example

```
var ordersAndDetailsSelectMany =
    ctx.SalesOrderHeaders.SelectMany(
        order => order.SalesOrderDetails
    );
```

Providers LINQ to Objects, LINQ to SQL

Signature 2

```
public static IEnumerable<TResult> SelectMany<TSource, TResult>(
    this IEnumerable<TSource> source,
    Func<TSource, int, IEnumerable<TResult>> selector)
```

Parameters

source	Parent collection
selector	Child collection with row ID

Example

```
var prodRevSelectMany =
    productReviewers
    .SelectMany(
        (pRev, row) =>
            from pr in pRev.Reviews
            select
                new
                {
                    Row = row,
                    Name = pRev.Name,
                    Review = pr.Details
                }
    );
```


Provider LINQ to Objects

Signature 3

```
public static IQueryable<TResult> SelectMany<TSource, TCollection,
TResult>(
    this IQueryable<TSource> source,
    Expression<Func<TSource, IEnumerable<TCollection>>> collectionSelector,
    Expression<Func<TSource, TCollection, TResult>> resultSelector)
```

Parameters

source	Parent collection
collectionSelector	Child collection
resultSelector	Custom projection over resulting sequence

Example

```
var ordDetSelectedSelectMany =
    ctx.SalesOrderHeaders.SelectMany(
        order => order.SalesOrderDetails,
        (order, detail) =>
            new
            {
                OrderNumber = order.SalesOrderNumber,
                Amount = detail.LineTotal
            }
    );
```

Providers LINQ to Objects, LINQ to SQL

Signature 4

```
public static IEnumerable<TResult> SelectMany<TSource, TCollection, TResult>(
    this IEnumerable<TSource> source,
    Func<TSource, int, IEnumerable<TCollection>> collectionSelector,
    Func<TSource, TCollection, TResult> resultSelector)
```

Parameters

source	Parent collection
collectionSelector	Child collection with row ID
resultSelector	Custom projection over sequence

Example

```
var prodRevSelectedSelectMany =
    productReviewers
        .SelectMany(
```

```
(pRev, row) =>
    from rev in pRev.Reviews
    select
        new
        {
            Row = row,
            Reviews = pRev.Reviews
        },
(pRev, revs) =>
    new
    {
        Row = revs.Row,
        Name = pRev.Name,
        Count = revs.Reviews.Count
    }
);
```

Provider LINQ to Objects

SequenceEqual

Description Determines if two sequences contain the same elements.

Signature 1

```
public static bool SequenceEqual<TSource>(
    this IQueryable<TSource> source1,
    IEnumerable<TSource> source2)
```

Parameters

source1	First sequence
source2	Second sequence

Example

```
var reviewersEqual =
    productReviewers
    .SequenceEqual(
        productReviewers
    );
```

Provider LINQ to Objects

Signature 2

```
public static bool SequenceEqual<TSource>(
    this IEnumerable<TSource> first,
    IEnumerable<TSource> second)
```

Parameters

first	First sequence

second	Second sequence
--------	-----------------

Example

```
var reviewersEqualCompared =
    productReviewers
        .SequenceEqual(
            productReviewers,
            new Reviewer()
        );
```

Provider LINQ to Objects

Single

Description Returns the object from a sequence with one item.

Signature 1

```
public static TSource Single<TSource>(
    this IQueryable<TSource> source)
```

Parameter

source	Sequence to operate on
--------	------------------------

Example

```
var getAddress =
    (from addr in ctx.Addresses
     where addr.AddressID == 9
     select addr)
        .Single();
```

Providers LINQ to Objects, LINQ to SQL

Signature 2

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

Parameters

source	Sequence to operate on
predicate	Filters sequence results

Example

```
var getAddressFiltered =
    ctx.Addresses.Single(
        addr => addr.AddressID == 11
```

);

Providers LINQ to Objects, LINQ to SQL

SingleOrDefault

Description Returns the object from a sequence with one item, or a default value for the type if sequence is empty or contains more than one item.

Signature 1

```
public static TSource SingleOrDefault<TSource>(
    this IQueryable<TSource> source)
```

Parameter

source	Sequence to operate on
--------	------------------------

Example

```
var getAddressDefault =
    (from addr in ctx.Addresses
     where addr.AddressID == 9999
     select addr)
    .SingleOrDefault();
```

Providers LINQ to Objects, LINQ to SQL

Signature 2

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

Parameters

source	Sequence to operate on
predicate	Filters sequence results

Example

```
var getAddressFilteredDefault =
    ctx.Addresses.SingleOrDefault(
        addr => addr.AddressID == 11
    );
```

Providers LINQ to Objects, LINQ to SQL

Skip

Description Ignores a specified number of items from sequence.

Signature

```
public static IQueryable<TSource> Skip<TSource>(  
    this IQueryable<TSource> source,  
    int count)
```

Parameters

source	Sequence to operate on
count	Number of items to ignore

Example

```
var skippedAddr = ctx.Addresses.Skip(10);
```

Providers LINQ to Objects, LINQ to SQL

SkipWhile

Description Ignores items from a sequence while a condition is true.

Signature 1

```
public static IEnumerable<TSource> SkipWhile<TSource>(  
    this IEnumerable<TSource> source,  
    Func<TSource, bool> predicate)
```

Parameters

source	Sequence to operate on
predicate	Ignore values while this evaluates to true

Example

```
var skippedWhile =  
    productReviewers.SkipWhile(  
        pRev => pRev.Specialty != “Computers”  
    );
```

Provider LINQ to Objects

Signature 2

```
public static IEnumerable<TSource> SkipWhile<TSource>(  
    this IEnumerable<TSource> source,  
    Func<TSource, int, bool> predicate)
```

Parameters

source	Sequence to operate on
--------	------------------------

predicate	Ignore values while this evaluates to true—contains index of current item in a sequence
-----------	---

Example

```
var skippedWhileRows =
    productReviewers.SkipWhile(
        (pRev, row) =>
            pRev.Specialty == "Computers" ||
            row < 1
    );
```

Provider LINQ to Objects

Take

Description Includes a specified number of items in a sequence.

Signature

```
public static IQueryable<TSource> Take<TSource>(
    this IQueryable<TSource> source,
    int count)
```

Parameters

source	Sequence to operate on
count	Number of items to include

Example

```
var addresses =
    ctx.Addresses.Take(10);
```

Providers LINQ to Objects, LINQ to SQL

Take While

Description Includes items in a sequence while a condition is true.

Signature 1

```
public static IEnumerable<TSource> TakeWhile<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, bool> predicate)
```

Parameters

source	Sequence to operate on
predicate	Includes values while this evaluates to true

Example

```
var reviewersWhile =  
    productReviewers.TakeWhile(  
        pRev => pRev.Specialty == "Food"  
    );
```

Provider LINQ to Objects

Signature 2

```
public static IEnumerable<TSource> TakeWhile<TSource>(  
    this IEnumerable<TSource> source,  
    Func<TSource, int, bool> predicate)
```

Parameters

source	Sequence to operate on
predicate	Include values while this evaluates to true—contains index of current item in a sequence

Example

```
var reviewersWhileRow =  
    productReviewers.TakeWhile(  
        (pRev, row) =>  
            pRev.Specialty == "Food" ||  
            row < 1  
    );
```

Provider LINQ to Objects

ThenBy

Description Sorts sequence in ascending order for subsequent key.

