The Addison-Wesley Microsoft Technology Series

LINQ to Objects Using C# 4.0

Using and Extending LINQ to Objects and Parallel LINQ (PLINQ)

Troy Magennis

Foreword by Barry Vandevier, *Chief Information Officer, Sabre Holdings*

LINQ to Objects Using C# 4.0

This page intentionally left blank

LINQ to Objects Using C# 4.0

Using and Extending LINQ to Objects and Parallel LINQ (PLINQ)

Troy Magennis

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Library of Congress Cataloging-in-Publication Data:

Magennis, Troy, 1970-
LINQ to objects using C# 4.0 : using and extending LINQ to objects and parallel LINQ (PLINQ) / Troy Magennis.
p. cm.
Includes bibliographical references and index.
ISBN 978-0-321-63700-0 (pbk. : alk. paper) 1. Microsoft LINQ. 2. Query languages (Computer science) 3. C#
(Computer program language) 4. Microsoft .NET Framework. I. Title.
QA76.73.L228M345 2010
006.7'882—dc22
2009049530

ISBN-13: 978-0-321-63700-0
ISBN-10: 0-321-63700-3

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

First printing March 2010

To my wife, Janet Doherty, for allowing me to spend those extra hours tapping away on the keyboard; thank you for your support and love.

This page intentionally left blank

Contents

Foreword . . . x
Preface . . . xii
Acknowledgments . . . xix
About the Author . . . xx

Chapter 1: Introducing LINQ . . . 1

What Is LINQ? . . . 1
The (Almost) Current LINQ Story . . . 3
LINQ Code Makeover—Before and After Code Examples . . . 5
Benefits of LINQ . . . 12
Summary . . . 15
References . . . 15

Chapter 2: Introducing LINQ to Objects . . . 17

LINQ Enabling C# 3.0 Language Enhancements . . . 17
LINQ to Objects Five-Minute Overview . . . 30
Summary . . . 39
References . . . 39

Chapter 3: Writing Basic Queries . . . 41

Query Syntax Style Options . . . 41
How to Filter the Results (Where Clause) . . . 49
How to Change the Return Type (Select Projection) . . . 54
How to Return Elements When the Result
Is a Sequence (Select Many) . . . 59
How to Get the Index Position of the Results . . . 61
How to Remove Duplicate Results . . . 62
How to Sort the Results . . . 63
Summary . . . 73

Chapter 4: Grouping and Joining Data 75

How to Group Elements 75
How to Join with Data in Another Sequence 93
Summary 119

Chapter 5: Standard Query Operators 121

The Built-In Operators 121
Aggregation Operators—Working with Numbers 123
Conversion Operators—Changing Types 131
Element Operators 144
Equality Operator—SequenceEqual 153
Generation Operators—Generating Sequences of Data 155
Merging Operators 159
Partitioning Operators—Skipping and Taking Elements 160
Quantifier Operators—All, Any, and Contains 164
Summary 171

Chapter 6: Working with Set Data 173

Introduction 173
The LINQ Set Operators 174
The HashSet<T> Class 185
Summary 192

Chapter 7: Extending LINQ to Objects 195

Writing a New Query Operator 195
Writing a Single Element Operator 196
Writing a Sequence Operator 208
Writing an Aggregate Operator 216
Writing a Grouping Operator 222
Summary 232

Chapter 8: C# 4.0 Features 233

Evolution of C# 233
Optional Parameters and Named Arguments 234
Dynamic Typing 243
COM-Interop and LINQ 251
Summary 260
References 260

Chapter 9: Parallel LINQ to Objects 261

Parallel Programming Drivers 261
Multi-Threading Versus Code Parallelism 264
Parallelism Expectations, Hindrances, and Blockers 267
LINQ Data Parallelism 271
Writing Parallel LINQ Operators 289
Summary 301
References 301

Glossary 303
Index 307

Foreword

I have worked in the software industry for more than 15 years, the last four years as CIO of Sabre Holdings and the prior four as CTO of Travelocity. At Sabre, on top of our large online presence through Travelocity, we transact $70 billion in annual gross travel sales through our network and serve over 200 airline customers worldwide. On a given day, we will process over 700 million transactions and handle 32,000 transactions per second at peak. Working with massive streams of data is what we do, and finding better ways to work with this data and improve throughput is my role as CIO.

Troy is our VP over Architecture at Travelocity, where I have the pleasure of watching his influence on a daily basis. His perspective on current and future problems and depth of detail are observed in his architectural decisions, and you will find this capability very evident in this book on the subject of LINQ and PLINQ.

Developer productivity is a critical aspect for every IT solution-based business, and Troy emphasizes this in every chapter of his book. Languages and language features are a means to an end, and language features like LINQ offer key advances in developer productivity. By simplifying all types of data manipulation by adding SQL-style querying within the core .NET development languages, developers can focus on solving business problems rather than learning a new query language for every data source type. Beyond developer productivity, the evolution in technology from individual processor speed improvements to multi-core processors opened up a big hole in run-time productivity as much of today's software lacks investment in parallelism required to better utilize these new processors. Microsoft's investment in Parallel LINQ addresses this hole, enabling much higher utilization of today's hardware platforms.

Open-standards and open-frameworks are essential in the software industry. I'm pleased to see that Microsoft has approached C# and LINQ in an open and inclusive way, by handing C# over as an ECMA/ISO standard, allowing everyone to develop new LINQ data-sources and to extend the LINQ query language operators to suit their needs. This approach showcases the traits of many successful open-source initiatives and demonstrates the competitive advantages openness offers.

Decreasing the ramp-up speed for developers to write and exploit the virtues of many-core processors is extremely important in today's world and will have a very big impact in technology companies that operate at the scale of Sabre. Exposing common concurrent patterns at a language level offers the best way to allow current applications to scale safely and efficiently as core-count increases. While it was always possible for a small percentage of developers to reliably code concurrency through OpenMP or hand-rolled multi-threading frameworks, parallel LINQ allows developers to take advantage of many-core scalability with far fewer concerns (thread synchronization, data segmentation, merging results, for example). This approach will allow companies to scale this capability across a much higher percentage of developers without losing focus on quality. So roll up your sleeves and enjoy the read!

—Barry Vandevier
Chief Information Officer, Sabre Holdings

Preface

LINQ to Objects Using C# 4.0 takes a different approach to the subject of Language Integrated Query (LINQ). This book focuses on the LINQ syntax and working with in-memory collections rather than focusing on replacing other database technologies. The beauty of LINQ is that once you master the syntax and concepts behind how to compose clever queries, the underlying data source is mostly irrelevant. That's not to say that technologies such as LINQ to SQL, LINQ to XML, and LINQ to Entities are un-important; they are just not covered in this book.

Much of the material for this book was written during late 2006 when Language Integrated Query (LINQ) was in its earliest preview period. I was lucky enough to have a window of time to learn a new technology when LINQ came along. It became clear that beyond the clever data access abilities being demonstrated (DLINQ at the time, LINQ to SQL eventually), LINQ to Objects would have the most impact on the day-to-day developers' life. Working with in-memory collections of data is one of the more common tasks performed, and looking through code in my previous projects made it clear just how complex my for-loops and nested if-condition statements had evolved. LINQ and the language enhancements being proposed were going to change the look and feel of the way we programmed, and from where I was sitting that was fantastic.

The initial exploration was published on the HookedOnLINQ.com Wiki (120 odd pages at that time), and the traffic grew over the next year or two to a healthy level. Material could have been pulled together for a publication at that time (and been first to market with a book on this subject, something my Addison-Wesley editor will probably never forgive me for), but I felt knowing the syntax and the raw operators wasn't a book worth reading. It was critical to know how LINQ works in the real world and how to use it on real projects before I put that material into ink. The first round of books for any new programming technology often go slightly deeper than the online-documentation, and I wanted to wait and see how the LINQ story unfolded in real-world applications and write the first book of the second-generation—the book that isn't just reference, but has integrity that only real-world application can ingrain.

The LINQ story is a lot deeper and has wider impact than most people realize at first glance of any TechEd session recording or user-group presentation. The ability to store and pass code as a data structure and to control when and how that code is executed builds a powerful platform for working with all matter of data sources. The few LINQ providers shipped by Microsoft are just the start, and many more are being built by the community through the extension points provided. After mastering the LINQ syntax and understanding the operators' use (and how to avoid misuse), any developer can work more effectively and write cleaner code. This is the purpose of this book: to assist the reader in beginning the journey, to introduce how to use LINQ for more real-world examples and to dive a little deeper than most books on the subject, to explore the performance benefits of one solution over another, and to deeply look at how to create custom operators for any specific purpose.

I hope you agree after reading this book that it does offer an insight into how to use LINQ to Objects on real projects and that the examples go a step further in explaining the patterns that make LINQ an integral part of day-to-day programming from this day forward.

Who Should Read This Book

The audience for this book is primarily developers who write their applications in C# and want to understand how to employ and extend the features of LINQ to Objects. LINQ to Objects is a wide set of technology pieces that work in tandem to make working with in-memory data sources easier and more powerful. This book covers both the initial C# 3.0 implementation of LINQ and the updates in C# 4.0. If you are accustomed to the LINQ syntax, this book goes deeper than most LINQ reference publication and delves into areas of performance and how to write custom LINQ operators (either as sequential algorithms or using parallel algorithms to improve performance).

If you are a beginning C# developer (or new to C# 3.0 or 4.0), this book introduces the code changes and syntax so that you can quickly master working with objects and collections of objects using LINQ. I've tried to strike a balance and not jump directly into examples before covering the basics. You obviously should know how to build a LINQ query statement before you start to write your own custom sequential or parallel operators to determine the number of mountain peaks around the world that are taller than 8,000 meters (26,000 feet approximately). But you will get to that in the latter chapters.

