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

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

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/.

CHAPTER 3

WRITING BASIC QUERIES

Goals of this chapter:

- Understand the LINQ syntax options.
- Introduce how to write basic queries.
- Demonstrate how to filter, project, and order data using LINQ queries.

The main goal of this chapter is to introduce the basics of writing queries. These basics include the syntax options available for writing queries, how to filter data, how to order data, and how to return the exact result set you want. By the end of this chapter, you will understand how to write the most common query elements and, in the following chapter, you will expand this understanding into the more advanced query features of grouping and joining with other sources.

Query Syntax Style Options

Most previous examples in this book have used the query expression syntax, but there are two styles for writing LINQ queries. Not all operators are available through the query expression syntax built into the C# compiler, and to use the remaining operators (or to call your own operators), the extension method query syntax or a combination of the two is necessary. You will continually need to know both styles of query syntax in order to read, write, and understand code written using LINQ.

- **Extension method format** (also known as the dot notation syntax)—The extension method format is simply where multiple extension methods are cascaded together, each returning an `IEnumerable<T>` result to allow the next extension method to flow on from the previous result and so on (known as a *fluid interface*).

```
int[] nums = new int[] {0,4,2,6,3,8,3,1};

var result1 = nums.Where(n => n < 5).OrderBy (n => n);

// or with line-breaks added for clarity
var result2 = nums
              .Where(n => n < 5)
              .OrderBy (n => n);
```

- **Query Expression format** (preferred, especially for joins and groups)—Although not all standard query operators are supported by the query expression syntax, the benefit to the clarity of code when they are is very high. The query expression syntax is much gentler than the extension method syntax in that it simplifies the syntax by removing lambda expressions and by introducing a familiar SQL-like representation.

```
int[] nums = new int[] {0,4,2,6,3,8,3,1};

var result = from n in nums
             where n < 5
             orderby n
             select n;
```

- **Query Dot syntax** (a combination of the two formats)—This format combines a query expression syntax query surrounded by parentheses, followed by more operators using the Dot Notation syntax. As long as the query expression returns an `IEnumerable<T>`, it can be followed by an extension method chain.

```
int[] nums = new int[] {0,4,2,6,3,8,3,1};

var result = (from n in nums
              where n < 5
              orderby n
              select n).Distinct();
```

WHICH QUERY SYNTAX TO USE? Personal preference will dictate this, but the goal is to use the syntax that is easiest to read and that will help developers who come after you to understand your intentions. With this in mind, don't unnecessarily mix the syntax styles in one query; mixing the styles makes the query harder to read by forcing the reader to count brackets and determine

which part of a query the extension method syntax applies to. If you do mix styles, keep them together; for example, use the query expression at the start surrounded by parentheses, then the extension methods at the end for the necessary operators (as shown in all examples in this book when mixing was needed).

My preference (because of my SQL background perhaps) is to use the query expression syntax wherever possible and then revert to using the extension method syntax when using an operator not supported by query expressions (the `Distinct` operator, for instance), but I always add these operators at the end of the query. I'll sometimes use all expression method syntax but never when the query has a `Join` or `GroupBy` operator.

Each query syntax has its own merits and pitfalls, which the following sections cover in detail.

Query Expression Syntax

The query expression syntax provided in C# 3.0 and later versions makes queries clearer and more concise. The compiler converts the query expression into extension method syntax during compilation, so the choice of which syntax to use is based solely on code readability.

Figure 3-1 shows the basic form of query expressions built into C# 3.0.

```
IEnumerable<T>|T  query-expression-identifier  =
    from  identifier  in  expression
    let opt  identifier  =  expression
    where opt  boolean-expression
    join opt  type opt  identifier  in  expression  on
              expression  equals  expression  into opt  identifier
    orderby opt  ordering-clause(s)  ascending|descending  opt
    group opt  expression  by  expression  into  opt  identifier
    select  expression  into opt  identifier
```

Figure 3-1 The basic query expression syntax form. The C# 3.0 Language Specification outlines exactly how this form is translated into extension methods for compilation.

NOTE The fact that the order of the keywords is different in SQL is unfortunate for those who are SQL masters; however, one very compelling reason for the difference was to improve the developer experience. The From-Where-Select order allows the editing environment (Visual Studio in this case) to provide full Intellisense support when writing the query. The moment you write the `from` clause, the properties of that element appear as you then write the `where` clause. This wouldn't be the case (and isn't in SQL Server's query editing tools) if the C# designers followed the more familiar Select-From-Where keyword ordering.

Most of the query expression syntax needs no explanation for developers experienced with other query syntax, like SQL. Although the order is different than in traditional query languages, each keyword name gives a strong indication as to its function, the exception being the `let` and `into` clauses, which are described next.

Let—Create a Local Variable

Queries can often be written with less code duplication by creating a local variable to hold the value of an intermediate calculation or the result of a subquery. The `let` keyword enables you to keep the result of an expression (a value or a subquery) in scope throughout the rest of the query expression being written. Once assigned, the variable cannot be reassigned with another value.