Signature 1

```
public static IOOrderedQueryable<TSource> ThenBy<TSource, TKey>(  
    this IOOrderedQueryable<TSource> source,  
    Expression<Func<TSource, TKey>> keySelector)
```

Parameters

source	Sequence to operate on
keySelector	Key to sort on

Example

```
var orderedAddressThenBy =  
    ctx.Addresses  
        .OrderBy(  
            addr => addr.CountryRegion
```

```
)
    .ThenBy(
        addr => addr.StateProvince
    );
```

Providers LINQ to Objects, LINQ to SQL

Signature 2

```
public static IEnumerable<TSource> ThenBy<TSource, TKey>(
    this IEnumerable<TSource> source,
    Func<TSource, TKey> keySelector,
    IComparer<TKey> comparer)
```

Parameters

source	Sequence to operate on
keySelector	Key to sort on
comparer	Custom comparer object

Example

```
var orderedReviewsThenBy =
    productReviewers
        .OrderBy(
            pRev => pRev.Specialty
        )
        .ThenBy(
            pRev => pRev.Name
        );
```

Provider LINQ to Objects

ThenByDescending

Description Sorts sequence in descending order by subsequent key.

Signature 1

```
public static IQueryable<TSource> ThenByDescending<TSource, TKey>(
    this IQueryable<TSource> source,
    Expression<Func<TSource, TKey>> keySelector)
```

Parameters

source	Sequence to operate on
keySelector	Key to sort on

Example


```
var orderedAddressThenByDescending =
    ctx.Addresses
        .OrderBy(
            addr => addr.CountryRegion
        )
        .ThenByDescending(
            addr => addr.StateProvince
        );
```

Providers LINQ to Objects, LINQ to SQL

Signature 2

```
public static IEnumerable<TSource> ThenByDescending<TSource, TKey>(
    this IEnumerable<TSource> source,
    Func<TSource, TKey> keySelector,
    IComparer<TKey> comparer)
```

Parameters

source	Sequence to operate on
keySelector	Key to sort on
comparer	Custom comparer object

Example

```
var orderedReviewsThenByDescending =
    productReviewers
        .OrderBy(
            pRev => pRev.Specialty
        )
        .ThenByDescending(
            pRev => pRev.Name
        );
```

Provider LINQ to Objects

ToArray

Description Converts collection to an *Array*.

Signature

```
public static TSource[] ToArray<TSource>(
    this IEnumerable<TSource> source)
```

Parameter

source	Sequence to operate on
--------	------------------------

Example

```
var reviewerArray = productReviewers.ToArray();
```

Provider LINQ to Objects

ToDictionary

Description Converts sequence to a *Dictionary*, where there is a one-to-one relationship between key and value.

Signature 1

```
public static Dictionary<TKey, TSource> ToDictionary<TSource, TKey>(
    this IEnumerable<TSource> source,
    Func<TSource, TKey> keySelector)
```

Parameters

source	Sequence to operate on
keySelector	Dictionary key associated with value

Example

```
var reviewerDictionary =
    productReviewers
    .ToDictionary(
        pRev => pRev.Name
    );

var revDict = reviewerDictionary["Tony"];
```

Provider LINQ to Objects

Signature 2

```
public static Dictionary<TKey, TSource> ToDictionary<TSource, TKey>(
    this IEnumerable<TSource> source,
    Func<TSource, TKey> keySelector,
    IEqualityComparer<TKey> comparer)
```

Parameters

source	Sequence to operate on
keySelector	Dictionary key associated with value
comparer	Compares keys

Example

```
var reviewerDictionaryCmp =
    productReviewers
    .ToDictionary(
```

```
pRev => pRev.Name,  
new ReviewerSpecialtyComparer()  
);
```

Provider LINQ to Objects

Signature 3

```
public static Dictionary<TKey, TElement> ToDictionary<TSource, TKey,  
TElement>(  
    this IEnumerable<TSource> source,  
    Func<TSource, TKey> keySelector,  
    Func<TSource, TElement> elementSelector)
```

Parameters

source	Sequence to operate on
keySelector	Dictionary key associated with value
elementSelector	Custom projection on the value

Example

```
var reviewerDictProj =  
    productReviewers  
    .ToDictionary(  
        pRev => pRev.Name,  
        pRev =>  
            new  
            {  
                Name = pRev.Name,  
                Reviews = pRev.Reviews  
            }  
    );
```

Provider LINQ to Objects

Signature 4

```
public static Dictionary<TKey, TElement> ToDictionary<TSource, TKey, TElement>(  
    this IEnumerable<TSource> source,  
    Func<TSource, TKey> keySelector,  
    Func<TSource, TElement> elementSelector,  
    IEqualityComparer<TKey> comparer)
```

Parameters

source	Sequence to operate on
keySelector	Dictionary key associated with value
elementSelector	Custom projection on the value

comparer	Compares keys
----------	---------------

Example

```
var reviewerDictProjCmp =
    productReviewers
        .ToDictionary(
            pRev => pRev.Name,
            pRev =>
                new
                {
                    Name = pRev.Name,
                    Reviews = pRev.Reviews
                },
            new ReviewerSpecialtyComparer()
        );
```

Provider LINQ to Objects

ToList

Description Converts collection to a *List<T>*.

Signature

```
public static List<TSource> ToList<TSource>(
    this IEnumerable<TSource> source)
```

Parameter

source	Sequence to operate on
--------	------------------------

Example

```
var reviewersList = productReviewers.ToList();
```

Provider LINQ to Objects

ToLookup

Description Converts sequence to a *Lookup<TKey, TValue>*, where there is a one-to-many relationship between key and value. The relationship between key and value is what distinguishes *ToDictionary* and *ToLookup*. *ToDictionary* is one-to-one, while *ToLookup* is one-to-many.

Signature 1

```
public static ILookup<TKey, TSource> ToLookup<TSource, TKey>(
    this IEnumerable<TSource> source,
    Func<TSource, TKey> keySelector)
```

Parameters

source	Sequence to operate on
--------	------------------------

keySelector	Lookup key associated with value
-------------	----------------------------------

Example

```
var specialtyLookup =
    productReviewers
    .ToLookup(
        pRev => pRev.Specialty
    );

specialtyLookup["Food"]
    .ToList()
    .ForEach(
        spec => Console.WriteLine(spec.Name)
    );
```

Provider LINQ to Objects

Signature 2

```
public static ILookup<TKey, TSource> ToLookup<TSource, TKey>(
    this IEnumerable<TSource> source,
    Func<TSource, TKey> keySelector,
    IEqualityComparer<TKey> comparer)
```

Parameters

source	Sequence to operate on
keySelector	Lookup key associated with value
comparer	Compares keys

Example

```
var specialtyLookupCompared =
    productReviewers
    .ToLookup(
        pRev => pRev.Specialty,
        new ReviewerSpecialtyComparer()
    );
```

Provider LINQ to Objects

Signature 3

```
public static ILookup<TKey, TElement> ToLookup<TSource, TKey, TElement>(
    this IEnumerable<TSource> source,
    Func<TSource, TKey> keySelector,
    Func<TSource, TElement> elementSelector)
```

Parameters

source	Sequence to operate on
--------	------------------------

keySelector	Lookup key associated with value
elementSelector	Custom projection on the value

Example

```
var specialtyLookupProjection =
    productReviewers
    .ToLookup(
        pRev => pRev.Specialty,
        pRev =>
            new
            {
                Name = pRev.Name,
                Reviews = pRev.Reviews
            }
    );
```

Provider LINQ to Objects

Signature 4

```
public static ILookup<TKey, TElement> ToLookup<TSource, TKey, TElement>(
    this IEnumerable<TSource> source,
    Func<TSource, TKey> keySelector,
    Func<TSource, TElement> elementSelector,
    IEqualityComparer<TKey> comparer)
```

Parameters

source	Sequence to operate on
keySelector	Lookup key associated with value
elementSelector	Custom projection on the value
comparer	Compares keys

Example

```
var specialtyLookupProjectionCompared =
    productReviewers
    .ToLookup(
        pRev => pRev.Specialty,
        pRev =>
            new
            {
                Name = pRev.Name,
                Reviews = pRev.Reviews
            },
        new ReviewerSpecialtyComparer()
    );
```

Provider LINQ to Objects

Union

Description Combines all values from two sequences.