Overview of the Book

LINQ to Objects Using C# 4.0 starts by introducing the intention and benefits LINQ offers developers in general. Chapter 1, "Introducing LINQ," talks to the motivation and basic concepts LINQ introduces to the world of writing .NET applications. Specifically, this chapter introduces before and after code makeovers to demonstrate LINQ's ability to simplify coding problems. This is the first and only chapter that talks about LINQ to SQL and LINQ to XML and does this to demonstrate how multiple LINQ data sources can be used from the one query syntax and how this powerful concept will change application development. This chapter concludes by listing the wider benefits of embracing LINQ and attempts to build the big picture view of what LINQ actually is, a more complex task than it might first seem.

Chapter 2, "Introducing LINQ to Objects," begins exploring the underlying enabling language features that are necessary to understand how the LINQ language syntax compiles. A fast-paced, brief overview of LINQ's features wraps up this chapter; it doesn't cover any of them in depth but just touches on the syntax and capabilities that are covered at length in future chapters.

Chapter 3, "Writing Basic Queries," introduces reading and writing LINQ queries in C# and covers the basics of choosing what data to project, in what format to select that data, and in what order the final result should be placed. By the end of this chapter, each reader should be able to read the intention behind most queries and be able to write simple queries that filter, project, and order data from in-memory collections.

Chapter 4, "Grouping and Joining Data," covers the more advanced features of grouping data in a collection and combining multiple data sources. These partitioning and relational style queries can be structured and built in many ways, and this chapter describes in depth when and why to use one grouping or joining syntax over another.

Chapter 5, “Standard Query Operators,” lists the many additional standard operators that can be used in a LINQ query. LINQ has over 50 operators, and this chapter covers the operators that go beyond those covered in the previous chapters.

Chapter 6, “Working with Set Data,” explores working with set-based operators. There are multiple ways of performing set operations over in-memory collections, and this chapter explores the merits and pitfalls of both.

Chapter 7, “Extending LINQ to Objects,” discusses the art of building custom operators. The examples covered in this chapter demonstrate how to build any of the four main types of operators and includes the common coding and error-handling patterns to employ in order to closely match the built-in operators Microsoft supplies.

Chapter 8, “C# 4.0 Features,” is where the additional C# 4.0 language features are introduced with particular attention to how they extend the LINQ to Objects story. This chapter demonstrates how to use the dynamic language features to make LINQ queries more fluent to read and write and how to combine LINQ with COM-Interop in order to use other applications as data sources (for example, Microsoft Excel).

Chapter 9, “Parallel LINQ to Objects,” closely examines the motivation and art of building application code that can support multi-core processor machines. Not all queries will see a performance improvement, and this chapter discusses the expectations and likely improvement most queries will see. This chapter concludes with an example of writing a custom parallel operator to demonstrate the thinking process that goes into correctly coding parallel extensions in addition to those provided.

Conventions

There is significant code listed in this book. It is an unavoidable fact for books about programming language features that they must demonstrate those features with code samples. It was always my intention to show lots of examples, and every chapter has dozens of code listings. To help ease the burden, I followed some common typography conventions to make them more readable. References to classes, variables, and other code entities are distinguished in a `monospace` font. Short code listings that are to be read inline with the surrounding text are also presented in a `monospace` font, but on their own lines, and they sometimes contain code comments (lines beginning with `//` characters) for clarity.

```
// With line-breaks added for clarity
var result = nums
              .Where(n => n < 5)

              .OrderBy (n => n);
```

Longer listings for examples that are too big to be inline with the text or samples I specifically wanted to provide in the sample download project are shown using a similar `monospace` font, but they are denoted by a listing number and a short description, as in the following example, Listing 3-2.

Listing 3-2 Simple query using the Query Expression syntax

```
List<Contact> contacts = Contact.SampleData();

var q = from c in contacts
         where c.State == "WA"
         orderby c.LastName, c.FirstName
         select c;

foreach (Contact c in q)
    Console.WriteLine("{0} {1}",
         c.FirstName, c.LastName);
```

Each example should be simple and consistent. For simplicity, most examples write their results out to the Console window. To capture these results in this book, they are listed in the same font and format as code listings, but identified with an output number, as shown in Output 3-1.

Output 3-1

```
Stewart Kagel
Chance Lard
Armando Valdes
```

Sample data for the queries is listed in tables, for example, Table 2-2. Each column maps to an object property of a similar legal name for queries to operate on.

Words in **bold** in normal text are defined in the Glossary, and only the first occurrence of the word gets this treatment. When a **`bold monospace`** font in code is used, it is to draw your attention to a particular key point being explained at that time and is most often used when an example evolves over multiple iterations.

Sample Download Code and Updates

All of the samples listed in the book and further reference material can be found at the companion website, the HookedOnLINQ.com reference wiki and website at http://hookedonlinq.com/LINQBook.ashx.

Some examples required a large sample data source and the Geonames database of worldwide geographic place names and data. These data files can be downloaded from http://www.geonames.org/ and specifically the http://download.geonames.org/export/dump/allCountries.zip file. This file should be downloaded and placed in the same folder as the executable sample application is running from to successfully run those specific samples that parse and query this source.

Choice of Language

I chose to write the samples in this book using the C# language because including both C# and VB.Net example code would have bloated the number of pages beyond what would be acceptable. There is no specific reason why the examples couldn't have been in any other .NET language that supports LINQ.

System Requirements

This book was written with the code base of .NET 4 and Visual Studio 2010 over the course of various beta versions and several community technical previews. The code presented in this book runs with Beta 2. If the release copy of Visual Studio 2010 and .NET 4 changes between this book publication and release, errata and updated code examples will be posted on the companion website at http://hookedonlinq.com/LINQBook.ashx.

To run the samples available from the book's companion website, you will need to have Visual Studio 2010 installed on your machine. If you don't have access to a commercial copy of Visual Studio 2010, Microsoft has a freely downloadable version (Visual Studio 2010 Express Edition), which is capable of running all examples shown in this book. You can download this edition from http://www.microsoft.com/express/.

Acknowledgments

It takes a team to develop this type of book, and I want our team members to know how appreciated their time, ideas, and effort have been. This team effort is what sets blogging apart from publishing, and I fully acknowledge the team at Addison-Wesley, in particular my editors Joan Murray and Olivia Basegio for their patience and wisdom.

To my technical reviewers, Nick Paldino, Derik Whittaker, Steve Danielson, Peter Ritchie, and Tanzim Saqib—thank you for your insights and suggestions to improve accuracy and clarity. Each of you had major impact on the text and code examples contained in this book.

Some material throughout this book, at least in spirit, was obtained by reading the many blog postings from Microsoft staff and skilled individuals from our industry. In particular I'd like to thank the various contributors to the Parallel FX team blog (http://blogs.msdn.com/pfxteam/), notably Igor Ostrovsky (strongly influenced my approach to aggregations), Ed Essey (helped me understand the different partitioning schemes used in PLINQ), and Stephen Toub. Stephen Toub also has my sincere thanks for giving feedback on the Parallel LINQ chapter during its development (Chapter 9), which dramatically improved the content accuracy and depth.

I would also like to acknowledge founders and contributors to Geonames.org (http://geonames.org), whose massive set of geographic data is available for free download under creative commons attribution license. This data is used in Chapter 9 to test PLINQ performance on large data sets.

Editing isn't easy, and I'd like to acknowledge the patience and great work of Anne Goebel and Chrissy White in making my words flow from post-tech review to production. I know there are countless other staff who touched this book in its final stages of production, and although I don't know your names, thank you.

Finally, I'd like to acknowledge readers like you for investing your time to gain a deeper understanding of LINQ to Objects. I hope after reading it you agree that this book offers valuable insights on how to use LINQ to Objects in real projects and that the examples go that step further in explaining the patterns that make LINQ an integral part of day-to-day programming from this day forward. Thank you.

About the Author

Troy Magennis is a Microsoft Visual C# MVP, an award given to industry participants who dedicate time and effort to educating others about the virtues of technology choices and industry application.

A keen traveler, Troy currently works for Travelocity, which manages the travel and leisure websites travelocity.com, lastminute.com, and zuji. As vice president of Architecture, he leads a talented team of architects spread across four continents committed to being the traveler's companion.

Technology has always been a passion for Troy. After cutting his teeth on early 8-bit personal computers (Vic20s, Commodore 64s), he moved into electronics engineering, which later led to positions in software application development and architecture for some of the most prominent corporations in automotive, banking, and online commerce.

Troy's first exposure to LINQ was in 2006 when he took a sabbatical to learn it and became hooked, ultimately leading him to publish the popular HookedOnLINQ website.

CHAPTER 8

C# 4.0 Features

Goals of this chapter:

- Define new C# 4.0 language features.
- Demonstrate the new language features in the context of LINQ to Objects.

C# is an evolving language. This chapter looks at the new features added into C# 4.0 that combine to improve code readability and extend your ability to leverage LINQ to Object queries over dynamic data sources. The examples in this chapter show how to improve the coding model for developers around reading data from various sources, including text files and how to combine data from a **COM-Interop** source into a LINQ to Objects query.

Evolution of C#

C# is still a relatively new language (circa 2000) and is benefiting from continuing investment by Microsoft's languages team. The C# language is an ECMA and ISO standard. (ECMA is an acronym for European Computer Manufacturers Association, and although it changed its name to Ecma International in 1994, it kept the name Ecma for historical reasons.[1]) The standard ECMA-334 and ISO/IEC 23270:2006 is freely available online at the Ecma International website[2] and describes the language syntax and notation. However, Microsoft's additions to the language over several versions take some time to progress through the standards process, so Microsoft's release cycle leads Ecma's acceptance by at least a version.