In the following code, a local variable is assigned, called `average`, that holds the average value for the entire source sequence, calculated once but used in the Select projection on each element:

```
var variance = from element in source
               let average = source.Average()
               select Math.Pow((element - average), 2);
```

The `let` keyword is implemented purely by the compiler, which creates an anonymous type that contains both the original range variable (`element` in the previous example) and the new `let` variable. The previous query maps directly to the following (compiler translated) extension method query:

```
var variance =
   source.Select (
      element =>
         new
```

```
            {
                element = element,
                average = source.Average ()
            }
    )
    .Select (temp0 =>
                Math.Pow (
                    ((double)temp0.element - temp0.average)
                    , 2));
```

Each additional `let` variable introduced will cause the current anonymous type to be cascaded within another anonymous type containing itself and the additional variable—and so on. However, all of this magic is transparent when writing a query expression.

Into—Query Continuation

The `group`, `join`, and `select` query expression keywords allow the resulting sequence to be captured into a local variable and then used in the rest of the query. The `into` keyword allows a query to be continued by using the result stored into the local variable at any point after its definition.

The most common point `into` is employed is when capturing the result of a group operation, which along with the built-in join features is covered extensively in Chapter 4, "Grouping and Joining Data." As a quick preview, the following example groups all elements of the same value and stores the result in a variable called `groups`; by using the `into` keyword (in combination with the `group` keyword), the `groups` variable can participate and be accessed in the remaining query statement.

```
var groupings = from element in source
                group element by element into groups
                select new {
                    Key = groups.Key,
                    Count = groups.Count()
};
```

Comparing the Query Syntax Options

Listing 3-1 uses extension method syntax, and Listing 3-2 uses query expression syntax, but they are functionally equivalent, with both generating the identical result shown in Output 3-1. The clarity of the code in the query expression syntax stems from the removal of the lambda expression semantics and the SQL style operator semantics. Both syntax styles are

functionally identical, and for simple queries (like this example), the benefit of code clarity is minimal.

Listing 3-1 Query gets all contacts in the state of "WA" ordered by last name and then first name using extension method query syntax—see Output 3-1

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

var q = contacts.Where(c => c.State == "WA")
                .OrderBy(c => c.LastName)
                .ThenBy(c => c.FirstName);

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

Listing 3-2 The same query as in Listing 3-1 except using query expression syntax—see Output 3-1

```
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);
```

Output 3-1

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

There are extensive code readability advantages to using the query expression syntax over the extension method syntax when your query

contains join and/or group functionality. Although not all joining and grouping functionality is natively available to you when using the query expression syntax, the majority of queries you write will not require those extra features. Listing 3-3 demonstrates the rather clumsy extension method syntax for Join (clumsy in the fact that it is not clear what each argument means in the `GroupBy` extension method just by reading the code). The functionally equivalent query expression syntax for this same query is shown in Listing 3-4. Both queries produce the identical result, as shown in Output 3-2.

If it is not clear already, my personal preference is to use the query expression syntax whenever a `Join` or `GroupBy` operation is required in a query. When a standard query operator isn't supported by the query expression syntax (as is the case for the `.Take` method for example), you parenthesize the query and use extension method syntax from that point forward as Listing 3-4 demonstrates.

Listing 3-3 Joins become particularly complex in extension method syntax. This query returns the first five call-log details ordered by most recent—see Output 3-2

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

var q = callLog.Join(contacts,
                     call => call.Number,
                     contact => contact.Phone,
                     (call, contact) => new
                     {
                         contact.FirstName,
                         contact.LastName,
                         call.When,
                         call.Duration
                     })
               .OrderByDescending(call => call.When)
               .Take(5);

foreach (var call in q)
    Console.WriteLine("{0} - {1} {2} ({3}min)",
        call.When.ToString("ddMMM HH:m"),
        call.FirstName, call.LastName, call.Duration);
```

Listing 3-4 Query expression syntax of the query identical to that shown in Listing 3-3—see Output 3-2

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

var q = (from call in callLog
         join contact in contacts on
            call.Number equals contact.Phone
         orderby call.When descending
         select new
         {
             contact.FirstName,
             contact.LastName,
             call.When,
             call.Duration
         }).Take(5);

foreach (var call in q)
    Console.WriteLine("{0} - {1} {2} ({3}min)",
        call.When.ToString("ddMMM HH:m"),
        call.FirstName, call.LastName, call.Duration);
```

Output 3-2

```
07Aug 11:15 - Stewart Kagel (4min)
07Aug 10:35 - Collin Zeeman (2min)
07Aug 10:5 - Mack Kamph (1min)
07Aug 09:23 - Ariel Hazelgrove (15min)
07Aug 08:12 - Barney Gottshall (2min)
```

EXTENSION METHOD DEVELOPER TIPS

- Express the most limiting query method first; this reduces the workload of the successive operators.
- Split each operator onto a different line (including the period joiner). This allows you to comment out individual operators when debugging.
- Be consistent—within an application use the same style throughout.
- To make it easier to read queries, don't be afraid to split up the query into multiple parts and indent to show hierarchy.

QUERY EXPRESSION DEVELOPER TIPS

- If you need to mix extension methods with query expressions, put them at the end.
- Keep each part of the query expression on a separate line to allow you to individually comment out individual clauses for debugging.