Signature 1

```
public static IQueryable<TSource> Union<TSource>(  
    this IQueryable<TSource> source1,  
    IEnumerable<TSource> source2)
```

Parameters

source1	First sequence
source2	Second sequence

Example

```
var unionProducts =  
    bikePurchasers  
    .Union(  
        helmetPurchasers  
    );
```

Providers LINQ to Objects, LINQ to SQL

Signature 2

```
public static IEnumerable<TSource> Union<TSource>(  
    this IEnumerable<TSource> first,  
    IEnumerable<TSource> second,  
    IEqualityComparer<TSource> comparer)
```

Parameters

first	First sequence
second	Second sequence
comparer	Custom compare object

Example

```
var unionProdRevProdPurch =  
    productReviewers  
    .Union(  
        productPurchasers  
    );
```

Provider LINQ to Objects

Where

Description Filters sequence results.

Signature 1

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

Parameters

source	Sequence to operate on
predicate	Condition to filter on

Example

```
var canadianAddresses =
    ctx.Addresses.Where(
        addr => addr.CountryRegion == "Canada"
    );
```

Providers LINQ to Objects, LINQ to SQL

Signature 2

```
public static IEnumerable<TSource> Where<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, int, bool> predicate)
```

Parameters

source	Sequence to operate on
predicate	Condition to filter on—including index of current item

Example

```
var foodReviewers =
    productReviewers.Where(
        (pRev, row) =>
            pRev.Specialty == "Food" &&
            row < 1
    );
```

Provider LINQ to Objects

Index

{ } (curly braces), [22](#), [23](#), [190](#)

() parenthesis, [190](#), [191](#)

=> operator, [23](#)

== operator, [48](#)

&& (and) operator, [40](#)

|| (or) operator, [40](#)

A

Add methods, [22](#), [130](#)–131, [167](#)

AddAfterSelf method, [167](#)

AddBeforeSelf method, [167](#)

AddFirst method, [167](#)

ADO.NET

 Data Model, [243](#)

 Data Services, [133](#)

 security, [91](#)

 via LINQ to DataSet, [111](#)–118

ADO.NET Entity Framework (AEF) architecture, [121](#)–122

AdventureWorks database, [4](#), [236](#)

AdventureWorks.edmx file, [126](#)

AdventureWorksLT database, [4](#), [68](#), [69](#)

AdventureWorksLT database objects, [112](#)–113

AdventureWorksLT database schema, [69](#), [70](#)

AEF (ADO.NET Entity Framework) architecture, [121](#)–122

Aggregate operator, [290](#), [292](#)

AggregateException keyword, [282](#)–283

All operator, [293](#)–294

Always value, [257](#)

Amdahl's Law, [275](#)

ancestors, [163](#)–164

Ancestors axis method, [163](#)–164

AncestorsAndSelf method, [164](#)

AND conditions, [198](#)

and (&&) operator, [40](#)

anonymous delegates, [23](#)

- anonymous methods, [22](#), [23](#), [54](#)
- anonymous types, [26](#)–28
 - as composite keys, [46](#)
 - considerations, [27](#)–28
 - limiting number of parameters with, [91](#)–92
 - overview, [26](#)
 - projecting into, [37](#)–38
 - simple, [26](#)
- Any* operator, [294](#)–295
- application design, [235](#)–269
 - Business Logic Layer, [243](#)–247
 - Data Access Layer, [242](#)–243
 - data concurrency, [254](#)–258
 - deferred execution, [258](#)–259, [267](#)–269
 - deferred loading, [259](#)–269
 - immediate loading, [262](#)–265, [267](#)
 - LinqDataSource control, [236](#)–241
 - n-layer architecture, [241](#)–254
 - rapid application development, [236](#)
 - User Interface layer, [247](#)–254
- ArrayList* keyword, [58](#)–59
- ascending* keyword, [44](#)
- ascending order, [43](#)–44
- AsEnumerable* operator, [295](#)
- AsOrdered* method, [276](#)–277
- AsParallel* method, [274](#)–277, [279](#), [283](#)
- ASP.NET, [221](#), [236](#)–241
- ASP.NET WebForm, [236](#)–241
- AsQueryable* operator, [295](#)
- Assembly language, [5](#)
- AsSequential* method, [276](#)
- AssociateWith* method, [265](#), [266](#)
- Association* attributes, [88](#)–89
- associations, creating, [139](#)–140
- AsUnordered* method, [276](#), [277](#)
- Attach* method, [99](#)–101

AttributeMappingSource class, [183](#)

attributes

- association, [88](#)–89

- filtering data by, [162](#)–163

- LINQ to SQL, [81](#)–82

auto-generated files, [10](#)

auto-implemented properties, [5](#)–8

Average operator, [296](#)

axes, [163](#)

axis method filters, [166](#)

axis methods, [163](#)–166

axis nodes, [163](#)–164

B

backing stores, [6](#)–9

base URI, [155](#)

base URL, [221](#), [231](#)

Between method, [213](#)–214

BinaryExpression type, [195](#), [196](#), [204](#)

BLL (Business Logic Layer), [241](#), [243](#)–247

BLL objects, [247](#)

Body property, [195](#)–196

bool parameter, [99](#), [100](#)

braces { }, [22](#), [23](#)

Browsable attribute, [129](#)

BuildUrl method, [230](#)–231

Business Logic Layer (BLL), [241](#), [243](#)–247

C

C# 3.0 language, [4](#)

C# compiler, [8](#), [11](#), [14](#)–17

C# *decimal* type, [12](#), [13](#)

C# identifiers, [8](#)

C# language, [4](#), [177](#)

Cast operator, [297](#)

CategoryAndProducts custom type, [63](#)–64

CData text, [159](#)

ChangeConflictException keyword, [256](#), [257](#)

ChangeConflicts keyword, [257](#)

Chen, Pin-Shan (Peter), [120](#), [121](#)

child objects, [262](#), [265](#)–267

classes

AttributeMappingSource, [183](#)

CustomerManager, [243](#)–244, [247](#)–251

DataContext, [73](#)

DateExtensions class, [213](#)

Enumerable, [219](#), [274](#)

Expression, [193](#)–195

Framework Class Library, [218](#)–219

GetInnermostWhere, [229](#)

InnermostWhereFinder, [229](#)

partial, [73](#)

Query, [226](#)

Queryable, [218](#), [219](#), [227](#)

Reviewer, [286](#)–287

ReviewerKey, [287](#)–288

static, [13](#), [213](#)–214

Status, [217](#)–218

TransactionOptions, [255](#)

TransactionScope, [255](#)

TwitterContext, [221](#), [227](#)–233

TwitterQueryable, [225](#), [227](#)

XAttribute, [150](#)–152

XContainer, [150](#), [151](#)

XDocument, [157](#)–158

XElement, [150](#)–155, [233](#)

XmlMappingSource, [182](#)–183

Click event, [23](#)

`/code` option, [175](#)

CodePlex, [216](#)

collection initializers, [20](#)–22

collection object projections, [36](#)

collections

Dictionary, [21](#)–22

entity, [79](#)–80

example, [34](#)–35

non-*IEnumerable*<*T*>, [58](#)–59

searchTerms, [198](#)–202

System.Collections.Generic, [228](#)

Column attribute, [82](#)

comma-separated lists, [190](#), [191](#)

compareStr parameter, [13](#), [14](#)

CompareTo method, [13](#)–14

Compile method, [193](#)

composite keys, [46](#), [47](#), [51](#)–54

composition, modularized, [267](#)–269

Concat operator, [297](#)

Concatenation operator, [290](#)

Conceptual Schema Definition Language (CSDL), [121](#)–122, [135](#), [141](#)–144

concurrency, data, [254](#)–258

ConcurrencyConflictException, [257](#)

concurrent programming, [271](#)–283

configuration files, [276](#)

Configure ListView editor, [250](#)

