Microsoft

MCTS EXAM
70-516

Accessing Data with Microsoft® .NET Framework 4

Glenn Johnson

SELF-PACED

Training Kit

How to access your CD files

The print edition of this book includes a CD. To access the CD files, go to http://aka.ms/627390/files, and look for the Downloads tab.

Note: Use a desktop web browser, as files may not be accessible from all ereader devices.

Questions? Please contact: mspinput@microsoft.com

Microsoft Press

Exam 70-516: TS: Accessing Data with Microsoft .NET Framework 4

OBJECTIVE	CHAPTER	LESSON
MODELING DATA (20%)		
Map entities and relationships by using the Entity Data Model.	Chapter 6	Lesson 1
Map entities and relationships by using LINQ to SQL.	Chapter 4	Lesson 1
Create and customize entity objects.	Chapter 6	Lesson 1
Connect a POCO model to the entity Framework.	Chapter 6	Lesson 1
Create the database from the Entity Framework model.	Chapter 6	Lesson 1
Create model-defined functions.	Chapter 6	Lesson 1
MANAGING CONNECTIONS AND CONTEXT (18%)		
Configure connection strings and providers.	Chapter 2	Lesson 1
Create and manage a data connection.	Chapter 2	Lesson 1
Secure a connection.	Chapter 2	Lesson 1
Manage the DataContext and ObjectContext.	Chapter 4 Chapter 6	Lesson 1 Lesson 1
Implement eager loading.	Chapter 4 Chapter 6 Chapter 7	Lesson 1 Lesson 1 Lesson 1
Cache data.	Chapter 1 Chapter 4	Lesson 1 Lesson 3
Configure ADO.NET Data Services.	Chapter 7	Lesson 1, 2
QUERYING DATA (22%)		
Execute a SQL query.	Chapter 2	Lesson 2
Create a LINQ query.	Chapter 3 Chapter 4	Lesson 1, 2 Lesson 2
Create an Entity SQL (ESQL) query.	Chapter 3 Chapter 4 Chapter 6	Lesson 1, 2 Lesson 2 Lesson 2
Handle special data types.	Chapter 1 Chapter 2	Lesson 2 Lesson 2
Query XML.	Chapter 5	Lesson 1, 2, 3
Query data by using ADO.NET Data Services.	Chapter 7	Lesson 1
MANIPULATING DATA (22%)		
Create, update, or delete data by using SQL statements.	Chapter 2	Lesson 2
Create, update, or delete data by using DataContext.	Chapter 4	Lesson 3
Create, update, or delete data by using ObjectContext.	Chapter 6	Lesson 2
Manage transactions.	Chapter 2 Chapter 6	Lesson 3 Lesson 2
Create disconnected objects.	Chapter 1	Lesson 1

DEVELOPING AND DEPLOYING RELIABLE APPLICATIONS (18%)		
Monitor and collect performance data.	Chapter 8	Lesson 1
Handle exceptions.	Chapter 8	Lesson 2
Protect data.	Chapter 8	Lesson 3
Synchronize data.	Chapter 2 Chapter 8	Lesson 3 Lesson 4
Deploy ADO.NET components.	Chapter 9	Lesson 1

Exam Objectives The exam objectives listed here are current as of this book's publication date. Exam objectives are subject to change at any time without prior notice and at Microsoft's sole discretion. Please visit the Microsoft Learning Web site for the most current listing of exam objectives: http://www.microsoft.com/learning/en/us/Exam.aspx?ID=70-516.

Microsoft®

MCTS Self-Paced Training Kit (Exam 70-516): Accessing Data with Microsoft® .NET Framework 4

Glenn Johnson

PUBLISHED BY
Microsoft Press
A Division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Library of Congress Control Number: 2011927329

ISBN: 978-0-7356-2739-0

Printed and bound in the United States of America.

Microsoft Press books are available through booksellers and distributors worldwide. If you need support related to this book, email Microsoft Press Book Support at mspinput@microsoft.com. Please tell us what you think of this book at http://www.microsoft.com/learning/booksurvey.

Acquisitions Editor: Martin Del Re
Developmental Editor: Karen Szall
Project Editor: Valerie Woolley
Editorial Production: nSight, Inc.
Technical Reviewer: Christophe Nasarre; Technical Review services provided by Content Master, a member of CM Group, Ltd.
Copyeditor: Kerin Forsyth
Indexer: Luci Haskins
Cover: Twist Creative • Seattle

This product is printed digitally on demand.