How to Filter the Results (Where Clause)

One of the main tasks of a LINQ query is to restrict the results from a larger collection based on some criteria. This is achieved using the `Where` operator, which tests each element within a source collection and returns only those elements that return a true result when tested against a given predicate expression. A *predicate* is simply an expression that takes an element of the same type of the items in the source collection and returns true or false. This predicate is passed to the `Where` clause using a lambda expression.

The extension method for the `Where` operator is surprisingly simple; it iterates the source collection using a `foreach` loop, testing each element as it goes, returning those that pass. Here is a close facsimile of the actual code in the `System.Linq` library:

```
public delegate TResult Func<T1, TResult>(T1 arg1);

public static IEnumerable<T> Where<T>(
          this IEnumerable<T> source,
          Func<T, bool> predicate) {

    foreach (T element in source) {
        if (predicate(element))
              yield return element;
    }
}
```

The LINQ to Objects `Where` operator seems pretty basic on the surface, but its implementation is simple due to the powerful `yield return` statement that first appeared in the .NET Framework 2.0 to make building collection iterators easier. Any code implementing the built-in enumeration pattern (as codified by any collection that implements the interface

IEnumerable) natively supports callers asking for the next item in a collection—at which time the next item for return is computed (supported by the foreach keyword as an example). Any collection implementing the IEnumerable<T> pattern (which also implements IEnumerable) will be extended by the Where operator, which will return a single element at a time when asked, as long as that element satisfies the predicate expression (returns true).

Filter predicate expressions are passed to the extension method using a lambda expression (for a recap on what a lambda expression is see Chapter 2, "Introducing LINQ to Objects"), although if the query expression syntax is used, the filter predicate takes an even cleaner form. Both of these predicate expression styles are explained and covered in detail in the following sections.

Where Filter Using a Lambda Expression

When forming a predicate for the Where operator, the predicate takes an input element of the same type as the elements in the source collection and returns true or false (a Boolean value). To demonstrate using a simple Where clause predicate, consider the following code:

```
string[] animals = new string[] { "Koala", "Kangaroo",
    "Spider", "Wombat", "Snake", "Emu", "Shark",
    "Sting-Ray", "Jellyfish" };

var q = animals.Where(
    a => a.StartsWith("S") && a.Length > 5);

foreach (string s in q)
    Console.WriteLine(s);
```

In this code, each string value from the animals array is passed to the Where extension method in a range variable called a. Each string in a is evaluated against the predicate function, and only those strings that pass (return true) are returned in the query results. For this example, only two strings pass the test and are written to the console window. They are

```
Spider

Sting-Ray
```

The C# compiler behind the scenes converts the lambda expression into a standard anonymous method call (the following code is functionally equivalent):

```
var q = animals.Where(
    delegate(string a) {
          return a.StartsWith("S") && a.Length > 5; });
```

What Is Deferred Execution?

The `Where` clause will only begin testing the predicate when somebody (you through a `foreach` statement or one of the other standard query operators that have an internal `foreach` statement) tries to iterate through the results; until then, the iterator framework just remembers exactly where it was the last time it was asked for an element. This is called deferred execution, and it allows you some predictability and control over when and how a query is executed. If you want results immediately you can call `ToList()`, `ToArray()` or one of the other standard operators that cause immediate actualization of the results to another form; otherwise, the query will begin evaluation only when you begin iterating over it.

Where Filter Query Expressions (Preferred)

The query expression `where` clause syntax drops the explicit range variable definition and the lambda expression operator (`=>`), making it more concise and more familiar to the SQL-style clauses that many developers understand. It is the preferred syntax for these reasons. Rewriting the previous example using query expression syntax demonstrates these differences, as follows:

```
string[] animals = new string[] { "Koala", "Kangaroo",
    "Spider", "Wombat", "Snake", "Emu", "Shark",
    "Sting-Ray", "Jellyfish" };

var q = from a in animals
        where a.StartsWith("S") && a.Length > 5
        select a;

foreach (string s in q)
    Console.WriteLine(s);
```

Using an External Method for Evaluation

Although you can write queries and inline the code for the filter predicate, you don't have to. If the predicate is lengthy and might be used in more than one query expression, you should consider putting it in its own method body (good practice for any duplicated code). Rewriting the previous examples using an external predicate function shows the technique:

```
string[] animals = new string[] { "Koala", "Kangaroo",
     "Spider", "Wombat", "Snake", "Emu", "Shark",
     "Sting-Ray", "Jellyfish" };

var q = from a in animals
        where MyPredicate(a)
        select a;

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

public bool MyPredicate(string a)
{
     if (a.StartsWith("S") && a.Length > 5)
         return true;
     else
         return false;
}
```

To further demonstrate this technique with a slightly more complex scenario, the code shown in Listing 3-5 creates a predicate method that encapsulates the logic for determining "a deadly animal." By encapsulating this logic in one method, it doesn't have to be duplicated in multiple places in an application.