ConflictMode enum parameter, [257](#)

/conn option, [173](#)

connection strings, [73](#)–76, [172](#), [173](#)

connections

closing, [76](#)

dynamically specifying, [73](#)–74

managing with DataContext, [73](#)–76

via SqlMetal.exe, [172](#)–173

Console.WriteLine statement, [13](#)–14

Contains method, [96](#)

Contains operator, [298](#)–299

context objects, [237](#), [239](#)

/context option, [177](#)

continuation clause, [47](#)

continuation variable, [94](#)

ContinueOnConflict keyword, [257](#)

Conversion operators, [290](#)

Count operator, [299](#)

CPU

- load, [273](#)–274

- PLINQ performance testing, [277](#)–282

- speed of, [280](#)

create, read, update, and delete (CRUD) operations, [68](#), [113](#)

CreatedAt property, [233](#)

CreateDatabase method, [83](#), [84](#)

createdAtDate string, [233](#)

CreateOrExpression method, [202](#), [206](#)

CreateQuery method, [224](#), [227](#)

credentials, [221](#), [227](#)–228, [233](#)

cross-joins, [57](#), [95](#)–96

CRUD (create, read, update, and delete) operations, [68](#), [113](#)

CSDL (Conceptual Schema Definition Language), [121](#)–122, [135](#), [141](#)–144

curly braces { }, [22](#), [23](#), [190](#)

custom business objects, [181](#)–182

custom types, [39](#)–40, [63](#)–64

Customer entity objects, [244](#)

CustomerManager class, [243](#)–244, [247](#)–251

D

DAL (Data Access Layer), [241](#), [242](#)–243

data

- deleting, [98](#)–99

- example, [4](#)

- extracting from UI controls, [59](#)–65

- filtering. *See* filtering data inserting, [97](#)

- locking strategies, [254](#)

- selecting columns for display, [238](#), [240](#)

- updating, [98](#)

Data Access Layer (DAL), [241](#), [242](#)–243

data binding, [247](#)–248

data concurrency, [254](#)–258

data object methods, [251](#)

data return formats, [232](#)

data source communication, [219](#)–221

Data Source Configuration Wizard, [236](#)–241, [248](#), [249](#)

data sources

- choosing, [237](#), [238](#)

- location of, [221](#)

- querying, [59](#)–65

DataAdapter, [118](#)

Database attribute, [81](#)–82

database connections

- closing, [76](#)

- dynamically specifying, [73](#)–74

- managing with DataContext, [73](#)–76

- via SqlMetal.exe, [172](#)–173

Database Markup Language. *See* DBML

/database option, [172](#)

Database Read-only property, [69](#)

databases

- AdventureWorksLT, [4](#)

- connections. *See* database connections

- creating, [82](#)–83

- deleting records from, [98](#)–99, [246](#)–247

- locking records, [254](#)

- mapping entities to, [142](#)–146

DataContext class, [73](#)

DataContext entity representation, [77](#)–79

DataContext Log, [206](#)–207

DataContext objects, [70](#)–84

- creating with LINQ to SQL Designer, [70](#)–73

- database connection management, [73](#)–76

- described, [70](#)

- entity relationships, [84](#)–89

- generated SQL, [91](#)–92

- instantiating, [73](#), [74](#), [76](#), [83](#)

LINQ to SQL, [236](#)–237

loading external mapping files with, [182](#)–183

object mapping, [76](#)–83

object tracking, [83](#)

queries, [90](#)–91

showing, [237](#)

viewing code for, [71](#)–73

DataContext services, [73](#)–84

DataContract attribute, [132](#), [178](#)

DataContract serialization, [178](#)

DataLoadOptions attribute, [263](#), [265](#)

DataLoadOptions instance, [264](#)

DataMember attribute, [178](#)

DataMemberAttribute attribute, [134](#)

DataObject attribute, [243](#), [245](#), [248](#)

DataObjectMethod attribute, [244](#), [245](#), [246](#), [249](#)

DataObjectMethodType enum, [244](#)

DataRow results, [113](#)–114

DataSet tables, [113](#)–114, [118](#)

DataSets

LINQ to DataSet, [111](#)–118

modifying, [114](#)–115

querying, [113](#)–115, [115](#)–117

setting up, [112](#)–113

special operations, [117](#)–118

strongly typed, [115](#)–117

DataViews, [117](#)

DateExtensions class, [213](#)

dateParts array, [233](#)

DateTime struct, [213](#)–214

DBML (Database Markup Language), [172](#)

.dbml extension, [174](#)

DBML files

creating, [172](#), [173](#)

described, [172](#), [174](#)

working with, [174](#)–177

/dbml option, [172](#), [173](#)

decimal type, [12](#), [13](#)

declarations, [158](#)–159

declarative programming approach, [32](#), [33](#)

DefaultIfEmpty operator, [49](#)–50, [300](#)–301

deferred execution, [26](#), [258](#)–259, [267](#)–269

deferred loading, [259](#)–269

DeferredLoading, [264](#)

DeferredLoadingEnabled, [262](#)

degreeOfParallelization parameter, [275](#), [283](#)

delegates, [23](#)

delete operation, [98](#)–99, [246](#)–247

DeleteObject method, [131](#)

DeleteOnSubmit method, [99](#)

deleting database records, [246](#)–247

DescendantsAndSelf method, [165](#)

descendents, [164](#)–165

Descendents axis method, [164](#)–165

descending keyword, [44](#)

descending order, [43](#)–44

deserialization, [177](#), [178](#)

Dictionary collections, [21](#)–22

Distinct operator, [57](#), [301](#)–302

DivideByZeroException, [283](#)

dot (instance member) operator, [13](#)

dual-core/single-processor CPU, [280](#), [281](#)

dynamic queries

 building, [199](#)–205

 executing, [205](#)–207

DynamicInvoke, [230](#)

E

EDM (Entity Data Model), [122](#)–134. *See also* Entity Relationship (ER) Model

 accessing via *ObjectContext*, [126](#)–131

 creating associations, [139](#)–140

 creating entities, [135](#)–141

defining database connection for, [142](#)–144

deleting database data, [131](#)

entities representing objects, [138](#)–139

entity types, [132](#)–134

implementing inheritance, [140](#)–141

managing, [125](#)–126

mapping entities to database, [142](#)–146

querying with LINQ to Entities, [147](#)

setting up, [135](#)

top-down design, [134](#)–146

updating existing data, [131](#)

EDM constructors, [127](#)–128

EDM entities, [128](#)–130

EdmComplexPropertyAttribute attribute, [134](#)

EdmScalarPropertyAttribute attribute, [134](#)

Element operators, [290](#)

ElementAt operator, [302](#)

ElementAtOrDefault operator, [302](#)–303

ElementsAfterSelf method, [165](#)–166

ElementsBeforeSelf method, [165](#)–166

ElementType property, [222](#)

Empty operator, [303](#)

encapsulation, [5](#)