Each version of C# has a number of new features and generally a major theme. The major themes have been generics and nullable types in C# 2.0, LINQ in C# 3.0, and dynamic types in C# 4.0. The major features added in each release are generally considered to be the following:

- **C# 2.0—**Generics (.NET Framework support was added, and C# benefited from this); iterator pattern (the `yield` keyword); anonymous methods (the `delegate` keyword), nullable types, and the null coalescing operator (`??`).
- **C# 3.0—**Anonymous types, extension methods, object initializers, collection initializers, implicitly typed local variables (`var` keyword), lambda expressions (`=>`), and the LINQ query expression pattern.
- **C# 4.0—**Optional Parameters and Named Arguments, Dynamic typing (`dynamic` type), improved COM-Interop, and Contra and Co-Variance.

The new features in C# 3.0 that launched language support for LINQ can be found in Chapter 2, "Introducing LINQ to Objects," and this chapter documents each of the major new features in C# 4.0 from the perspective of how they impact the LINQ story.

Optional Parameters and Named Arguments

A long-requested feature for C# was to allow for method parameters to be optional. Two closely related features in C# 4.0 fulfill this role and enable us to either omit arguments that have a defined default value when calling a method, and to pass arguments by name rather than position when calling a method.

OPTIONAL PARAMETERS OR OPTIONAL ARGUMENTS? Optional parameters and named parameters are sometimes called optional arguments and named arguments. These names are used interchangeably in this book, and in most literature, including the C# 4.0 specification that uses both, sometimes in the same section. I use "argument" when referring to a value passed in from a method call and "parameter" when referring to the method signature.

The main benefit of these features is to improve **COM-Interop** programming (which is covered shortly) and to reduce the number of method overloads created to support a wide range of parameter overloads. It is a common programming pattern to have a master method signature containing all parameters (with the actual implementation) chained to a number of overloaded methods that have a lesser parameter signature set calling the master method with hard-coded default values. This common coding pattern becomes unnecessary when optional parameters are used in the definition of the aforementioned master method signature, arguably improving code readability and debugging by reducing clutter. (See Listing 8-2 for an example of the old and new way to create multiple overloads.)

There has been fierce debate on these features on various email lists and blogs. Some C# users believe that these features are not necessary and introduce uncertainty in versioning. For example if version 2 of an assembly changes a default parameter value for a particular method, client code that was assuming a specific default might break. This is true, but the existing chained method call pattern suffers from a similar issue—default values are coded into a library or application somewhere, so thinking about when and how to handle these hard-coded defaults would be necessary using either the existing chained method pattern or the new optional parameters and named arguments. Given that optional parameters were left out of the original C# implementation (even when the .NET Runtime had support and VB.NET utilized this feature), we must speculate that although this feature is unnecessary for general programming, coding COM-Interop libraries without this feature is unpleasant and at times infuriating—hence, optional parameters and specifying arguments by name has now made its way into the language.

COM-Interop code has always suffered due to C#'s inability to handle optional parameters as a concept. Many Microsoft Office Component Object Model (**COM**) libraries, like those built to automate Excel or Word for instance, have method signatures that contain 25 optional parameters. Previously you had no choice but to pass dummy arguments until you reached the "one" you wanted and then fill in the remaining arguments until you had fulfilled all 25. Optional parameters and named arguments solve this madness, making coding against COM interfaces much easier and cleaner. The code shown in Listing 8-1 demonstrates the before and after syntax of a simple Excel COM-Interop call to open an Excel spreadsheet. It shows how much cleaner this type of code can be written when using C# 4.0 versus any of its predecessors.

Listing 8-1 Comparing the existing way to call COM-Interop and the new way using optional parameters

```
// Old way - before optional parameters
var excel = new Microsoft.Office.Interop.Excel.Application();
try
{
    Microsoft.Office.Interop.Excel.Workbook workBook =
           excel.Workbooks.Open(fileName, Type.Missing,
              Type.Missing, Type.Missing, Type.Missing,
              Type.Missing, Type.Missing, Type.Missing,
              Type.Missing, Type.Missing, Type.Missing,
              Type.Missing, Type.Missing, Type.Missing,
              Type.Missing);

    // do work with Excel...

    workBook.Close(false, fileName);
}
finally
{
    excel.Quit();
}

// New Way - Using optional parameters
var excel = new Microsoft.Office.Interop.Excel.Application();
try
{
    Microsoft.Office.Interop.Excel.Workbook workBook =
           excel.Workbooks.Open(fileName);

    // do work with Excel...

    workBook.Close(false, fileName);
}
finally
{
    excel.Quit();
}
```

The addition of object initializer functionality in C# 3.0 took over some of the workload of having numerous constructor overloads by allowing public properties to be set in line with a simpler constructor (avoiding having a constructor for every Select projection needed). Optional parameters and named arguments offer an alternative way to simplify coding a LINQ Select projection by allowing variations of a type's constructor with a lesser set of parameters. Before diving into how to use these features in LINQ queries, it is necessary to understand the syntax and limitations of these new features.

Optional Parameters

The first new feature allows default parameters to be specified in a method signature. Callers of methods defined with default values can omit those arguments without having to define a specific overload matching that lesser parameter list for convenience.

To define a default value in a method signature, you simply add a constant expression as the default value to use when omitted, similar to member initialization and constant definitions. A simple example method definition that has one mandatory parameter (`p1`, just like normal) and an optional parameter definition (`p2`) takes the following form:

```
public void MyMethod( int p1, int p2 = 5 );
```

The following invocations of method `MyMethod` are legal (will compile) and are functionally equivalent as far as the compiler is concerned:

```
MyMethod( 1, 5 );
MyMethod( 1 ); // the declared default for p2 (5) is used
```

The rules when defining a method signature that uses optional parameters are:

1. Required parameters cannot appear after any optional parameter.
2. The default specified must be a constant expression available at compile time or a value type constructor without parameters, or `default(T)` where `T` is a value type.
3. The constant expression must be implicitly convertible by an identity (or nullable conversion) to the type of the parameter.
4. Parameters with a `ref` or `out` modifier cannot be optional parameters.
5. Parameter arrays (`params`) can occur after optional parameters, but these cannot have a default value assigned. If the value is omitted by the calling invocation, an empty parameter array is used in either case, achieving the same results.

Valid optional parameter definitions take the following form:

```
public void M1(string s, int i = 1) { }
public void M2(Point p = new Point()) { }
public void M3(Point p = default(Point)) { }
public void M4(int i = 1, params string[] values) { }
```

The following method definitions using optional parameters will *not* compile:

```
//"Optional parameters must appear after all required parameters"
public void M1 (int i = 1, string s) {}

//"Default parameter value for 'p' must be a compile-time constant"
//(Can't use a constructor that has parameters)
public void M2(Point p = new Point(0,0)) {}

//"Default parameter value for 'p' must be a compile-time constant"
//(Must be a value type (struct or built-in value types only))
public void M5(StringBuilder p = new StringBuilder()) {}

//"A ref or out parameter cannot have a default value"
public void M6(int i = 1, out string s = "") {}

//"Cannot specify a default value for a parameter array"
public void M7(int i = 1, params string[] values = "test") {}
```

To understand how optional parameters reduce our code, Listing 8-2 shows a traditional overloaded method pattern and the equivalent optional parameter code.

Listing 8-2 Comparing the traditional cascaded method overload pattern to the new optional parameter syntax pattern

```
// Old way - before optional parameters
public class OldWay
{
    // multiple overloads call the one master
    // implementation of a method that handles all inputs
    public void DoSomething(string formatString)
    {
        // passing 0 as param1 default,
        // and true as param2 default.
        DoSomething(formatString, 0, true);
    }

    public void DoSomething(string formatString, int param1)
    {
        DoSomething(formatString, param1, true);
    }

    public void DoSomething(string formatString, bool param2)
    {
        DoSomething(formatString, 0, param2);
    }

    // the actual implementation. All variations call this
    // method to implement the methods function.
    public void DoSomething(
        string formatString,
        int param1,
        bool param2)
    {
        Console.WriteLine(
            String.Format(formatString, param1, param2));
    }
}

// New Way - Using optional parameters
public class NewWay
{
    // optional parameters have a default specified.
    // optional parameters must come after normal params.
    public void DoSomething(
        string formatString,
        int param1 = 0,
        bool param2 = true)
    {
        Console.WriteLine(
            String.Format(formatString, param1, param2));
    }
}
```

Named Arguments

Traditionally, the position of the arguments passed to a method call identified which parameter that value matched. It is possible in C# 4.0 to specify arguments by name, in addition to position. This is helpful when many parameters are optional and you need to target a specific parameter without having to specify all proceeding optional parameters.

Methods can be called with any combination of positionally specified and named arguments, as long as the following rules are observed:

1. If you are going to use a combination of positional and named arguments, the positional arguments must be passed first. (They cannot come after named arguments.)
2. All non-optional parameters must be specified somewhere in the invocation, either by name or position.
3. If an argument is specified by position, it cannot then be specified by name as well.