Listing 3-5 `Where` clause using external method—see Output 3-3

```
string[] animals = new string[] { "Koala", "Kangaroo",
     "Spider", "Wombat", "Snake", "Emu", "Shark",
     "Sting-Ray", "Jellyfish" };

var q = from a in animals
        where IsAnimalDeadly(a)
        select a;
```

```
foreach (string s in q)
    Console.WriteLine("A {0} can be deadly.", s);


public static bool IsAnimalDeadly(string s)
{
    string[] deadly = new string[] {"Spider", "Snake",
        "Shark", "Sting-Ray", "Jellyfish"};

    return deadly.Contains(s);
}
```

Output 3-3

```
A Spider can be deadly.
A Snake can be deadly.
A Shark can be deadly.
A Sting-Ray can be deadly.
A Jellyfish can be deadly.
```

Filtering by Index Position

The standard query operators expose a variation of the `Where` operator that surfaces the index position of each collection element as it progresses. The zero-based index position can be passed into a lambda expression predicate by assigning a variable name as the second argument (after the element range variable). To surface the index position, a lambda expression must be used, and this can only be achieved using the extension method syntax. Listing 3-6 demonstrates the simplest usage, in this case simply returning the first and only even-index positioned elements from the source collection, as shown in Output 3-4.

Listing 3-6 The index position can be used as part of the **`Where`** clause predicate expression when using lambda expressions—see Output 3-4

```
string[] animals = new string[] { "Koala", "Kangaroo",
    "Spider", "Wombat", "Snake", "Emu", "Shark",
    "Sting-Ray", "Jellyfish" };

// get the first then every other animal (index is odd)
var q = animals.Where((a, index) => index % 2 == 0);
```

```
foreach (string s in q)
    Console.WriteLine(s);
```

Output 3-4

```
Koala
Spider
Snake
Shark
Jellyfish
```

How to Change the Return Type (Select Projection)

When you write queries against a database system in SQL, specifying a set of columns to return is second nature. The goal is to limit the columns returned to only those necessary for the query in order to improve performance and limit network traffic (the less data transferred, the better). This is achieved by listing the column names after the `Select` clause in the following format. In most cases, only the columns of interest are returned using the SQL syntax form:

```
Select * from Contacts

Select ContactId, FirstName, LastName from Contacts
```

The first query will return every column (and row) of the contacts table; the second will return only the three columns explicitly listed (for every row), saving server and network resources. The point is that the SQL language syntax allows a different set of rows that does not match any existing database table, view, or schema to be the structure used in returning data. Select projections in LINQ query expressions allow us to achieve the same task. If only few property values are needed in the result set, those properties or fields are the only ones returned.

LINQ selection projections allow varied and powerful control over how and what data shape is returned from a query expression.

The different ways a select projection can return results are

- As a single result value or element
- In an `IEnumerable<T>` where `T` is of the same type as the source items
- In an `IEnumerable<T>` where `T` is any existing type constructed in the select projection
- In an `IEnumerable<T>` where `T` is an anonymous type created in the select projection
- In an `IEnumberable<IGrouping<TKey, TElement>>`, which is a collection of grouped objects that share a common key

Each projection style has its use, and each style is explained by example in the following sections.

HOW MUCH DATA ARE YOU PROJECTING IN A SELECT PROJECTION? As with all good data-access paradigms, the goal should be to return the fewest properties as possible when defining a query result shape. This reduces the memory footprint and makes the result set easier to code against because there are fewer properties to wade through.

Return a Single Result Value or Element

Some of the standard query operators return a single value as the result, or a single element from the source collection; these operators are listed in Table 3-1. Each of these operators end any cascading of results into another query, and instead return a single result value or source element.

Table 3-1 Sample Set of Operators that Return a Specific Result Value Type (covered in Chapters 5 and 6)

Return Type	Standard Query Operator
Numeric	`Aggregate, Average, Max, Min, Sum, Count, LongCount`
Boolean	`All, Any, Contains, SequenceEqual`
Type <T>	`ElementAt, ElementAtOrDefault, First, FirstOrDefault, Last, LastOrDefault, Single, SingleOrDefault, DefaultIfEmpty`

As an example, the following simple query returns the last element in the integer array, writing the number 2 to the Console window:

```
int[] nums = new int[] { 5, 3, 4, 2 };
int last = nums.Last();
ConsoleWriteLine(last);
```

Return the Same Type as the Source—IEnumerable<TSource>

The most basic projection type returns a filtered and ordered subset of the original source items. This projection is achieved by specifying the range variable as the argument after the `select` keyword. The following example returns an `IEnumerable<Contact>`, with the type `Contact` being inferred from the element type of the source collection:

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

IEnumerable<Contact> q = from c in contacts
                         select c;
```

A more appropriate query would filter the results and order them in some convenient fashion. You are still returning a collection of the same type, but the number of elements and their order will be different.

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

IEnumerable<Contact> q = from c in contacts
                         where c.State == "WA"
                         orderby c.LastName,
                                 c.FirstName ascending
                         select c;
```

Return a Different Type Than the Source—IEnumerable<TAny>

Any type can be projected as part of the `select` clause, not just the source type. The target type can be any available type that could otherwise be

manually constructed with a plain `new` statement from the scope of code being written.

If the type being constructed has a parameterized constructor containing all of the parameters you specifically need, then you simply call that constructor. If no constructor matches the parameter's needed for this projection, either create one or consider using the C# 3.0 type initializer syntax (as covered in Chapter 2). The benefit of using the new type initializer semantics is that it frees you from having to define a specific constructor each time a new projection signature is needed to cater for different query shapes. Listing 3-7 demonstrates how to project an `IEnumerable<ContactName>` using both constructor semantics.