Contents at a Glance

Contents

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

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

www.microsoft.com/learning/booksurvey/

Introduction

This training kit is designed for developers who write or support applications that access data written in C# or Visual Basic using Visual Studio 2010 and the Microsoft .NET Framework 4.0 and who also plan to take the Microsoft Certified Technology Specialist (MCTS) exam 70-516. Before you begin using this kit, you must have a solid foundation-level understanding of Microsoft C# or Microsoft Visual Basic and be familiar with Visual Studio 2010.

The material covered in this training kit and on exam 70-516 relates to the data access technologies in ADO.NET 4.0 with Visual Studio 2010. The topics in this training kit cover what you need to know for the exam as described on the Skills Measured tab for the exam, which is available at *http://www.microsoft.com/learning/en/us/exam.aspx?ID=70-516&locale=en-us#tab2*.

By using this training kit, you will learn how to do the following:

- Work with the ADO.NET disconnected classes
- Work with the ADO.NET connection classes
- Write and execute LINQ queries
- Implement LINQ to SQL classes
- Implement LINQ to XML in your applications
- Implement the ADO.NET Entity Framework in your applications
- Create and Implement WCF Data Service applications
- Monitor and Collect ADO.NET performance data
- Synchronize offline data
- Deploy Data Access applications

Refer to the objective mapping page in the front of this book to see where in the book each exam objective is covered.

System Requirements

The following are the minimum system requirements your computer needs to meet to complete the practice exercises in this book and to run the companion CD. To minimize the time and expense of configuring a physical computer for this training kit, it's recommended, but not required, that you use a virtualized environment, which will allow you to work in a sandboxed environment. This will let you make changes without worrying about

your day-to-day environment. Virtualization software is available from Microsoft (Virtual PC, Virtual Server, and Hyper-V) and other suppliers such as VMware (VMware Workstation) and Oracle (VirtualBox).

Hardware Requirements

Your computer should meet the following minimum hardware requirements:

- 2.0 GB of RAM (more is recommended)
- 80 GB of available hard disk space
- DVD-ROM drive
- Internet connectivity

Software Requirements

The following software is required to complete the practice exercises:

- Windows 7. You can download an Evaluation Edition of Windows 7 at the Microsoft Download Center at *http://technet.microsoft.com/en-us/evalcenter/cc442495*.
- SQL Server 2008 Developer Edition is recommended because some labs and sample code use this edition for permanently mounted databases. An Evaluation Edition is available from *http://msdn.microsoft.com/en-us/evalcenter/bb851668.aspx*.
- SQL Server 2008 Express Edition is recommended because some labs and sample code use this edition for User Instance mounted databases. A full release is available from *http://www.microsoft.com/express/Database*.

NOTE **SQL SERVER INSTALLATION**

If you are using a 64-bit OS, you should install 64-bit SQL Server before installing Visual Studio 2010. Visual Studio 2010 includes, and attempts to install, the 32-bit SQL Server 2008 Express Edition.

If you install the 64-bit versions of SQL Server first, the Visual Studio 2010 installer will see that SQL Server Express Edition is already installed and will skip over installing the 32-bit SQL Server 2008 Express Edition.

- Visual Studio 2010. You can download an evaluation edition from *http://msdn.microsoft.com/en-us/evalcenter/default*. Although the labs and code samples were generated using Visual Studio 2010 Premium Edition, you can use the Express Edition of Visual Studio for many of the labs, which is available from *http://www.microsoft.com/express*.

Code Samples

The code samples are provided in Visual C# and Visual Basic. You will find a folder for each chapter that contains CS (C#) and VB (Visual Basic) code. In these folders, you will find the sample code solution and a folder for each lesson that contains the practice code. The Practice Code folder contains Begin and Completed folders, so you can choose to start at the beginning and work through the practice or you can run the completed solution.

Using the CD

A companion CD is included with this training kit. The companion CD contains the following:

- **Practice tests** You can reinforce your understanding of the topics covered in this training kit by using electronic practice tests that you customize to meet your needs. You can run a practice test that is generated from the pool of Lesson Review questions in this book. Alternatively, you can practice for the 70-516 certification exam by using tests created from a pool of over 200 realistic exam questions, which give you many practice exams to ensure that you are prepared.
- **Code Samples** All of the Visual Basic and C# code you see in the book you will also find on the CD.
- **An eBook** An electronic version (eBook) of this book is included for when you do not want to carry the printed book with you.

Companion Content for Digital Book Readers: If you bought a digital edition of this book, you can enjoy select content from the print edition's companion CD.
Visit *http://www.microsoftpressstore.com/title/9780735627390* to get your downloadable content. This content is always up-to-date and available to all readers.