To understand the basic syntax, the following example creates a `System.Drawing.Point` by using named arguments. It should be noted that there is no constructor for this type that takes the y-size, x-size by position—this reversal is solely because of named arguments.

```
// reversing the order of arguments.
Point p1 = new Point(y: 100, x: 10);
```

The following method invocations will not compile:

```
//"Named argument 'x' specifies a parameter for which a
// positional argument has already been given"
Point p3 = new Point(10, x: 10);

// "Named argument specifications must appear after all
// fixed arguments have been specified"
Point p4 = new Point(y: 100, 10);

// "The best overload for '.ctor' does not have a
// parameter named 'x'"
Point p5 = new Point(x: 10);
```

To demonstrate how to mix and match optional parameters and named arguments within method or constructor invocation calls, the code shown in Listing 8-3 calls the method definition for `NewWay` in Listing 8-2.

Listing 8-3 Mixing and matching positional and named arguments in a method invocation for methods that have optional and mandatory parameters

```
NewWay newWay = new NewWay();

// skipping an optional parameter
newWay.DoSomething(
     "({0},{1}) New way - param1 skipped.",
     param2: false);

// any order, but if it doesn't have a default
// it must be specified by name somewhere!
newWay.DoSomething(
     param2: false,
     formatString: "({0},{1}) New way - params specified" +
                   " by name, in any order.",
     param1: 5);
```

Using Named Arguments and Optional Parameters in LINQ Queries

Named arguments and optional parameters offer an alternative way to reduce code in LINQ queries, especially regarding flexibility in what parameters can be omitted in an object constructor.

Although anonymous types make it convenient to project the results of a query into an object with a subset of defined properties, these anonymous types are scoped to the local method. To share a type across methods, types, or assemblies, a concrete type is needed, meaning the accumulation of simple types or constructor methods just to hold variations of data shape projections. Object initializers reduce this need by allowing a concrete type to have a constructor without parameters and public properties used to assign values in the Select projection. Object-oriented purists take issue with a parameterless constructor being a requirement; it can lead to invalid objects being created by users who are unaware that certain properties must be set before an object is correctly initialized for use—an opinion I strongly agree with. (You can't compile using the object initialization syntax unless the type concerned has a parameterless constructor, even if there are other constructors defined that take arguments.)

Optional parameters and named arguments can fill this gap. Data can be projected from queries into concrete types, and the author of that concrete type can ensure that the constructor maintains integrity by defining the default values to use when an argument is omitted. Many online discussions have taken place discussing if this is a good pattern; one camp thinks it doesn't hurt code readability or maintainability to use optional parameters in a constructor definition, and the other says refactoring makes it an easy developer task to define the various constructors required in a given type, and hence of no value. I see both sides of that argument and will leave it up to you to decide where it should be employed.

To demonstrate how to use named arguments and optional parameters from a LINQ query, the example shown in Listing 8-4 creates a subset of contact records (in this case, contacts from California) but omits the email and phone details. The Console output from this example is shown in Output 8-1.

Listing 8-4 Example LINQ query showing how to use named arguments and optional parameters to assist in projecting a lighter version of a larger type—see Output 8-1

```
var q = from c in Contact.SampleData()
        where c.State == "CA"
        select new Contact(
            c.FirstName, c.LastName,
            state: c.State,
            dateOfBirth: c.DateOfBirth
            );

foreach (var c in q)
    Console.WriteLine("{0}, {1} ({2}) - {3}",
        c.LastName, c.FirstName,
        c.DateOfBirth.ToShortDateString(), c.State);

public class Contact
{
    // constructor defined with optional parameters
    public Contact(
        string firstName,
        string lastName,
        DateTime dateOfBirth,
        string email = "unknown",  // optional
        string phone = "",         // optional
        string state = "Other")    // optional
    {
        FirstName = firstName;
        LastName = lastName;
        DateOfBirth = dateOfBirth;
        Email = email;
        Phone = phone;
        State = state;
    }

    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Email { get; set; }
    public string Phone { get; set; }
    public DateTime DateOfBirth { get; set; }
    public string State { get; set; }

    public static List<Contact> SampleData() ...
    // sample data the same as used in Table 2-1.
}
```

Output 8-1

```
Gottshall, Barney (10/19/1945) - CA
Deane, Jeffery (12/16/1950) - CA
```

Dynamic Typing

The wow feature of C# 4.0 is the addition of dynamic typing. Dynamic languages such as Python and Ruby have major followings and have formed a reputation of being super-productive languages for building certain types of applications.

The main difference between these languages and C# or VB.NET is the type system and specifically when (and how) member names and method names are resolved. C# and VB.NET require (or required, as you will see) that static types be available during compile time and will fail if a member name or method name does not exist. This static typing allows for very rigorous error checking during compile time, and generally improves code performance because the compiler can make targeted optimizations based on exact member name and method name resolution. Dynamic-typed languages on the other hand enable the member and method lookups to be carried out at runtime, rather than compile time. Why is this good? The main identifiable reason is that this allows code to locate members and methods dynamically at runtime and handle additions and enhancements without requiring a recompile of one system or another.

I'm going to stay out of the religious debate as to which is better. I believe there are positives and negatives in both approaches, and C# 4.0 allows you to make the choice depending on the coding problem you need to solve. Dynamic typing allows very clean coding patterns to be realized, as you will see in an upcoming example, where we code against the column names in a CSV file without the need for generating a backing class for every different CSV file format that might need to be read.

Using Dynamic Types

When a variable is defined with the type `dynamic`, the compiler ignores the call as far as traditional error checking is concerned and instead stores away the specifics of the action for the executing runtime to process at a later time (at execution time). Essentially, you can write whatever method calls (with whatever parameters), indexers, and properties you want on a dynamic object, and the compiler won't complain. These actions are picked up at runtime and executed according to how the dynamic type's binder determines is appropriate.

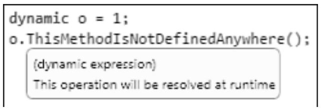
A binder is the code that gets the payload for an action on a dynamic instance type at runtime and resolves it into some action. Within the C# language, there are two paths code can take at this point, depending on the binder being used (the binding is determined from the actual type of the dynamic instance):

- The dynamic type does *not* implement the `IDynamicMetaObjectProvider` interface. In this case, the runtime uses reflection and its traditional method lookup and overload resolution logic before immediately executing the actions.
- The dynamic type implements the `IDynamicMetaObjectProvider` interface, by either implementing this interface by hand or by inheriting

the new dynamic type from the `System.Dynamic.DynamicObject` type supplied to make this easier.

Any traditional type of object can be declared as type `dynamic`. For all dynamic objects that don't implement the interface `IDynamicMetaObjectProvider`, the `Microsoft.CSharp.RuntimeBinder` is used, and reflection is employed to look up property and method invocations at runtime. The example code shown in Figure 8-1 shows the Intellisense balloon in Visual Studio 2010, which demonstrates an integer type declared as `dynamic`. (The runtime resolves the type by the initialization expression, just like using the local type inference `var` keyword.) No compile error occurs at design time or compile time, even though the method call is not defined anywhere in the project. When this code is executed, the runtime uses reflection in an attempt to find and execute the fictitious method `ThisMethodIsNotDefinedAnywhere`, which of course fails with the exception:

```
Microsoft.CSharp.RuntimeBinder.RuntimeBinderException: 'int' does
not contain a definition for 'ThisMethodIsNotDefinedAnywhere'
```



```
dynamic o = 1;
o.ThisMethodIsNotDefinedAnywhere();
```

(dynamic expression)
This operation will be resolved at runtime

Figure 8-1 Any object declared as type dynamic is resolved at runtime. No errors will be reported at compile time.

If that method had been actually declared, it would have been simply invoked just like any traditional method or property call.

The ability to have a type that doesn't implement the `IDynamicObject` interface should be rare. The `dynamic` keyword shouldn't be used in place of a proper type definition when that type is known at compile time. It also shouldn't be used in place of the `var` keyword when working with anonymous types, as that type is known at compile time. The `dynamic` keyword should only be used to declare `IDynamicMetaObjectProvider` implementers and for interoperating with other dynamic languages and for COM-Interop.

WHEN TO USE VAR AND DYNAMIC TYPE DECLARATIONS It might seem confusing as to which type definition should be used based on circumstance. Here are my recommendations:

Concrete type—Use whenever you know the type at coding time. If you know the type when defining a field, property, or return type—use it!

The var keyword—Use this when declaring an anonymous type or when capturing the result of a LINQ query projecting to an anonymous type.

The dynamic keyword—Use only when declaring a dynamic type, generally meaning a type that implements the `IDynamicMetaObjectProvider` interface. These can be custom types or one of the run-time binders provided by Microsoft for COM-Interop that interoperate with dynamic languages like IronPython and IronRuby. Declaring types that do not implement this interface incurs the overhead of reflection at runtime.

Specific binders are written to support specific purposes. IronRuby, IronPython, and COM-Interop are just a few of the bindings available to support dynamic language behavior from within C# 4.0. However, you can write your own and consume these types in order to solve some common coding problems, as you will see shortly in an example in which text file data is exposed using a custom dynamic type and this data is used as the source of a LINQ to Objects query.

Using Dynamic Types in LINQ Queries

Initially you might be disappointed to learn that dynamic types aren't supported in LINQ. LINQ relies exclusively on extension methods to carry out each query expression operation. Extension methods cannot be resolved at runtime due to the lack of information in the compiled assembly. Extension methods are introduced into scope by adding the assembly containing the extension into scope via a `using` clause, which is available at compile time for method resolutions, but not available at runtime—hence no LINQ support. However, this only means you can't define collection types as dynamic, but you can use dynamic types at the instance level (the types in the collections being queried), as you will see in the following example.