#endregion directive, [21](#)

entities

adding to DataContext, [84](#)–85

associations, [85](#)–87

deferred loading, [259](#)–269

described, [68](#), [138](#)

inheritance, [104](#)–107

mapping to database, [142](#)–146

multiple relationships between, [87](#)–88

overview, [120](#)–121

primary keys, [82](#), [83](#)

returning IDs of, [245](#)–246

entity collections, [79](#)–80

- Entity Data Model (EDM), [122](#)–134
- Entity Data Model Wizard, [243](#)
- entity declarations, [80](#)–81
- entity notifications, [11](#)
- entity properties, [81](#)
- Entity Relationship (ER) Model, [120](#)–121. *See also* EDM
- entity relationships, [84](#)–89
- entity representation, [77](#)–79
- enum types, [195](#)
- Enumerable* class, [219](#), [274](#)
- enumeration, [225](#), [227](#), [229](#)
- Equality operator, [290](#)
- equals* keyword, [48](#)
- ER (Entity Relationship) Model, [120](#)–121. *See also* EDM
- errors, [161](#)–162
- Evaluator.cs file, [220](#)
- EventHandler* delegate parameters, [23](#)
- example collections, [34](#)–35
- example data, [4](#)
- example database, [68](#)–70
- Except* operator, [303](#)–304
- exceptions
 - optimistic concurrency, [256](#)
 - PLINQ, [282](#)–283
- Execute* method, [224](#), [229](#), [230](#)
- ExecuteCommand*, [104](#)
- ExecuteQuery* command, [103](#)–104
- execution, deferred, [26](#), [258](#)–259, [267](#)–269
- Expression* class, [193](#)–195
- expression nodes, [196](#)–197
- expression trees, [193](#)–207
 - converting lambdas to, [192](#)–197
 - converting to lambdas, [192](#)–197
 - creating, [199](#)–205
 - extracting lambda data from, [195](#)–197
 - object diagram, [205](#)

overview, [193](#)–195

types, [193](#)–195

using with LINQ queries, [197](#)–207

Where Or problem, [198](#)–199

Expression type, [195](#)

expression types, [193](#)–195

Expression.Or type, [204](#)

expressions, [22](#)–24

ExpressionTreeHelper.cs file, [220](#)

ExpressionTreeVisitor.cs file, [220](#)

ExpressionType type, [197](#)

extension methods, [12](#)–17

 chaining together, [212](#)

 composability of, [211](#)–212

 constructing code with, [209](#)–214

 creating, [12](#), [213](#)–214

 custom, [213](#)–214

 matching query expressions, [210](#)–212

 precedence, [13](#)–14

 static classes, [213](#)–214

 using, [13](#)

ExtensionAttribute attribute, [17](#)

extensions, [174](#), [177](#)

external mapping files, [172](#), [178](#)–183

F

FailOnFirstConflict, [257](#)

FCL (Framework Class Library), [218](#)–219

FCL types, [218](#)–219

feature set, [218](#)

files

 auto-generated, [10](#)

 configuration, [276](#)

 DBML, [172](#)–177

 filtering by name, [174](#)

 mapping, [73](#), [172](#), [178](#)–183

reading XML from, [154](#)–155

XML. *See* XML documents

Filter operators, [290](#)

filtering child objects, [265](#)–267

filtering data, [40](#)–41

on application IDs, [98](#)

by attributes, [162](#)–163

by file name, [174](#)

First operator, [304](#)–305

FirstOrDefault operator, [305](#)–306

flattening object hierarchies, [55](#)–56

floating point array type, [59](#)

floating point value, [59](#)

foreach loop, [47](#), [90](#)–91, [92](#), [199](#)

Framework Class Library (FCL), [218](#)–219

friend queries, [216](#), [217](#), [228](#)

friend requests, [231](#)

from clause, [25](#)

Func delegates, [190](#)–192

functional construction, [151](#)–152

functions

LINQ to SQL, [102](#)–103

/functions option, [174](#)

G

GAC (Global Assembly Cache), [272](#)

Generation operators, [291](#)

get accessors, [6](#)–8, [81](#)

GetCustomerDeferred method, [269](#)

GetCustomerID method, [268](#)

GetCustomers method, [244](#)–245, [249](#)

GetDollars extension method, [12](#)–15

GetEnumerator method, [229](#)

GetEnumerator overloads, [225](#)

GetInnermostWhere class, [229](#)

GetStatusList method, [230](#)

Global Assembly Cache (GAC), [272](#)

group by clause, [44](#)–46

group by statement, [92](#)

group joins, [50](#)–51

GroupBy operator, [306](#)–312

grouping

- described, [44](#)

- into hierarchies, [44](#)–47

- LINQ to SQL, [92](#)–93

- by multiple properties, [45](#)–46

- objects, [47](#)

- sets, [44](#)–47

- by single property, [44](#)–45

Grouping operators, [291](#)

GroupJoin operator, [312](#)–314

H

hierarchies, [44](#)–47

HttpRequest, [233](#)

I

ID property, [26](#), [27](#)

SqlConnection, [73](#)

IEnumerable<T> types, [36](#)–37, [219](#)

IL (Intermediate Language), [4](#), [5](#)

IL Disassembler (ILDASM), [5](#), [11](#)

ILDASM (IL Disassembler), [5](#), [11](#)

immediate loading, [262](#)–265, [267](#)

imperative programming approach, [32](#), [33](#)

implementing methods, [9](#)–10

implicitly typed local variables, [24](#)

IN queries, [96](#)

inheritance

- described, [138](#)

- entity, [104](#)–107

- implementing, [140](#)–141

mapping relationships, [145](#)

Inheritance object, [141](#)

initialization, [17](#), [24](#)

initializers

collection, [20](#)–22

object, [17](#)–20

inner joins, [48](#)–49, [93](#)

InnerExceptions, [283](#)

InnermostWhereFinder class, [229](#)

Insert method, [245](#)–246

InsertAllOnSubmit method, [97](#)

InsertOnSubmit method, [97](#)

instance member (dot) operator, [13](#)

IntelliSense support, [102](#)

Intermediate Language (IL), [4](#), [5](#)

Intersect operator, [314](#)–315

InvalidOperationException, [98](#)

InvalidQueryException.cs file, [220](#)

IOrderedQueryable<T> interface, [218](#), [224](#)

IParallelEnumerable extension methods, [273](#)

IQueryable<T> interface

exposing, [228](#)–229

implementing, [224](#)–225

overview, [218](#), [222](#)–223

IQueryProvider interface

implementing, [226](#)–227

overview, [223](#)–224

IsolationLevel enum, [255](#)

IsValid keyword, [161](#)–162

J

join clause, [48](#)–49

Join operator, [291](#), [315](#)–316

joining objects, [48](#)–57

with composite keys, [51](#)–54

cross-joins, [57](#), [95](#)–96

- example, [48](#)–49
- group joins, [50](#)–51
- inner joins, [48](#)–49, [93](#)
- left joins, [49](#)–50
- left outer joins, [93](#), [94](#)
- LINQ to SQL joins, [93](#)–94
- overview, [48](#)–49
- select many* joins, [55](#)–57

JSON option, [232](#)

K

- KeepChanges* value, [257](#)
- KeepCurrentValues* value, [257](#)–258
- key/value pairs, [22](#)

L

- lambda expressions, [22](#)–24
 - components, [195](#)
 - considerations, [24](#)
 - converting expression trees to, [192](#)–197
 - converting to expression trees, [192](#)–193
 - examples, [188](#)–190
 - extracting data from expression trees, [195](#)–197
 - Func* delegates, [190](#)–192
 - replacing anonymous methods/delegates, [23](#)
 - using, [22](#)–23
 - working with, [188](#)–192
- lambda* method, [37](#)
- LambdaExpression*, [195](#), [202](#), [204](#), [205](#)
- Language Integrated Query. *See* LINQ
- /language* option, [177](#)
- Last* operator, [317](#)
- Left* expression, [196](#)
- left joins, [49](#)–50
- left outer joins, [93](#), [94](#)
- let* clause, [41](#)–42, [233](#)

Like method, [199](#)

LINQ (Language Integrated Query)

- custom LINQ providers, [215](#)–234

- described, [4](#)

- designing applications with. *See* application design

- example data, [4](#)

- introduction to, [3](#)–39

- n-layer architecture, [241](#)–254

- third-party providers, [216](#)

LINQ providers, [215](#)–234

- data source communication, [219](#)–221

- development process, [218](#)–222

- FCL types, [218](#)–219

- implementation types, [219](#)

- implementing interfaces, [222](#)–227

- .NET Framework interfaces, [218](#)–219

- resources, [221](#)–222

LINQ queries. *See* queries

LINQ to DataSet

- ADO.NET, [111](#)–118

- data source communication, [219](#)