How to Install the Practice Tests

To install the practice test software from the companion CD to your hard disk, perform the following steps:

1. Insert the companion CD into your CD drive and accept the license agreement. A CD menu appears.

> ***NOTE*** **IF THE CD MENU DOES NOT APPEAR**
>
> **If the CD menu or the license agreement does not appear, AutoRun might be disabled on your computer. Refer to the Readme.txt file on the CD for alternate installation instructions.**

2. Click Practice Tests and follow the instructions on the screen.

How to Use the Practice Tests

To start the practice test software, follow these steps:

1. Click Start, All Programs, and then select Microsoft Press Training Kit Exam Prep.

   A window appears that shows all the Microsoft Press training kit exam prep suites installed on your computer.

2. Double-click the lesson review or practice test you want to use.

> ***NOTE*** **LESSON REVIEWS VS. PRACTICE TESTS**
>
> **Select the (70-516): Accessing Data with Microsoft .NET Framework 4 lesson review to use the questions from the "Lesson Review" sections of this book. Select the (70-516): Accessing Data with Microsoft .NET Framework 4 practice test to use a pool of questions similar to those that appear on the 70-516 certification exam.**

Lesson Review Options

When you start a lesson review, the Custom Mode dialog box appears so that you can configure your test. You can click OK to accept the defaults, or you can customize the number of questions you want, how the practice test software works, which exam objectives you want the questions to relate to, and whether you want your lesson review to be timed. If you are retaking a test, you can select whether you want to see all the questions again or only the questions you missed or did not answer.

After you click OK, your lesson review starts. The following list explains the main options you have for taking the test:

- To take the test, answer the questions and use the Next and Previous buttons to move from question to question.
- After you answer an individual question, if you want to see which answers are correct—along with an explanation of each correct answer—click Explanation.

- If you prefer to wait until the end of the test to see how you did, answer all the questions and then click Score Test. You will see a summary of the exam objectives you chose and the percentage of questions you got right overall and per objective. You can print a copy of your test, review your answers, or retake the test.

Practice Test Options

When you start a practice test, you choose whether to take the test in Certification Mode, Study Mode, or Custom Mode:

- **Certification Mode** Closely resembles the experience of taking a certification exam. The test has a set number of questions. It is timed, and you cannot pause and restart the timer.
- **Study Mode** Creates an untimed test during which you can review the correct answers and the explanations after you answer each question.
- **Custom Mode** Gives you full control over the test options so that you can customize them as you like.

In all modes, the user interface when you are taking the test is basically the same but with different options enabled or disabled depending on the mode. The main options are discussed in the previous section, "Lesson Review Options."

When you review your answer to an individual practice test question, a "References" section is provided that lists where in the training kit you can find the information that relates to that question and provides links to other sources of information. After you click Test Results to score your entire practice test, you can click the Learning Plan tab to see a list of references for every objective.

How to Uninstall the Practice Tests

To uninstall the practice test software for a training kit, use the Program And Features option in Windows Control Panel.

Acknowledgments

The author's name appears on the cover of a book, but I am only one member of a much larger team. Thanks very much to Valerie Woolley, Christophe Nasarre, and Karen Szall for working with me, being patient with me, and making this a great book. Christophe Nasarre was my technical reviewer, and he was far more committed to the project than any reviewer I've worked with in the past. I certainly could not have completed this book without his help.

Each of these editors contributed significantly to this book and I hope to work with them all in the future.

And a special thanks to Kristy Saunders for writing all of the practice test questions for the practice test located on the CD.

Support & Feedback

The following sections provide information on errata, book support, feedback, and contact information.

Errata

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

1. Go to *www.microsoftpressstore.com*.
2. In the Search box, enter the book's ISBN or title.
3. Select your book from the search results.
4. On your book's catalog page, under the cover image, you will see a list of links.
5. Click View/Submit Errata.

You will find additional information and services for your book on its catalog page. If you need additional support, please send an email message to Microsoft Press Book Support at *mspinput@microsoft.com*.

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

We Want to Hear from You

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

http://www.microsoft.com/learning/booksurvey

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

Stay in Touch

Let us keep the conversation going! We are on Twitter: *http://twitter.com/MicrosoftPress*.

Preparing for the Exam

Microsoft certification exams are a great way to build your resume and let the world know about your level of expertise. Certification exams validate your on-the-job experience and product knowledge. While there is no substitution for on-the-job experience, preparation through study and hands-on practice can help you prepare for the exam. We recommend that you round out your exam preparation plan by using a combination of available study materials and courses. For example, you might use the Training kit and another study guide for your "at home" preparation, and take a Microsoft Official Curriculum course for the classroom experience. Choose the combination that you think works best for you.

Note that this Training Kit is based on publicly available information about the exam and the author's experience. To safeguard the integrity of the exam, authors do not have access to the live exam.