For this example we create a type that allows comma delimited text files to be read and queried in an elegant way, often useful when importing data from another application. By "elegant" I mean not hard-coding any column name definitions into string literals in our importing code, but rather, allowing direct access to fields just like they are traditional property accessors. This type of interface is often called a *fluent interface*. Given the sample CSV file content shown in Listing 8-5, the intention is to allow coders to directly reference the data columns in each row by their relevant header names, defined in the first row—that is FirstName, LastName, and State.

Listing 8-5 Comma separated value (CSV) file content used as example content

```
FirstName,LastName,State
Troy,Magennis,TX
Janet,Doherty,WA
```

The first row contains the column names for each row of the file, and this particular implementation expects this to always be the case. When writing LINQ queries against files of this format, referring to each row value in a column by the header name makes for easily comprehensible queries. The goal is to write the code shown in Listing 8-6, and this code compiling without a specific backing class from every CSV file type to be processed. (Think of it like coding against a dynamic anonymous type for the given input file header definition.)

Listing 8-6 Query code fluently reading CSV file content without a specific backing class

```
var q = from dynamic line in new CsvParser(content)
        where line.State == "WA"
        select line.LastName;
```

Dynamic typing enables us to do just that and with remarkably little code. The tradeoff is that any property name access isn't tested for type safety or existence during compile time. (The first time you will see an error is at runtime.) To fulfill the requirement of not wanting a backing class for each specific file, the `line` type shown previously must be of type dynamic. This is necessary to avoid the compile-time error that would be otherwise reported when accessing the State and LastName properties, which don't exist.

To create our new dynamic type, we need our type to implement `IDynamicMetaObjectProvider`, and Microsoft has supplied a starting point in the `System.Dynamic.DynamicObject` type. This type has virtual implementations of the required methods that allow a dynamic type to be built and allows the implementer to just override the specific methods needed for a given purpose. In this case, we need to override the `TryGetMember` method, which will be called whenever code tries to read a property member on an instance of this type. We will process each of these calls by returning the correct text out of the CSV file content for this line, based on the index position of the passed-in property name and the header position we read in as the first line of the file.

Listing 8-7 shows the basic code for this dynamic type. The essential aspects to support dynamic lookup of individual CSV fields within a line as simple property access calls are shown in this code. The property name is passed to the `TryGetMember` method in the `binder` argument, and can be retrieved by `binder.Name`, and the correct value looked up accordingly.

Listing 8-7 Class to represent a dynamic type that will allow the LINQ code (or any other code) to parse a single comma-separated line and access data at runtime based on the names in the header row of the text file

```
public class CsvLine : System.Dynamic.DynamicObject
{
    string[] _lineContent;
    List<string> _headers;

    public CsvLine(string line, List<string> headers)
    {
        this._lineContent = line.Split(',');
        this._headers = headers;
    }

    public override bool TryGetMember(
        GetMemberBinder binder,
        out object result )
    {
        result = null;

       // find the index position and get the value
        int index = _headers.IndexOf(binder.Name);
        if (index >= 0 && index < _lineContent.Length)
        {
            result = _lineContent[index];
            return true;
        }

        return false;
    }
}
```

To put in the plumbing required for parsing the first row, a second type is needed to manage this process, which is shown in Listing 8-8, and is called `CsvParser`. This is in place to determine the column headers to be used for access in each line after that and also the `IEnumerable` implementation that will furnish each line to any query (except the header line that contains the column names).

The constructor of the `CsvParser` type takes the CSV file content as a string and parses it into a string array of individual lines. The first row (as is assumed in this implementation) contains the column header names, and this is parsed into a `List<string>` so that the index positions of these column names can be subsequently used in the `CsvLine` type to find the correct column index position of that value in the data line being read. The `GetEnumerator` method simply skips the first line and then constructs a dynamic type `CsvLine` for each line after that until all lines have been enumerated.

Listing 8-8 The IEnumerable class that reads the header line and returns each line in the content as an instance of our CsvLine dynamic type

```
public class CsvParser : IEnumerable
{
    List<string> _headers;
    string[] _lines;

    public CsvParser(string csvContent)
    {
        _lines = csvContent.Split('\n');

        // grab the header row and remember positions
        if (_lines.Length > 0)
            _headers = _lines[0].Split(',').ToList();
    }

    public IEnumerator GetEnumerator()
    {
        // skip the header line
        bool header = true;

        foreach (var line in _lines)
            if (header)
                header = false;
            else
                yield return new CsvLine(line, _headers);
    }
}
```

Listing 8-9 shows the LINQ query that reads data from a CSV file and filters based on one of the column values. The important aspects of this example are the dynamic keyword in the `from` clause, and the ability to directly access the properties `State`, `FirstName`, and `LastName` from an instance of our `CsvLine` dynamic type. Even though there is no explicit backing type for those properties, they are mapped from the header row in the CSV file itself. This code will only compile in C# 4.0, and its output is all of the rows (in this case just one) that have a value of "WA" in the third column position (State), as shown in Output 8-2.

Listing 8-9 Sample LINQ query code that demonstrates how to use dynamic types in order to improve code readability and to avoid the need for strict backing classes—see Output 8-2

```
string content =
    "FirstName,LastName,State\n
    Troy,Magennis,TX\n
    Janet,Doherty,WA";

var q = from dynamic c in new CsvParser(content)
        where c.State == "WA"
        select c;

foreach (var c in q)
{
    Console.WriteLine("{0}, {1} ({2})",
        c.LastName,
        c.FirstName,
        c.State);
}
```

Output 8-2

```
Doherty, Janet (WA)
```

As this example has shown, it is possible to mix dynamic types with LINQ. The key point to remember is that the actual element types can be dynamic, but not the collection being queried. In this case, we built a simple enumerator that reads the CSV file and returns an instance of our dynamic type. Any CSV file, as long as the first row contains legal column names (no spaces or special characters that C# can't resolve as a property name), can be coded against just as if a backing class containing those columns names was created by code.

COM-Interop and LINQ

COM interoperability has always been possible within C# and .NET; however, it was often less than optimal in how clean and easy the code was to write (or read). Earlier in this chapter, named arguments and optional parameters were introduced, which improve coding against COM objects. And with the additional syntax improvements offered by the dynamic type, the code readability and conciseness is further improved.

To demonstrate how **COM-Interop** and LINQ might be combined, the following example shows a desirable coding pattern to read the contents of a Microsoft Excel spreadsheet, and use that data as a source for a LINQ to Objects query.

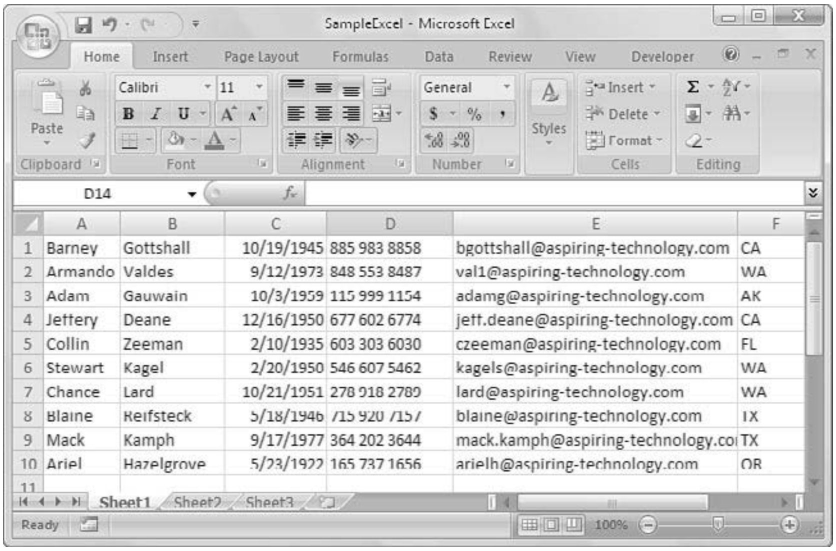
```
const int stateCol = 5;

var q = from row in GetExcelRowEnumerator(filename, 1)
        where row[stateCol] == "WA"
        select row;
```

This query returns rows where the State column equals "WA" from the spreadsheet data shown in Figure 8-2. Microsoft Excel is often used as the source for importing raw data, and although there are many ways to import this data using code, the ability to run LINQ queries over these spreadsheets directly from C# is useful.

The strategy used to implement the Microsoft Excel interoperability and allow LINQ queries over Excel data is:

1. Add a COM-Interop reference to the Microsoft Excel library in order to code against its object model in C#.
2. Return the data from a chosen spreadsheet (by its filename) into an `IEnumerable` collection (row by row) that can be used as a source for a LINQ query.



	A	B	C	D	E	F
1	Barney	Gottshall	10/19/1945	885 983 8858	bgottshall@aspiring-technology.com	CA
2	Armando	Valdes	9/12/1973	848 553 8487	val1@aspiring-technology.com	WA
3	Adam	Gauwain	10/3/1959	115 999 1154	adamg@aspiring-technology.com	AK
4	Jeffery	Deane	12/16/1950	677 602 6774	jeff.deane@aspiring-technology.com	CA
5	Collin	Zeeman	2/10/1935	603 303 6030	czeeman@aspiring-technology.com	FL
6	Stewart	Kagel	2/20/1950	546 607 5462	kagels@aspiring-technology.com	WA
7	Chance	Lard	10/21/1951	278 918 2789	lard@aspiring-technology.com	WA
8	Blaine	Reifsteck	5/18/1946	715 920 7157	blaine@aspiring-technology.com	TX
9	Mack	Kamph	9/17/1977	364 202 3644	mack.kamph@aspiring-technology.co	TX
10	Ariel	Hazelgrove	5/23/1922	165 737 1656	arielh@aspiring-technology.com	OR
11						

FIGURE 8-2 Sample Microsoft Excel spreadsheet to query using LINQ.