LINQ to Entities, [119](#)–148

- AEF architecture, [121](#)–122

- data source communication, [221](#)

- Entity Data Model (EDM), [122](#)–134

- introduction to, [120](#)–121

- overview, [120](#)

- querying EDM with, [147](#)

LINQ to MySLQ, [216](#)

LINQ to NHibernate, [216](#)

LINQ to Objects, [31](#)–65. *See also* objects

- advantages of, [32](#)–33

- calculating intermediate values, [41](#)–42

- data source communication, [219](#)

- example collections, [34](#)–35

- example setup, [286](#)–288

extracting data from UI controls, [59](#)–65

filtering data, [40](#)–41

grouping sets, [44](#)–47

IEnumerable<T>, [36](#)–37

implementing projections, [35](#)–41

joining objects, [48](#)–57

overview, [32](#)–33

practical example of, [59](#)–65

projecting into anonymous types, [37](#)–38

querying data sources, [59](#)–65

selecting single field/property, [37](#)

sorting query results, [42](#)–44

LINQ to Objects queries, [36](#)–37

LINQ to Oracle, [216](#)

LINQ to SQL, [67](#)–107

attaching objects, [99](#)–101

attributes, [81](#)–82

code generation with SqlMetal, [171](#)–184

data source communication, [219](#)

database connection management, [73](#)–76

DataContext object creation, [70](#)–84

deleting data, [98](#)–99

entity collections, [79](#)–80

entity declarations, [80](#)–81

entity inheritance, [104](#)–107

entity properties, [81](#)

entity relationships, [84](#)–89

entity representation, [77](#)–79

example database, [68](#)–70

example setup, [289](#)

functions, [102](#)–103

generated SQL, [91](#)–92

inserting data, [97](#)

n-layer architecture, [241](#)–242

object mapping, [76](#)–83

Object Relational Designer, [104](#)–107

object tracking, [83](#)–84

overview, [68](#)

queries, [89](#)–96

raw SQL, [103](#)–104

reading *ListView* controls with, [64](#)–65

security, [91](#)

stored procedures, [101](#)–102

updating data, [98](#)

LINQ to SQL DataContext, [236](#)–237

LINQ to SQL Designer, [70](#)–76

LINQ to TerraServer, [229](#)

LINQ to TerraServer provider, [221](#)

LINQ to TerraServer sample, [219](#)

LINQ to TerraServer web site, [221](#)

LINQ to Twitter, [215](#)–234. *See also* LINQ providers; Twitter

data source communication, [219](#)–221

interfaces, [218](#)–219, [222](#)–227

introduction to, [216](#)–218

resources, [221](#)–222

URLs, [221](#), [230](#)–231

XML option, [232](#)–233

LINQ to XML, [149](#)–169. *See also* XML documents

adding raw text, [159](#)

ancestors, [163](#)–164

attributes, [150](#)

axis method filters, [166](#)

axis methods, [163](#)–166

CData text, [159](#)

data source communication, [221](#)

declarations, [158](#)–159

descendents, [164](#)–165

element after/before self, [165](#)–166

elements, [150](#)–151

functional construction, [151](#)–152

namespaces, [151](#), [156](#)–157

objects, [150](#)–151

overview, [150](#)

processing instructions, [160](#)

queries, [162](#)–166

retrieving XML, [153](#)–155

Twitter, [232](#)–233

XML strings, [153](#)–154

LINQ transactions, [255](#)

LinqDataSource control, [236](#)–241

LinqDataSource wizard, [236](#)–241

LinqProgrammingExtensions type, [14](#), [15](#)

LinqToSqlMapping.xml schema, [182](#)

LinqToTwitter namespace, [216](#)

LinqToTwitter.dll assembly, [216](#)

LinqToTwitter.dll project, [216](#), [219](#)

lists

comma-separated, [190](#), [191](#)

controls, [39](#)

ListView controls

configuring, [250](#), [252](#)

data binding, [247](#)–248

nested, [60](#)–62

reading with LINQ to SQL, [64](#)–65

selecting new data source, [236](#)–237

ListView form, [60](#)–62

LiveView control, [236](#)–237, [238](#)

Load method, [154](#)–155, [158](#), [233](#)

loading

deferred, [259](#)–269

immediate, [262](#)–265, [267](#)

LoadOptions approach, [264](#)–265

LoadOptions enum, [155](#)

LoadOptions property, [264](#)

LoadWith code, [266](#)

LoadWith generic method, [263](#)

local variables, [24](#), [36](#)

locking database records, [254](#)

Log property, [91](#)–92

login errors, [84](#)

LongCount operator, [318](#)–319

LongProcess method, [273](#)–274

loops

 example, [33](#)

foreach, [47](#), [90](#)–91, [92](#)

M

/map option, [179](#)

mapping files, [73](#), [172](#), [178](#)–183

Mapping Schema Language (MSL), [121](#)–122, [141](#)–146

MappingSource, [73](#), [183](#)

materialization, [259](#)

Max operator, [319](#)–320

MemberBinding type, [195](#)

MemberChangeConflict, [257](#)

MemberExpression, [202](#)

methods

 anonymous, [22](#), [23](#)

CompareTo, [13](#)–14

 extension, [12](#)–17

 implementing, [9](#)–10

 partial, [8](#)–11, [73](#)

 static, [13](#)–15, [17](#)

Min operator, [320](#)–321

Model entity, [145](#)

Model parameter, [92](#)

modifiers, [9](#), [10](#)

modularized composition, [267](#)–269

MSDN Code Gallery, [36](#)

MSDN Parallel Computing Developer Center, [272](#)

MSL (Mapping Schema Language), [121](#)–122, [141](#)–146

multicore/multiprocessor CPU, [272](#), [273](#), [274](#)–275, [280](#)

N

- name* argument, [14](#)
- Name* property, [26](#), [27](#), [145](#), [199](#), [202](#)
- /namespace* option, [177](#)
- namespace prefixes, [156](#)–157
- namespaces, [151](#), [156](#)–157
- navigation properties, [139](#)–140
- nested items
 - ListView* controls, [60](#)–62
 - switch* statements, [231](#)
- nested *ListView* form, [60](#)–62
- .NET attributes, [92](#)
- .NET Framework, [4](#), [218](#)–219
- .NET Framework SDK, [5](#), [219](#)
- .NET Languages, [4](#)
- .NET objects, [68](#)
- .NET XML APIs, [150](#), [155](#)
- NetworkCredential*, [233](#)
- Never* value, [257](#)
- new* operator, [26](#), [152](#)
- n-layer architecture, [241](#)–254
- NodeType* property, [196](#)–197
- non-*IEnumerable*<*T*> collections, [58](#)–59

O

- object hierarchies, [55](#)–56
- object IDs, [246](#)
- object initializers, [17](#)–20
- object instantiation, [18](#)–20
- object mapping, [76](#)–83
- Object Relational Designer, [104](#)–107
- object tracking, [83](#)–84
- ObjectContext
 - accessing EDM via, [126](#)–131, [128](#)–130
 - Add* methods, [130](#)–131
 - instantiating, [127](#)–128
 - overview, [126](#)–127

ObjectDataSource control, [151](#), [247](#), [250](#), [252](#), [253](#)

ObjectDumper utility, [36](#)

objects. *See also* LINQ to Objects

AdventureWorksLT, [4](#)

attaching, [99](#)–101

child, [262](#), [265](#)–267

context, [237](#), [239](#)

custom business, [181](#)–182

DataContext. *See* DataContext objects

grouped, [47](#)

immediate loading of, [263](#)–265

joining, [48](#)–57

.NET, [68](#)

parent, [262](#)

programming with LINQ to Entities, [119](#)–148

programming with LINQ to XML, [149](#)–169

showing all, [237](#)

OfType operator, [321](#)