NOTE Resist the temptation to overuse the type initializer syntax. It requires that all properties being initialized through this syntax be read and write (have a getter and setter). If a property should be read-only, don't make it read/write just for this feature. Consider making those constructor parameters optional using the C# 4.0 Optional Parameter syntax described in Chapter 8, "C# 4.0 Features."

Listing 3-7 Projecting to a collection of a new type—constructed using either a specific constructor or by using type initializer syntax

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

// using a parameterized constructor
IEnumerable<ContactName> q1 =
    from c in contacts
    select new ContactName(
           c.LastName + ", " + c.FirstName,
           (DateTime.Now - c.DateOfBirth).Days / 365);

// using Type Initializer semantics
// note: The type requires a parameterless constructor
IEnumerable<ContactName> q2 =
  from c in contacts
  select new ContactName
  {
      FullName = c.LastName + ", " + c.FirstName,
      YearsOfAge =
```

```
            (DateTime.Now - c.DateOfBirth).Days / 365
    };

// ContactName class definition
public class ContactName
{
    public string FullName { get; set; }
    public int YearsOfAge { get; set; }

    // constructor needed for
    // object initialization example
    public ContactName() {
    }

    // constructor required for
    // type projection example
    public ContactName(string name, int age)
    {
        this.FullName = name;
        this.YearsOfAge = age;
    }
}
```

Return an Anonymous Type—IEnumerable<TAnonymous>

Anonymous types is a new language feature introduced in C# 3.0 where the compiler creates a type on-the-fly based on the object initialization expression (the expression on the right side of the initial = sign). Discussed in detail in Chapter 2, this new type is given an uncallable name by the compiler, and without the `var` keyword (implicitly typed local variables), there would be no way to compile the query. The following query demonstrates projecting to an `IEnumerable<T>` collection where `T` is an anonymous type:

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

var q = from c in contacts
        select new
        {
```

```
            FullName = c.LastName + ", " + c.FirstName,
            YearsOfAge =
                (DateTime.Now - c.DateOfBirth).Days / 365
        };
```

The anonymous type created in the previous example is composed of two properties, `FullName` and `YearsOfAge`.

Anonymous types free us from having to write and maintain a specific type definition for every different return result collection needed. The only drawback is that these types are method-scoped and cannot be used outside of the method they are declared by (unless passed as a `System.Object` type, but this is not recommended because property access to this object subsequently will need to use reflection).

Return a Set of Grouped Objects—IEnumerable<IGrouping<TKey,TElement>>

It is possible for LINQ to Objects to group results that share common source values or any given characteristic that can be equated with an expression using the `group by` query expression keyword or the `GroupBy` extension method. This topic is covered in great detail in Chapter 4.

How to Return Elements When the Result Is a Sequence (Select Many)

The `SelectMany` standard query operator flattens out any `IEnumerable<T>` result elements, returning each element individually from those enumerable sources before moving onto the next element in the result sequence. In contrast, the `Select` extension method would stop at the first level and return the `IEnumerable<T>` element itself.

Listing 3-8 demonstrates how `SelectMany` differs from `Select`, with each variation aiming to retrieve each individual word within a set of phrase strings. To retrieve the words in Option 1, a sub `for` loop is required, but `SelectMany` automatically performs this subiteration of the original result collection, as shown in Option 2. Option 3 demonstrates that the same result can be achieved using multiple `from` statements in a query expression (which maps the query to use `SelectMany` operator behind the scenes). The Console output is shown in Output 3-5.

Listing 3-8 `Select` versus `SelectMany`—`SelectMany` drills into an `IEnumerable` result, returning its elements—see Output 3-5

```
string[] sentence = new string[] { "The quick brown",
    "fox jumps over", "the lazy dog."};

Console.WriteLine("option 1:"); Console.WriteLine("---------");

// option 1: Select returns three string[]'s with
// three strings in each.
IEnumerable<string[]> words1 =
    sentence.Select(w => w.Split(' '));

// to get each word, we have to use two foreach loops
foreach (string[] segment in words1)
    foreach (string word in segment)
        Console.WriteLine(word);

Console.WriteLine();
Console.WriteLine("option 2:"); Console.WriteLine("---------");

// option 2: SelectMany returns nine strings
// (sub-iterates the Select result)
IEnumerable<string> words2 =
    sentence.SelectMany(segment => segment.Split(' '));

// with SelectMany we have every string individually
foreach (var word in words2)
    Console.WriteLine(word);

// option 3: identical to Opt 2 above written using
// the Query Expression syntax (multiple froms)
IEnumerable<string> words3 =
    from segment in sentence
    from word in segment.Split(' ')
    select word;
```

Output 3-5

```
option 1:
---------
The
quick
brown
```

```
fox
jumps
over
the
lazy
dog.

option 2:
---------
The
quick
brown
fox
jumps
over
the
lazy
dog.
```

How does the `SelectMany` extension method work? It creates a nested `foreach` loop over the original result, returning each subelement using `yield return` statements. A close facsimile of the code behind `SelectMany` takes the following form:

```
static IEnumerable<S> SelectManyIterator<T, S>(
     this IEnumerable<T> source,
     Func<T, IEnumerable<S>> selector)
{
     foreach (T element in source)
     {
          foreach (S subElement in selector(element))
          {
               yield return subElement;
          }
     }
}
```

How to Get the Index Position of the Results

`Select` and `SelectMany` expose an overload that surfaces the index position (starting at zero) for each returned element in the Select projection. It is surfaced as an overloaded parameter argument of the selector lambda

expression and is only accessible using the extension method query syntax. Listing 3-9 demonstrates how to access and use the index position value in a Select projection. As shown in Output 3-6, this example simply adds a ranking number for each select result string.