Adding a COM-Interop Reference

COM-Interop programming in C# 4.0 is greatly improved in one way because of a new style of interop backing class that is created when you add a COM reference to a project. The improved backing class makes use of optional parameters and named arguments and the dynamic type features that were introduced earlier this chapter. Listing 8-10 demonstrates the old code required to access an Excel spreadsheet via COM, and contrasts it with the new programming style.

Listing 8-10 Comparing the existing way to call COM-Interop and the new way using the improved COM reference libraries

```
// Old way - old COM reference and no optional parameters
var excel = new Microsoft.Office.Interop.Excel.Application();

Microsoft.Office.Interop.Excel.Workbook workBook =
     excel.Workbooks.Open(
     fileName,
     Type.Missing, Type.Missing, Type.Missing, Type.Missing,
     Type.Missing, Type.Missing, Type.Missing, Type.Missing,
     Type.Missing, Type.Missing, Type.Missing, Type.Missing,
     Type.Missing, Type.Missing);

// New Way - new COM reference and using optional parameters
var excel = new Microsoft.Office.Interop.Excel.Application();

Microsoft.Office.Interop.Excel.Workbook workBook =
     excel.Workbooks.Open(fileName);
```

When a reference is added to a COM component using Visual Studio, a backing class is generated to allow coding against that model. Listing 8-11 shows one method of the Excel COM programming model generated by Visual Studio 2008. With no optional parameters, all arguments must be passed when calling these methods.

Listing 8-11 Old style COM-Interop backing class added by Visual Studio 2008 for part of the Microsoft Excel 12 Object Library

```
Microsoft.Office.Interop.Excel.Workbook Open(
     string Filename,
     object UpdateLinks,
     object ReadOnly,
     object Format,
     object Password,
     object WriteResPassword,
     object IgnoreReadOnlyRecommended,
     object Origin,
     object Delimiter,
     object Editable,
     object Notify,
     object Converter,
     object AddToMru,
     object Local,
     object CorruptLoad)
```

Listing 8-12 shows the Visual Studio 2010 COM-Interop backing class. The addition of the default values turned all but one of the parameters (the `Filename` argument) into optional parameters; therefore, all but the filename can be omitted when this method is called, allowing the simplification shown in Listing 8-10.

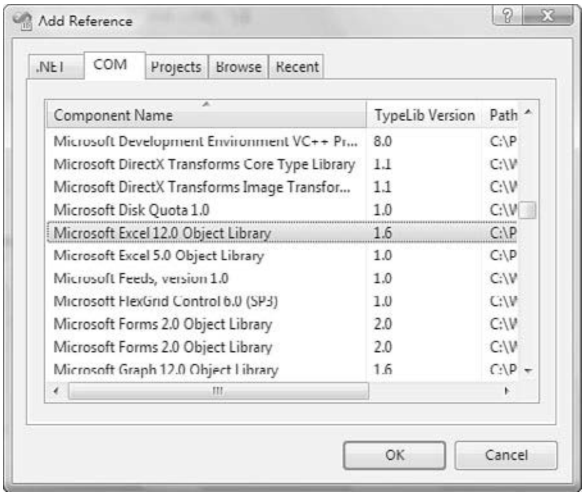
Listing 8-12 New style COM-Interop backing class added by Visual Studio 2010—notice the optional parameters and the dynamic types

```
Microsoft.Office.Interop.Excel.Workbook Open(
    string Filename,
    dynamic UpdateLinks = null,
    dynamic ReadOnly = null,
    dynamic Format = null,
    dynamic Password = null,
    dynamic WriteResPassword = null,
    dynamic IgnoreReadOnlyRecommended = null,
    dynamic Origin = null,
    dynamic Delimiter = null,
    dynamic Editable = null,
    dynamic Notify = null,
    dynamic Converter = null,
    dynamic AddToMru = null,
    dynamic Local = null,
    dynamic CorruptLoad = null)
```

Adding a COM reference is painless in Visual Studio. The step-by-step process is:

1. Open the Visual Studio C# project that the COM reference is being added to.

2. Choose `Project-Add Reference...` from the main menu (or right-click the References icon in the Solution Explorer and click `Add Reference...`).
3. Click the COM tab and find the COM Object you want to reference from within your project, as seen in Figure 8-3.
4. Click the OK button for the Add Reference dialog box.



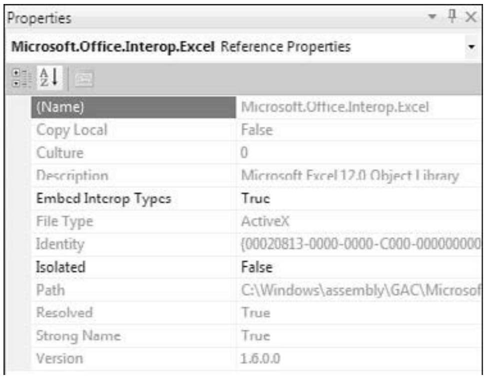
Component Name	TypeLib Version	Path
Microsoft Development Environment VC++ Pr...	8.0	C:\P
Microsoft DirectX Transforms Core Type Library	1.1	C:\W
Microsoft DirectX Transforms Image Transfor...	1.1	C:\W
Microsoft Disk Quota 1.0	1.0	C:\W
Microsoft Excel 12.0 Object Library	1.6	C:\P
Microsoft Excel 5.0 Object Library	1.0	C:\P
Microsoft Feeds, version 1.0	1.0	C:\W
Microsoft FlexGrid Control 6.0 (SP3)	1.0	C:\W
Microsoft Forms 2.0 Object Library	2.0	C:\W
Microsoft Forms 2.0 Object Library	2.0	C:\W
Microsoft Graph 12.0 Object Library	1.6	C:\P

FIGURE 8-3 The Add Reference dialog box in Visual Studio 2010.

NEW FEATURE: NOT DEPLOYING PRIMARY INTEROP ASSEMBLIES Primary Interop Assemblies are large pre-built .NET assemblies built for certain COM interfaces (like MS Excel and MS Word) to allow strongly typed coding at design time. These assemblies often were larger than the application being built (and loaded separately at runtime) and often caused version issues because they were deployed independently to the compiled application.

C# 4.0 introduces a no-PIA feature that compiles only the parts of the Primary Interop Assembly actually used into the assembly (much smaller) and avoids having to load a separate assembly at runtime (much faster).

This is the default behavior in Visual Studio 2010. To return to deploying the full PIA assembly (the default in previous versions of Visual Studio), set the Embed Interop Types property on the reference in question as shown in Figure 8-4.



Properties

Microsoft.Office.Interop.Excel Reference Properties

(Name)	Microsoft.Office.Interop.Excel
Copy Local	False
Culture	0
Description	Microsoft Excel 12.0 Object Library
Embed Interop Types	True
File Type	ActiveX
Identity	{00020813-0000-0000-C000-000000000
Isolated	False
Path	C:\Windows\assembly\GAC\Microsof
Resolved	True
Strong Name	True
Version	1.6.0.0

FIGURE 8-4 Set the Embed Interop Types to control whether the no-PIA feature is used (the default behavior, true) or the previous Visual Studio behavior is used (false).

Building the Microsoft Excel Row Iterator

To expose Microsoft Excel in a way that supports LINQ queries, an iterator must be built that internally reads data from an Excel spreadsheet and exposes this data as an `IEnumerable` collection, row by row. The skeleton of the Excel row iterator, without implementation is:

```
public IEnumerable<List<dynamic>> GetExcelRowEnumerator(
    string fileName,
    int sheetIndex)
{
    // Declare an array to hold our values
    // Create the COM reference to Excel
    // Open the workbook by filename
    // Get the excel worksheet, 1 for the first, etc.
    // Find the used range in the sheet
    // Read in the value array for all used rows and columns
    // Close Excel
    // Build and yield each row, one at a time
}
```

This iterator declaration takes arguments of a fully qualified filename to an Excel spreadsheet and a worksheet index as a one-based number and returns the values in each row (with each column's value as an item in a

`List<dynamic>` collection) from the chosen worksheet in the selected Excel file. The full implementation of this algorithm is shown in Listing 8-13.

This implementation isn't the strategy to be used for extremely large spreadsheets because it buffers the entire dataset into an in-memory array with a single call and then builds the row results from this array of values. This technique, however, is the fastest way to access data from Excel using COM-Interop because it avoids single cell or row access and keeps Excel open (a large executable memory footprint) for as short a time as possible. If an application is required to read a massive spreadsheet of data, experiment with alternative value access strategies supported by Excel's extensive object model, row by row perhaps, to avoid completely loading the entire array into memory upfront. This implementation is fine in performance and memory usage for most purposes.

THE USING DECLARATION FOR THE FOLLOWING EXAMPLES

To avoid having to prefix all calls to the Interop library with long namespaces, I added the following `using` declaration at the top of my class file:

```
using Excel = Microsoft.Office.Interop.Excel;
```

This simplified the code declarations and allowed me to use Excel.Application, Excel.Workbook (and others) rather than Microsoft.Office.Interop.Excel.Application, Microsoft.Office.Interop.Excel.Workbook, and so on.