OnCreated method, [73](#)

optimistic concurrency, [254](#), [255](#)–258

optimization, [26](#)

OR operation, [204](#)

or (||) operator, [40](#)

orderby clause, [42](#)–43

OrderBy operator, [321](#)–322

OrderByDescending operator, [322](#)–323

origStr parameter, [13](#), [14](#)

overloads, [99](#)–100, [152](#), [153](#)

OverwriteCurrentValues, [258](#)

P

Parallel Extensions, [272](#)

Parallel LINQ. *See* PLINQ

Parallel suffix, [279](#)

ParallelEnumerable type, [273](#)

parallelism, [273](#)–276

- ParallelQuery* type, [273](#)
- parameter lists, [189](#)–190
- ParameterExpression*, [195](#), [196](#), [202](#)
- ParameterFinder*, [230](#)
- ParameterFinder.cs file, [220](#)
- Parameters* property, [195](#)–196
- parent objects, [262](#)
- parenthesis (), [190](#), [191](#)
- Parse* method, [153](#)–154
- partial class, [73](#)
- partial methods, [8](#)–11, [73](#)
- partial modifiers, [9](#), [10](#)
- PartialEval*, [230](#)
- Partitioning operators, [291](#)
- /password* option, [172](#)–173
- Password* property, [228](#), [233](#)
- performance, [277](#)–282
- pessimistic concurrency, [254](#), [255](#)
- PLINQ (Parallel LINQ), [271](#)–283
 - described, [272](#)
 - exceptions, [282](#)–283
 - hardware considerations, [280](#)–282
 - methods, [276](#)–277
 - ordering results, [276](#)
 - parallelism, [273](#)–276
 - performance testing, [277](#)–282
 - setting up, [272](#)–273
 - system requirements, [272](#)
- PLINQ assembly, [272](#)
- PLINQ queries, [273](#)–280
- PLINQ types, [273](#)
- pluralization, [177](#)
- /pluralize* option, [177](#)
- precedence, [13](#)–14
- primary key, [82](#), [83](#)
- procedures, stored, [101](#)–102

processors

- load, [273](#)–274

- PLINQ performance testing, [277](#)–282

- speed of, [280](#)

prod variable, [24](#)

Product entity, [145](#)

ProductManager class, [62](#)–63

programming

- concurrent, [271](#)–283

- declarative vs. imperative, [32](#), [33](#)

Projection operators, [291](#)

projections

- anonymous types, [37](#)–38

- collection object, [36](#)

- custom types, [39](#)–40, [63](#)–64

- described, [35](#)

- implementing, [35](#)–41

- LINQ to Objects queries, [36](#)–37

properties

- auto-implemented, [5](#)–8

- backing stores, [6](#)–9

- conflict detection via, [256](#)–257

- grouping by multiple properties, [45](#)–46

- grouping by single property, [44](#)–45

- navigation, [139](#)–140

- sorting multiple properties, [43](#)

- sorting single property, [42](#)–43

Properties window, [74](#)

Properties.Settings, [221](#)

property values. *See* backing stores

propertyName parameter, [202](#)

PropertyOrField method, [202](#)

Provider property, [222](#), [223](#)

public queries, [216](#), [217](#), [229](#)–230, [231](#)

public requests, [231](#)

Q

Quantifier operators, [291](#)

queries

- DataContext, [90](#)–91

- deferred execution, [258](#)–259

- dynamic. *See* dynamic queries

- examples, [25](#)

- friend*, [216](#), [217](#)

- LINQ to Objects, [36](#)–37

- LINQ to SQL, [89](#)–96

- LINQ to XML, [162](#)–166

- PLINQ, [273](#)–280

- public*, [216](#), [217](#)

- REST*, [230](#)

- select many*, [55](#)–57

- standard query operator reference, [285](#)–345

- using sequences from, [25](#)–26

- using with expression trees, [197](#)–207

Query class, [226](#)

query continuation, [45](#), [46](#), [49](#), [50](#)

query expressions, [210](#)–212

query providers, [215](#)–234

query results, [42](#)–44

query syntax, [25](#)–26

Queryable class, [218](#), [219](#), [227](#)

querying data sources, [59](#)–65

QueryTwitter method, [232](#)–233

R

RAD (rapid application development), [236](#)

RAD tools, [247](#)

RAD UI, [253](#)

Range extension method, [274](#)

Range operator, [323](#)–324

range variable, [36](#), [41](#), [59](#), [233](#)

rapid application development. *See* RAD raw SQL, [103](#)–104

- Read-only property, [69](#)
- records
 - deleting from databases, [98](#)–99, [246](#)–247
 - locking, [254](#)

- RefreshMode* enum, [257](#)

- #region* directive, [21](#)

- Remove* method, [168](#)–169

- ReportPropertyChanged* event, [134](#)

- ReportPropertyChanging* event, [134](#)

- Representational State Transfer (REST) web service, [221](#)

- Resolve* methods, [257](#)

- REST* query, [230](#)

- REST URL, [221](#)

- REST (Representational State Transfer) web service, [221](#)

- return* keyword, [23](#)

- return* statement, [190](#)

- Reverse* operator, [325](#)

- Reviewer* class, [286](#)–287

- ReviewerKey* class, [287](#)–288

- Right* expression, [196](#)

- RSS feeds, [155](#)

- RSS option, [232](#)

S

- SalesOrderDetail*, [260](#), [261](#), [262](#), [263](#), [265](#)

- SalesOrderHeader*, [260](#), [261](#), [262](#)

- Save* method, [152](#)–153

- SaveChanges* method, [131](#)

- schemas

 - creating, [161](#)–162

 - LinqToSqlMapping.xml, [182](#)

- searchTermParam* parameter, [205](#)

- searchTerms* collection, [198](#)–202

- security

 - ADO.NET, [91](#)

 - LINQ to SQL, [91](#)

security credentials, [221](#), [227](#)–228, [233](#)

select clause, [25](#), [26](#), [45](#)

Select extension method, [36](#)–37

select many clause

 examining SQL for, [94](#)–96

 flattening object hierarchies, [55](#)–56

 performing cross-joins, [57](#)

select many joins, [55](#)–57

select many query, [55](#)–57

Select method, [244](#)–245

Select operator, [190](#), [191](#), [325](#)–326

SelectMany operator, [326](#)–328

SequenceEqual operator, [329](#)

sequences, [25](#)–26

Sequential suffix, [279](#)

Serializable attribute, [133](#)

serialization, [177](#), [178](#)

/serialization option, [177](#), [178](#)

/serialize option, [177](#), [178](#)

Server Explorer, [74](#), [84](#), [101](#)

/server option, [172](#)

servers

 SQL Server, [68](#)–69, [90](#), [92](#), [104](#)

 TerraServer, [219](#), [221](#)

set accessors, [6](#)–8, [81](#)

Set operators, [291](#)

shouldRound, [13](#)

Single operator, [330](#)

single-core/single-processor CPU, [274](#)–275, [279](#), [280](#)

SingleOrDefault operator, [98](#), [331](#)

Skip operator, [332](#)

SkipWhile operator, [332](#)–333

Solution Explorer, [71](#)

sort direction, [43](#)–44

sorting

 multiple properties, [43](#)

single property, [42](#)–43

Sorting operators, [292](#)

/sprocs option, [174](#)

SQL (Structured Query Language)

generated SQL, [91](#)–92

grouping with, [92](#)–93

joins, [93](#)–94

query syntax, [25](#)

raw SQL, [103](#)–104

SQL injection, [91](#)

SQL Server, [68](#)–69, [90](#), [92](#), [104](#)

SQL Server 2005, [68](#)

SQL Server 2005 Express database, [68](#)

SQL Server 2008, [68](#)

SQL Server Express, [68](#)

SQL Server Express Toolkit, [68](#)

SQL Server Management Studio Express, [68](#), [69](#)

SqlCommand objects, [20](#)