Listing 3-9 A zero-based index number is exposed by the **`Select`** and **`SelectMany`** operators—see Output 3-6

```
List<CallLog> callLog = CallLog.SampleData();

var q = callLog.GroupBy(g => g.Number)
               .OrderByDescending(g => g.Count())
               .Select((g, index) => new
               {
                   number = g.Key,
                   rank = index + 1,
                   count = g.Count()
               });

foreach (var c in q)
    Console.WriteLine(
        "Rank {0} - {1}, called {2} times.",
        c.rank, c.number, c.count);
```

Output 3-6

```
Rank 1 - 885 983 8858, called 6 times.
Rank 2 - 546 607 5462, called 6 times.
Rank 3 - 364 202 3644, called 4 times.
Rank 4 - 603 303 6030, called 4 times.
Rank 5 - 848 553 8487, called 4 times.
Rank 6 - 165 737 1656, called 2 times.
Rank 7 - 278 918 2789, called 2 times.
```

How to Remove Duplicate Results

The `Distinct` standard query operator performs the role of returning only unique instances in a sequence. The operator internally keeps track of the elements it has returned and skips the second and subsequent duplicate

elements as it returns resulting elements. This operator is covered in more detail in Chapter 6, "Working with Set Data," when its use in set operations is explored.

The `Distinct` operator is not supported in the query expression syntax, so it is often appended to the end of a query using extension method syntax. To demonstrate how it is used, the following code removes duplicate strings. The Console output from this code is

```
Peter
Paul
Mary
Janet
```

```
string[] names = new string[] { "Peter", "Paul",
    "Mary", "Peter", "Paul", "Mary", "Janet" };

var q = (from s in names
         where s.Length > 3
         select s).Distinct();

foreach (var name in q)
    Console.WriteLine(name);
```

How to Sort the Results

LINQ to Objects has comprehensive support for ordering and sorting results. Whether you need to sort in ascending order, descending order using different property values in any sequence, or all the way to writing a specific ordering algorithm of your own, LINQ's sorting features can accommodate the full range of ordering requirements.

Basic Sorting Syntax

The resulting collection of results from a query can be sorted in any desired fashion, considering culture and case sensitivity. When querying using extension method syntax, the `OrderBy`, `OrderByDescending`, `ThenBy`, and `ThenByDescending` standard query operators manage this process. The `OrderBy` and `ThenBy` operators sort in an ascending order (for example, a to z), and the `OrderByDescending` and `ThenByDescending` operators sort in descending order (z to a). Only the first sorting extension can use the `OrderBy` operators, and each subsequent sorting expression must use the

`ThenBy` operators, of which there can be zero or many depending on how much control you want over the subsorting when multiple elements share equal order after the previous expressions.

The following samples demonstrate sorting a source sequence first by the `[w]` key, then in descending order by the `[x]` key, and then in ascending order by the `[y]` key:

```
[source].OrderBy([w])
        .ThenByDescending([x])
        .ThenBy([y]);
```

When using the query expression syntax, each sorting key and the optional direction keyword needs to be separated by a comma character. If the descending or ascending direction keywords are not specified, LINQ assumes ascending order.

```
from [v] in [source]
orderBy [w], [x] descending, [y]
select [z];
```

The result from ordering a collection will be an `IOrderedEnumerable<T>`, which implements `IEnumerable<T>` to allow further query operations to be cascaded end-to-end.

The ordering extension methods are implemented using a basic but efficient Quicksort algorithm (see http://en.wikipedia.org/wiki/Quicksort for further explanation of how this algorithm works). The implementation LINQ to Objects uses is a sorting type called *unstable*, which simply means that elements that compare to equal key values may not retain their relative positions to the source collections (although this is simply solved by cascading the result into a `ThenBy` or `ThenByDescending` operator). The algorithm is fairly fast, and it lends itself to parallelization, which is certainly leveraged by Microsoft's investment in Parallel LINQ.

What Is Parallelization?

Parallelization refers to improving performance of applications by fully leveraging multiple processors and multiple-cores on those processors running code. Parallelization is covered in detail in Chapter 9, "Parallel LINQ to Objects," which also demonstrates how LINQ queries can fully benefit from multicore and multiprocessor hardware improvements.