Listing 8-13 Full code listing for an Excel row enumerator. Calling this method enumerates the values of a row in an Excel spreadsheet.

```
public IEnumerable<List<dynamic>> GetExcelRowEnumerator(
     string fileName,
     int sheetIndex)
{
     // declare an array to hold our values
     object[,] valueArray = null;

     // create the COM reference to Excel
     var excel = new Excel.Application();
     try
     {
        // open the workbook by filename
        Excel.Workbook workBook =
                excel.Workbooks.Open(fileName);

        if ( workBook != null &&
             sheetIndex < workBook.Sheets.Count )
        {
            // get the worksheet, 1 for the first, etc.
           Excel.Worksheet sheet =
                workBook.Sheets[sheetIndex];

            // find the used range in the sheet
            Excel.Range usedRange = sheet.UsedRange;

            // read in the value array, this is the fastest
            // way to get all values in one hit.
            valueArray = usedRange.get_Value(
                Excel.XlRangeValueDataType.xlRangeValueDefault);
        }

        workBook.Close(false, fileName);
    }
    finally
    {
        // finished with Excel now, close.
        excel.Quit();
    }

    // build and yield each row at a time
    for ( int rowIndex = 1;
           rowIndex <= valueArray.GetLength(0);
           rowIndex++)
    {
        List<dynamic> row =
            new List<dynamic>(
                valueArray.GetLength(1));

        // build a list of column values for the row
        for (int colIndex = 1;
              colIndex <= valueArray.GetLength(1);
              colIndex++)
        {

            row.Add(
                valueArray[rowIndex, colIndex]);
        }

        yield return row;
    }
}
```

Writing LINQ queries against Microsoft Excel data, like that shown in Figure 8-2, can be written in the code form shown in Listing 8-14. Output 8-3 shows the three rows returned from this code, which is all rows that have a State value of 'WA.'

Listing 8-14 Example code for reading the data from an Excel spreadsheet and running a LINQ query over its contents—see Output 8-3

```
string filename = Path.Combine(
    Environment.CurrentDirectory, "Data/SampleExcel.xlsx");


const int firstNameCol = 0;
const int lastNameCol = 1;
const int stateCol = 5;

var q = from row in GetExcelRowEnumerator(filename, 1)
        where row[stateCol] == "WA"
        select row;

Console.WriteLine("Customers in WA ({0})", q.Count());

foreach (var row in q)
{
    Console.WriteLine("{0}, {1}",
        row[lastNameCol].ToUpper(), row[firstNameCol] );
}
```

Output 8-3

```
Customers in WA (3)
VALDES, Armando
KAGEL, Stewart
LARD, Chance
```

The return type of the row values is declared as type `List<dynamic>`. It easily could have been declared as type `List<object>`. The downside of declaring the values as type of `object` rather than `dynamic` comes down to the ability to treat the value members as the underlying type. For example, in Listing 8-14 the statement `row[lastNameCol].ToUpper()` would fail if the element types were declared as `object`. The `object` type doesn't have a method called `ToUpper`, even though the underlying type it is representing is a string. And to access that method a type cast needs to be added, bloating the code out to `((string)row[lastNameCol]).ToUpper()`. Declaring the element type as dynamic in the collection allows the runtime to look up method names using reflection on the underlying type at runtime, however the particular type of that column value is declared as in Excel (in this case a string, but some columns are `DateTime` and `double`). The removal of the type casting when calling methods or properties on object types simplifies and improves code readability, at the expense of performance.

The `GetExcelRowEnumerator` method could be enhanced by combining the header row reading and accessibility in a similar fashion to that used by dynamic lookup in Listing 8-7, which would eliminate the need to hardcode the column index positions and allow the row data to be accessed by column header name using simple property access syntax.

Summary

This chapter introduced the new language features of C# 4.0 and demonstrated how they can be combined to extend the LINQ to Objects story. The examples provided showed how to improve the coding model around reading data from CSV text file sources and how to combine data from Microsoft Excel using COM-Interop into LINQ to Object queries.

References

1. Ecma International History from the Ecma International website hosted at http://www.ecma-international.org/memento/history.htm.

2. Ecma-334—C# Language Specification from the Ecma International website at http://www.ecma-international.org/publications/standards/Ecma-334.htm.

This page intentionally left blank

Index

A

adding COM-Interop interfaces, 253-256
advantages of LINQ, 13-15
aggregation operators, 123-125
  Aggregate, 123-125
  Average, 126-129
  Count, 129-131
  LongCount, 129-131
  LongSum, building, 219-222
  Max, 126-129
  Min, 126-129, 216-219
  Sum, 126-129
  writing, 216-222
Amdahl's law, 268
All operator, 164-166
anonymous types, 24-26
  returning, 58-59
Any operator, 166-169
arguments for extension methods, 18
AsEnumerable operator, 133
AsSequential operator, 285-287
Average operator, 126-129

B

benefits of LINQ, 13-15
bindings, 244
Box, Don, 2
building
  custom EqualityComparers, 184-185
  LongSum Operator, 219-222
  row iterator in Microsoft Excel, 256-260
  Segment operator, 226-232
  Soundex equality operator, 84-87
  TakeRange operator, 210-216
built-in performance optimization (LINQ to Objects), 200
built-in string comparers, 183-185

C

C# 2.0
  contract records, grouping and sorting versus LINQ approach, 5, 7
  data, summarizing from two collections versus LINQ approach, 8-12
  evolution of, 233-234
Callahan, David, 262
Cartesian Product, 94
case in-sensitive string ordering, 65-67
Cast operator, 133-134
choosing query syntax, 42
chunk partitioning, 277
classes, Hashset, 185-186, 191-192
code parallelism
  Amdahl's law, 268
  exceptions, 270
  overhead, 269
  synchronization, 269
  versus multi-threading, 264-267
Collection Initializers, 22
COM-Interop programming, 234
  combining with LINQ, 251-260
  references, adding, 253-256
  versus optional parameters, 235-237
combining LINQ and COM-Interop, 251-260
comparing
  LINQ Set operators and HashTable type methods, 186-192
  query syntax options, 45-49
composite keys
  grouping by, 80-83
  joining elements, 102
Concat operator, 174-176
Contains operator, 169-171
contract records, comparing LINQ and C# 2.0 grouping and sorting approaches, 5-7
conversion operators
  AsEnumerable, 133
  Cast, 133-134
  OfType, 134-136
  ToArray, 136
  ToDictionary, 136-139
  ToList, 140
  ToLookup, 140-143
cores, 264
Count operator, 129-131
CPUs
  cores, 264
  multi-threading versus code parallelism, 264-267
  overhead, 269
  processor speed, 263-264
  synchronization, 269
cross joins, 94-97
cultural-specific string ordering, 65-67

custom comparers, 83-87
custom EqualityComparers
building, 184-185
built-in string comparers, 183-185
customizing query result sort comparison functions, 67-72

D

data ordering, 270
declaring
anonymous types, 24-26
extension methods, 18-21
DefaultIfEmpty operator, 144-145
deferred execution, 51
delegate keyword, 26
Distinct operator, 177-178
dot notation syntax. *See* extension method format, 41
drivers for parallel programming, 261-262
duplicate results, removing, 62
dynamic typing, 243-245
bindings, 244
in LINQ queries, 246-251
when to use, 246

E

ECMA, 233
Element operators
DefaultIfEmpty, 144-145
ElementAt, 145-147
ElementAtOrDefault, 145-147
First, 147-149
FirstOrDefault, 147-149
Last, 149-151
LastOrDefault, 149-151
Single, 151-153
SingleOrDefault, 151-153
ElementAt operator, 145-147
ElementAtOrDefault operator, 145-147
Empty operator, 155-156
Equality operators, SequenceEqual, 154-155
EqualityComparers
built-in string comparers, 183-185
custom, 183-185
error handling, adding to parallel operators, 298-301
evolution of C#, 233-234
examples of LINQ to Objects query syntax, 30-38
Except operator, 178-179
exceptions, 270
expression trees, 3
extension method format, 41
extension methods, 18

F

features of C#
dynamic typing, 243
bindings, 244
in LINQ queries, 246-251
when to use, 246
named arguments, 240-243
optional parameters, 237-239, 241-243
filtering query results, 49
by index position, 53-54
deferred execution, 51
Where filter
with external methods for evaluation, 52-53
with Lambda Expression, 50
with query expressions, 51
First operator, 147-149
FirstOrDefault operator, 147-149
fluent interfaces, 247

G–H

Generation operators
Empty, 155-156
Range, 156-158
Repeat, 158-159
Geonames example of Parallel LINQ queries, 271-275
GroupBy extension method, 76-77
grouped objects, returning, 59
grouping collection
implementation of grouping operators, 223-225
grouping elements, 75
composite keys, grouping by, 80-83
custom comparers, specifying, 83-87
GroupBy extension method, 76-77
into new type, 88-90
keySelector expression, 77-80
query continuation, 90-93
grouping operators
grouping collection implementation, 223-225
Segment operator, building, 226-232
writing, 222-232
Gustafson, John L., 268

hash partitioning, 278
Hejlsberg, Anders, 2

I

implicitly typed local variables, 23-24
index position
obtaining from query results, 61-62
query results, filtering, 53-54
inner joins, 100
integers, nullable type, 128
interfaces (COM-Interop), adding, 253-256
Intersect operator, 180
Into keyword for query expression format, 45
invoking Parallel LINQ queries, 280

J

join operator, 99-104
join/into keyword combination, performing one-to-many joins, 112-115

joins, 93
  cross joins, 94-97
  Join operator, 103-104
  one-to-many, 111-112
    join/into keyword combination, 112-115
    performance comparisons, 117-119
    subqueries, 115-116
    ToLookup operator, 116-117
  one-to-one, 97
    join operator, 99-101
    performance comparisons, 107-111
    using cross joins, 106-107
    using SingleOrDefault operator, 105-106
    using subqueries, 104-105
  outer joins, 101