SqlConnection object, [73](#)–74

SqlMetal tool, [171](#)–184

custom business objects, [181](#)–182

database connection options, [172](#)–173

described, [172](#)

deserialization, [177](#), [178](#)

external mapping files, [172](#), [178](#)–183

serialization, [177](#), [178](#)

source code options, [175](#), [177](#)–178

working with DBML files, [174](#)–177

SSDL (Storage Schema Definition Language), [121](#)–122, [135](#), [141](#)–146

standard query operator reference, [285](#)–345

static classes, [13](#), [213](#)–214

static methods, [13](#)–15, [17](#)

Status class, [217](#)–218

Status objects, [232](#)–233

Status property, [228](#)–229

status requests, [231](#)

- Status.cs file, [220](#)
- Storage Schema Definition Language (SSDL), [121](#)–122, [135](#), [141](#)–146
- stored procedures, [101](#)–102
- StreamReader*, [154](#)
- streams, reading XML from, [154](#)
- StringReader*, [154](#)
- strings
 - lambda expressions, [189](#)
 - XML, [153](#)–154
- Structured Query Language (SQL), [25](#)
- SubmitChanges* method, [97](#)–99, [257](#)
- switch* statement, [231](#)
- System* namespace, [228](#)
- System.Collections.Generic* namespace, [228](#)
- System.Data.Linq* namespace, [183](#)
- System.Data.Linq.Mapping* namespace, [183](#)
- System.IO* namespace, [228](#)
- System.Linq* namespace, [183](#), [244](#)
- System.Linq.Expressions* namespace, [192](#), [193](#)–194, [228](#)
- System.Net* namespace, [228](#)
- System.System.Linq* namespace, [228](#)
- System.Threading* namespace, [282](#)
- System.Threading.dll assembly, [272](#)–273
- System.Transactions* namespace, [255](#)
- System.Xml.Linq* namespace, [228](#)

T

- Table* attribute, [82](#)
- TableAdapter, [118](#)
- tables
 - DataSet, [113](#)–114, [118](#)
 - mapping EDM entities to, [142](#)–146
- Take* operator, [95](#), [333](#)
- TakeWhile* operator, [334](#)–335
- ternary operators, [189](#)
- TerraServer, [219](#), [221](#)

- text, CData, [159](#)
- text, raw, [159](#)
- TextReader*, [154](#), [233](#)
- ThenBy* operator, [335](#)–336
- ThenByDescending* operator, [336](#)–337
- this* modifier, [17](#), [213](#), [214](#)
- threads
 - exceptions on, [282](#)
 - multiple, [274](#)–275, [279](#)–280
 - order of, [276](#)
- Time Stamp* property, [100](#)
- ToArray* operator, [337](#)
- ToDictionary* operator, [337](#)–340
- ToList* method, [27](#), [227](#), [259](#)
- ToList* operator, [340](#)
- ToLookup* operator, [340](#)–343
- top-down design approach, [134](#)–146
- TransactionOptions* class instance, [255](#)
- TransactionScope* class, [255](#)
- TransactionScopeOption* enum, [255](#)
- TSearchObject* type, [202](#)
- Twitter, [215](#)–234. *See also* LINQ to Twitter
 - communicating with, [232](#)–233
 - credentials, [221](#), [227](#)–228, [233](#)
 - data return formats, [232](#)
 - described, [216](#)
- Twitter accounts, [216](#)
- TwitterContext* class, [221](#), [227](#)–233
- TwitterContext* instantiation, [227](#)–228
- TwitterContext.cs* file, [220](#)
- TwitterQueryable* class, [225](#), [227](#)
- TwitterQueryable.cs* file, [220](#)
- TwitterQueryProvider* implementation, [226](#)–227
- TwitterQueryProvider.cs* file, [220](#)
- Type* parameter, [230](#), [231](#)
- type* variable, [230](#)

types

- anonymous, [26](#)–28, [37](#)–38

- custom, [39](#)–40, [63](#)–64

TypeSystem.cs file, [220](#)

U

ufnGetAllCategories function, [102](#)–103

UI controls, extracting data from, [59](#)–65

UI (User Interface) layer, [241](#), [247](#)–254

UML (Unified Modeling Language), [135](#), [136](#)

unidirectional, [178](#)

Unified Modeling Language (UML), [135](#), [136](#)

uniform resource identifiers (URIs), [155](#)

uniform resource locators (URLs)

- base, [221](#), [231](#)

- LINQ to TerraServer, [221](#)

- LINQ to Twitter, [221](#), [230](#)–231

- REST, [221](#)

- The Wayward WebLog, [221](#)

Union operator, [343](#)–344

update method, [246](#)

UpdateCheck attribute, [256](#)–257

URIs (uniform resource identifiers), [155](#)

URLs (uniform resource locators)

- base, [221](#), [231](#)

- LINQ to TerraServer, [221](#)

- LINQ to Twitter, [221](#), [230](#)–231

- REST, [221](#)

- The Wayward WebLog, [221](#)

User Interface (UI) layer, [241](#), [247](#)–254

User objects, [217](#), [233](#)

/user option, [172](#)–173

User.cs file, [220](#)

UserName property, [228](#), [233](#)

using declaration, [192](#), [198](#)

using statement, [245](#), [267](#)

V

validation, [160](#)–162, [243](#)

var keyword, [24](#)

var variable, [24](#)

variables

- implicitly typed, [24](#)

- local, [24](#)

- local*, [36](#)

- range*, [36](#), [41](#), [59](#)

.vb extension, [177](#)

VB.NET language, [4](#), [177](#)

/views option, [174](#)

Visual Studio 2008 (VS 2008)

- ILDASM tool, [5](#)

- LINQ to SQL Designer, [70](#)–76

- snippets, [6](#)

- strongly typed DataSets, [115](#)–117

VS 2008 Express, [5](#)

W

warning messages, [173](#)

Warren, Matt, [221](#)

Wayward WebLog, The, [221](#)

WCF (Windows Communications Foundation), [178](#)

WCF Web Service, [132](#)–133

web service API, [219](#)

web sites

- CodePlex, [216](#)

- LINQ to TerraServer, [221](#)

- MSDN Code Gallery, [36](#)

- MSDN Parallel Computing Developer Center, [272](#)

- Twitter, [216](#)

- The Wayward WebLog, [221](#)

WebForms, [236](#)–237, [238](#), [247](#), [252](#)–254

WhenChanged value, [257](#)

where clause, [25](#), [40](#)–42, [95](#), [229](#)

Where extension methods, [259](#)

Where operator, [198](#)–199, [205](#), [344](#)–345

Where Or problem, [198](#)–199

whereExpression, [229](#)

whitespace, [18](#), [155](#)

Windows Communications Foundation (WCF), [178](#)

X

XAttribute class, [150](#)–152

XCDATA instance, [159](#)

XContainer class, [150](#), [151](#)

XDocument class, [157](#)–158

XElement class, [150](#)–155, [233](#)

XML

- creating from strings, [153](#)–154

- described, [150](#)

- functional construction, [151](#)–152

- reading from files, [154](#)–155

- reading from streams, [154](#)

- reading from URIs, [155](#)

- reading from XmlReaders, [155](#)

- sources of, [153](#)

XML documents. *See also* LINQ to XML

- adding elements, [167](#)

- CData text, [159](#)

- creating, [151](#)–153

- declarations, [158](#)–159

- external mapping files, [178](#)–183

- manipulating, [166](#)–169

- modifying elements, [167](#)–168

- overloads, [152](#), [153](#)

- processing instructions, [160](#)

- removing elements, [168](#)–169

- schema creation, [161](#)–162

- validating, [160](#)–162

- working with, [157](#)–158

.xml extension, [174](#)

XML option, [232](#)

XML strings, [151](#)–155

XmlMappingSource class, [182](#)–183

XmlReaders, [155](#)