Reversing the Order of a Result Sequence (Reverse)

Another ordering extension method that reverses an entire sequence is the `Reverse` operator. It is simply called in the form: `[source].Reverse();`. An important point to note when using the `Reverse` operator is that it doesn't test the equality of the elements or carry out any sorting; it simply returns elements starting from the last element, back to the first element. The order returned will be the exact reciprocal of the order that would have been returned from the result sequence. The following example demonstrates the `Reverse` operator, returning `T A C` in the Console window:

```
string[] letters = new string[] { "C", "A", "T" };
var q = letters.Reverse();
foreach (string s in q)
    Console.Write(" " + s);
```

Case Insensitive and Cultural-specific String Ordering

Any standard query operator that involves sorting has an overload that allows a specific comparer function to be supplied (when written using extension method syntax). The .NET class libraries contain a handy helper class called `StringComparer`, which has a set of predefined static comparers ready for use. The comparers allow us to alter string sorting behavior, controlling case-sensitivity and current culture (the language setting for the current thread). Table 3-2 lists the static Comparer instances that can be used in any `OrderBy` or `ThenBy` ascending or descending query operator. (In addition, see the "Custom EqualityComparers When Using LINQ Set Operators" section in Chapter 6, which is specifically about the built-in string comparers and custom comparers.)

Table 3-2 The Built-in **`StringComparer`** Functions to Control String Case Sensitivity and Culture-aware String Ordering

Comparer	Description
`CurrentCulture`	Performs a case-sensitive string comparison using the word comparison rules of the current culture.
`CurrentCultureIgnoreCase`	Performs case-insensitive string comparisons using the word comparison rules of the current culture.

Table 3-2 The Built-in `StringComparer` Functions to Control String Case Sensitivity and Culture-aware String Ordering

Comparer	Description
`InvariantCulture`	Performs a case-sensitive string comparison using the word comparison rules of the invariant culture.
`InvariantCultureIgnoreCase`	Performs a case-insensitive string comparison using the word comparison rules of the invariant culture.
`Ordinal`	Performs a case-sensitive ordinal string comparison.
`OrdinalIgnoreCase`	Performs a case-insensitive ordinal string comparison.

Listing 3-10 demonstrates the syntax and effect of using the built-in string comparer instances offered by the .NET Framework. The Console output is shown in Output 3-7, where the default case-sensitive result can be forced to case-insensitive.

Listing 3-10 Case and culture sensitive/insensitive ordering using `StringComparer` functions—see Output 3-7

```
string[] words = new string[] {
    "jAnet", "JAnet", "janet", "Janet" };

var cs = words.OrderBy(w => w);
var ci = words.OrderBy(w => w,
    StringComparer.CurrentCultureIgnoreCase);

Console.WriteLine("Original:");
foreach (string s in words)
    Console.WriteLine(" " + s);

Console.WriteLine("Case Sensitive (default):");
foreach (string s in cs)
    Console.WriteLine(" " + s);

Console.WriteLine("Case Insensitive:");
foreach (string s in ci)
    Console.WriteLine(" " + s);
```

Output 3-7

```
Original:
 jAnet
 JAnet
 janet
 Janet
Case Sensitive (default):
 janet
 jAnet
 Janet
 JAnet
Case Insensitive:
 jAnet
 JAnet
 janet
 Janet
```

Specifying Your Own Custom Sort Comparison Function

To support any sorting order that might be required, custom sort comparison classes are easy to specify. A custom compare class is a class based on a standard .NET Interface called `IComparer<T>`, which exposes a single method: `Compare`. This interface is not specifically for LINQ; it is the basis for all .NET Framework classes that require sorting (or custom sorting) capabilities.

Comparer functions work by returning an integer result, indicating the relationship between a pair of instance types. If the two types are deemed equal, the function returns zero. If the first instance is less than the second instance, a negative value is returned, or if the first instance is larger than the second instance, the function returns a positive value. How you equate the integer result value is entirely up to you.

To demonstrate a custom `IComparer<T>`, Listing 3-11 demonstrates a comparison function that simply shuffles (in a random fashion) the input source. The algorithm simply makes a random choice as to whether two elements are less than or greater than each other. Output 3-8 shows the Console output from a simple sort of a source of strings in an array, although this result will be different (potentially) each time this code is executed.

Listing 3-11 Ordering using our custom **`IComparer<T>`** implementation to shuffle the results—see Output 3-8

```
public class RandomShuffleStringSort<T> : IComparer<T>
{
     internal Random random = new Random();

     public int Compare(T x, T y)
     {
          // get a random number: 0 or 1
          int i = random.Next(2);

          if (i == 0)
               return -1;
          else
               return 1;
     }
}

string[] strings = new string[] { "1-one", "2-two",
   "3-three", "4-four", "5-five" };

var normal = strings.OrderBy(s => s);
var custom = strings.OrderBy(s => s,
   new RandomShuffleStringSort<string>());

Console.WriteLine("Normal Sort Order:");
foreach (string s in normal) {
     Console.WriteLine(" " + s);
}

Console.WriteLine("Custom Sort Order:");
foreach (string s1 in custom) {
     Console.WriteLine(" " + s1);
}
```

Output 3-8

```
Normal Sort Order:
 1-one
 2-two
 3-three
 4-four
 5-five
```

```
Custom Sort Order:
 1-one
 2-two
 5-five
 4-four
 3-three
```

A common scenario that has always caused me trouble is where straight alphabetical sorting doesn't properly represent alpha-numeric strings. The most common example is alphabetic sorting strings such as `File1`, `File10`, `File2`. Naturally, the desired sorting order would be `File1`, `File2`, `File10`, but that's not alphabetical. A custom `IComparer` that will sort the alphabetic part and then the numeric part separately is needed to achieve this common scenario. This is called *natural sorting*.