K–L

keySelector expressions, handling null values, 77-80

Lambda Expressions, 26-28
  delegates, 27
  Where filters, 50
Last operator, 149-151
  building, 196-201
LastOrDefault operator, 149-151
LINQ
  combining with COM-Interop, 251-260
  queries
    dynamic typing, 246-251
    named arguments, 241-243
    optional parameters, 241-243
LINQ Language Compiler Enhancements, 3
"LINQ Project Overview" whitepaper, 2
LINQ set operators
  Concat, 174, 176
  Distinct, 177-178
  EqualityComparers
    built-in string comparers, 183-185
    custom, 183-185
  Except, 178-179
  Intersect, 180
  Union, 181-183
LINQ to Datasets, 4
LINQ to Entities, 4
LINQ to Objects, 4
  anonymous types, 24-26
  built-in performance optimizations, 200
  Collection Initializers, 22
  contract records, grouping and sorting versus C# 2.0 approach, 5, 7
  data, summarizing from two collections versus C# 2.0 approach, 8-12
  extension methods
    arguments, 18
    declaring, 18-21
  implicity typed local variables, 23-24
  Lambda Expressions, 26-28
  Object Initializers, 21-22
  queries, syntax examples, 30-38
  Query Expressions, 29-30
LINQ to SQL, 4
LINQ to XML, 4
local variables
  implicit typing, 23-24
  query expression format, creating, 44-45
LongCount operator, 129-131
LongSum operator, building, 219-222

M

Max operator, 126-129
Merging operators, Zip, 159-160
Microsoft Excel, building row iterators, 256-260
Min operator, 126-129
  writing, 216-219
Moore's Law, 261
multi-core processors, 263-264
multi-threading, versus code parallelism, 264-267

N

named arguments, 234-243
natural sorting, 69
normalization, 94, 97
null values, handling in keySelector expressions, 78-80
null-coalescing operators, 79
nullable type, 128

O

Object Initializers, 21-22
obtaining index position from query results, 61-62
OfType operator, 134-136
one-to-many joins, 94, 111-112
  join/into keyword combination, 112-115
  performance comparisons, 117-119
  subqueries, 115-116
  ToLookup operator, 116-117
one-to-one inner joins, 94
one-to-one joins, 97
  join operator, 99-101
  performance comparisons, 107-111
  using cross joins, 106-107
  using SingleOrDefault operator, 105-106
  using subqueries, 104-105
operators
  Equality operators, SequenceEqual, 154-155
  Generation operators
    Empty, 155-156
    Range, 156-158
    Repeat, 158-159
  Merging operators, Zip, 159-160
  Parallel LINQ operators
    error handling, 298-301
    testing, 295-297
    writing, 289-294
  Partitioning operators
    Skip, 161-162
    SkipWhile, 163-164
    Take, 161-162
    TakeWhile, 163-164

Quantifier operators
All, 164-166
Any, 166-169
Contains, 169-171
single element operators
Last, building, 196-201
RandomElement, building, 201-208
optional parameters, 234, 237-239
in LINQ queries, 241-243
versus COM-Interop programming, 235-237
ordering Parallel LINQ query results, 281-284
outer joins, 101

P

Parallel LINQ queries, 270
AsSequential operator, 285-287
data ordering, 281-284
data partitioning, 276
chunk partitioning, 277
hash partitioning, 278
range partitioning, 277
striped partitioning, 278
Geonames example, 271-275
invoking, 280
operators
error handling, 298-301
testing, 295-297
writing, 289-294
parallel execution, 279
parallel results, merging, 279
query analysis, 275-276
two-source operators, 287-289
parallel programming
Amdahl's law, 268
drivers, 261-262
exceptions, 270
overhead, 269
synchronization, 269
versus multi-threading, 264-267
parallelization, 64
parameters
optional, 234, 237-239
in LINQ queries, 241-243
versus COM-Interop programming, 235-237
Partitioning operators
Skip, 161-162
SkipWhile, 163-164
Take, 161-162
TakeWhile, 163-164
partitioning schemes, 276
chunk partitioning, 277
hash partitioning, 278
range partitioning, 277
striped partitioning, 278
performance, Amdahl's law, 268
PLINQ (Parallel Extensions to .NET and Parallel LINQ), 4
predicates, 49
Primary Interop Assemblies, 256
processors, multi-core, 263-264
projecting grouped elements into new type, 88-90
projections, 25

Q

Quantifier operators
All, 164-166
Any, 166-169
Contains, 169-171
queries
case in-sensitive string ordering, 65-67
cultural-specific string ordering, 65-67
duplicate results, removing, 62
extension method format, 41
index position of results, obtaining, 61-62
LINQ to Objects, syntax examples, 30-38
Parallel LINQ queries
AsSequential operator, 285-287
chunk partitioning, 277
data ordering, 281-284
data partitioning, 276-277
Geonames example, 271-275
hash partitioning, 278
invoking, 280
operators, error handling, 298-301
operators, testing, 295-297
operators, writing, 289-294
parallel execution, 279
parallel results, merging, 279
query analysis, 275-276
striped partitioning, 278
two-source operators, 287-289
query dot format, 42
query expression format, 42-44
Into keyword, 45
local variables, creating, 44-45
results
custom sort function, specifying, 67-72
reversing order, 65
sorting, 63-64
return type, changing, 54-59
SelectMany operator, 59-61
Standard Query Operators, 14
syntax
choosing, 42
comparing methods, 45-49
Where clause, 49-50
deferred execution, 51
filtering by index position, 53-54
query expressions, 51
with external methods, 52-53
with Lambda Expression, 50
query continuation, 90-93
query dot format, 42
query expression format, 42-44
Into keyword, 45
local variables, creating, 44-45
Where filters, 51

Query Expressions, 29-30
query operators, 121
  aggregation operators
    Aggregate operator, 123-125
    Average operator, 126-129
    Count operator, 129-131
    LongCount operator, 129-131
    Max operator, 126-129
    Min operator, 126-129
    Sum operator, 126-129
    writing, 216-222
  conversion operators
    AsEnumerable, 133
    Cast, 133-134
    OfType, 134-136
    ToArray, 136
    ToDictionary, 136-139
    ToList, 140
    ToLookup, 140-143
  element operators
    DefaultIfEmpty, 144-145
    ElementAt, 145-147
    ElementAtOrDefault, 145-147
    First, 147-149
    FirstOrDefault, 147-149
    Last, 149-151
    LastOrDefault, 149-151
    Single, 151-153
    SingleOrDefault, 151-153
  grouping operators, writing, 222-232
  sequence operators
    TakeRange, building, 210-216
    writing, 208-216
  single element operators, writing, 196-208

R

race conditions, 265-266
RandomElement operator, building, 201-208
Range operator, 156-158
range partitioning, 277
Repeat operator, 158-159
return type, changing, 54
  anonymous type, returning, 58-59
  different type as source, returning, 56-58
  grouped objects, returning, 59
  same type as source, returning, 56
  single result value, returning, 55
returning sequenced elements with SelectMany operator, 59-61
reversing query result order, 65
row iterator, building in Microsoft Excel, 256-260
Rusina, Alexandra, 200

S

Segment operator, building, 226-232
selecting query syntax, 42
selection projections
  query return type, changing, 54-55
    anonymous type, returning, 58-59
    different type as source, returning, 56-58
    grouped objects, returning, 59
    same type as source, returning, 56
    single result value, returning, 55
SelectMany operator, 59-61
sequence operators
  TakeRange, building, 210-216
  writing, 208-216
SequenceEqual operator, 154-155
sequences returning, 59-61
single element operators
  Last, building, 196-201
  RandomElement, building, 201-208
  writing, 196-208
Single operator, 151-153
SingleOrDefault operator, 151-153
  one-to-one joins, 105-106
Skip operator, 161-162
SkipWhile operator, 163-164
sorting query results
  case in-sensitive string ordering, 65-67
  cultural-specific string ordering, 65-67
  custom sort comparison functions, specifying, 67-72
  reversing result sequence, 65
  syntax, 63-64
Soundex equality operator, building, 84-87
Standard Query Operators, 14
striped partitioning, 278
subqueries
  one-to-many joins, performing, 115-116
  one-to-one joins, performing, 104-105
Sum operator, 126, 128-129
summarizing data from two collections, comparing LINQ and C# 2.0 approaches, 8-12
synchronization, 269
syntax
  LINQ to Objects query examples, 30-38
  queries
    choosing, 42
    methods, comparing, 45-49
    query expression format, 43-45
    results, sorting, 63-64

T

Take operator, 161-162
TakeRange operator, building, 210-216
TakeWhile operator, 163-164
ternary operators, 79

testing Parallel LINQ operators, 295-297
this modifier, 18
thread-level parallelism, 264
threading, 264
ToArray operator, 136
ToDictionary operator, 136-139
ToList operator, 140
ToLookup operator, 140-143
- one-to-many joins, performing, 116-117

Toub, Stephen, 278
two-source operators, 287-289

U–V

Union operator, 181-183
unstable sorting types, 64

Visual Studio, adding COM-Interop interfaces, 255-256

W

Where clause
- query expression syntax, 51
- query results, filtering, 49-50
  - by index position, 53-54
  - deferred execution, 51
- with external method for evaluation, 52-53

writing
- grouping operators, 222-232
- Min Operator, 216-219
- Parallel LINQ operators, 289-294
- query operators, grouping operators, 222-232
- sequence operators, 208-216
- single element operators, 196-208
- XML, comparing LINQ and C# 2.0 and 3.0 approaches, 8-12

X–Y–Z

XML, comparing LINQ and C# 2.0 writing approaches, 8-12

Zip operator, 159-160