Listing 3-12 and Output 3-9 demonstrate a custom sort class that correctly orders alpha strings that end with numbers. Anywhere this sort order is required, the class name can be passed into any of the `OrderBy` or `ThenBy` extension methods in the following way:

```
string[] partNumbers = new string[] { "SCW10", "SCW1",
     "SCW2", "SCW11", "NUT10", "NUT1", "NUT2", "NUT11" };

var custom = partNumbers.OrderBy(s => s,
     new AlphaNumberSort());
```

The code in Listing 3-12 first checks if either input string is null or empty. If either string is empty, it calls and returns the result from the default comparer (no specific alpha-numeric string to check). Having determined that there are two actual strings to compare, the numeric trailing section of each string is extracted into the variables `numericX` and `numericY`. If either string doesn't have a trailing numeric section, the result of the default comparer is returned (if no trailing numeric section exists for one of the strings, then a straight string compare is adequate). If both strings have a trailing numeric section, the alpha part of the strings is compared. If the strings are different, the result of the default comparer is returned (if the strings are different, the numeric part of the string is irrelevant). If both alpha parts are the same, the numeric values in `numericX` and `numericY` are compared, and that result is returned. The final result is that all strings are sorted alphabetically, and where the string part is the same between elements, the numeric section controls the final order.

Listing 3-12 Sorting using a custom comparer. This comparer properly sorts strings that end with a number—see Output 3-9

```
public class AlphaNumberSort : IComparer<string>
{
    public int Compare(string a, string b)
    {
        StringComparer sc =
            StringComparer.CurrentCultureIgnoreCase;

        // if either input is null or empty,
        // do a straight string comparison
        if (string.IsNullOrEmpty(a) ||
            string.IsNullOrEmpty(b))
                return sc.Compare(a, b);

        // find the last numeric sections
        string numericX = FindTrailingNumber(a);
        string numericY = FindTrailingNumber(b);

        // if there is a numeric end to both strings,
        // we need to investigate further
        if (numericX != string.Empty &&
            numericY != string.Empty)
        {
            // first, compare the string prefix only
            int stringPartCompareResult =
                sc.Compare(
                    a.Remove(a.Length - numericX.Length),
                    b.Remove(b.Length - numericY.Length));

            // the strings prefix are different,
            // return the comparison result for the strings
            if (stringPartCompareResult != 0)
                return stringPartCompareResult;

            // the strings prefix is the same,
            // need to test the numeric sections as well
            double nX = double.Parse(numericX);
            double nY = double.Parse(numericY);
            return nX.CompareTo(nY);
        }
        else
            return sc.Compare(a, b);
    }
```

```
    private static string FindTrailingNumber(string s)
    {
        string numeric = string.Empty;
        for (int i = s.Length - 1; i > -1; i--)
        {
            if (char.IsNumber(s[i]))
                numeric = s[i] + numeric;
            else
                break;
        }
        return numeric;
    }
}

string[] partNumbers = new string[] { "SCW10", "SCW1",
    "SCW2", "SCW11", "NUT10", "NUT1", "NUT2", "NUT11" };

var normal = partNumbers.OrderBy(s => s);
var custom = partNumbers.OrderBy(s => s,
    new AlphaNumberSort());

Console.WriteLine("Normal Sort Order:");
foreach (string s in normal)
    Console.WriteLine(" " + s);

Console.WriteLine("Custom Sort Order:");
foreach (string s in custom)
    Console.WriteLine(" " + s);
```

Output 3-9

```
Normal Sort Order:
NUT1
NUT10
NUT11
NUT2
SCW1
SCW10
SCW11
SCW2
```

```
Custom Sort Order:
 NUT1
 NUT2
 NUT10
 NUT11
 SCW1
 SCW2
 SCW10
 SCW11
```

NOTE To achieve the same result in most Windows operating systems (not Windows 2000, but ME, XP, 2003, Vista, and Windows 7) and without guarantee that it won't change over time (it bears the following warning "Note Behavior of this function, and therefore the results it returns, can change from release to release. It should not be used for canonical sorting applications"), Microsoft has an API that it uses to sort files in Explorer (and presumably other places).

```
internal static class Shlwapi
{
     // http://msdn.microsoft.com/
     // en-us/library/bb759947(VS.85).aspx
     [DllImport("shlwapi.dll",
                 CharSet = CharSet.Unicode)]
     public static extern int StrCmpLogicalW(
          string a,
          string b);
}

public sealed class
     NaturalStringComparer: IComparer<string>
{
     public int Compare(string a, string b)
     {
          return Shlwapi.StrCmpLogicalW(a, b);
     }
}
```

Summary

This chapter has covered the essential query functionality of filtering, ordering, and projecting the results into any resulting form you might require. Once you have understood and mastered these basic query essentials, you can confidently experiment with the more advanced query features offered by the other 40-odd standard query operators and begin writing your own operators if necessary.

INDEX

D

E

F

G–H

I

J

K–L

M

N

O

R

S

T

U–V

W

X–Y–Z