Microsoft®
CERTIFIED
Technology Specialist

CHAPTER 3

Introducing LINQ

There always seems to be problems when it comes to moving data between the database and the client application. One of the problems stems from the differences in data types at both locations; another big problem is the handling of null values at each location. Microsoft calls this an impedance mismatch.

Yet another problem stems from passing commands to the database as strings. In your application, these strings compile as long as the quote is at each end of the string. If the string contains a reference to an unknown database object, or even a syntax error, the database server will throw an exception at run time instead of at compile time.

LINQ stands for Language Integrated Query. LINQ is the Microsoft solution to these problems, which LINQ solves by providing querying capabilities to the database, using statements that are built into LINQ-enabled languages such as Microsoft C# and Visual Basic 2010. In addition, these querying capabilities enable you to query almost any collection.

Exam objectives in this chapter:

- Create a LINQ query.

Lessons in this chapter:

Before You Begin

You must have some understanding of C# or Visual Basic 2010. This chapter requires only the hardware and software listed at the beginning of this book.

REAL WORLD

Glenn Johnson

There are many scenarios in which your application has collections of objects that you want to query, filter, and sort. In many cases, you might need to return the results of these queries as a collection of different objects that contain only the information needed—for instance, populating a grid on the user interface with a subset of the data. LINQ really simplifies these query problems, providing an elegant, language-specific solution.

Lesson 1: Understanding LINQ

This lesson starts by providing an example of a LINQ expression so you can see what a LINQ expression looks like, and some of the basic syntax is covered as well. This lesson also shows you the Microsoft .NET Framework features that were added to make LINQ work. The features can be used individually, which can be useful in many scenarios.

After this lesson, you will be able to:

- Use object initializers.
- Implement implicitly typed local variables.
- Create anonymous types.
- Create lambda expressions.
- Implement extension methods.
- Understand the use of query extension methods.

Estimated lesson time: 60 minutes

A LINQ Example

LINQ enables you to perform query, set, and transform operations within your programming language syntax. It works with any collection that implements the *IEnumerable* or the generic *IEnumerable<T>* interface, so you can run LINQ queries on relational data, XML data, and plain old collections.

So what can you do with LINQ? The following is a simple LINQ query that returns the list of colors that begin with the letter B, sorted.

Sample of Visual Basic Code

```
Dim colors() =
{
    "Red",
    "Brown",
    "Orange",
    "Yellow",
    "Black",
    "Green",
    "White",
    "Violet",
    "Blue"
}

Dim results as IEnumerable(Of String)=From c In colors _
            Where c.StartsWith("B") _
            Order By c _
            Select c
```

Sample of C# Code

```
string[] colors =
{
    "Red",
    "Brown",
    "Orange",
    "Yellow",
    "Black",
    "Green",
    "White",
    "Violet",
    "Blue"
};

IEnumerable<string> results = from c in colors
              where c.StartsWith("B")
              orderby c
              select c;
```

The first statement in this example uses array initializer syntax to create an array of strings, populated with various colors, followed by the LINQ expression. Focusing on the right side of the equals sign, you see that the LINQ expression resembles a SQL statement, but the *from* clause is at the beginning, and the *select* clause is at the end.

You're wondering why Microsoft would do such a thing. SQL has been around for a long time, so why didn't Microsoft keep the same format SQL has? The reason for the change is that Microsoft could not provide IntelliSense if it kept the existing SQL command layout. By moving the *from* clause to the beginning, you start by naming the source of your data, and Visual Studio .NET can use that information to provide IntelliSense through the rest of the statement.

In the example code, you might be mentally equating the "from c in colors" with a *For Each* (*C# foreach*) loop; it is. Notice that *c* is the *loop* variable that references one item in the source collection for each of the iterations of the loop. What is *c*'s type and where is *c* declared? The variable called *c* is declared in the *from* clause, and its type is implicitly set to *string*, based on the source collection as an array of *string*. If your source collection is *ArrayList*, which is a collection of objects, *c*'s type would be implicitly set to *object*, even if *ArrayList* contained only strings. Like the *For Each* loop, you can set the type for *c* explicitly as well, and each element will be cast to that type when being assigned to *c* as follows:

Sample of Visual Basic Code

```
From c As String In colors _
```

Sample of C# Code

```
from string c in colors
```

In this example, the source collection is typed, meaning it is an array of *string*, so there is no need to specify the type for *c* because the compiler can figure this out.

The *from* clause produces a generic *IEnumerable* object, which feeds into the next part of the LINQ statement, the *where* clause. Internally, imagine the *where* clause has code to iterate over the values passed into the *where* clause and output only the values that meet the specified criteria. The *where* clause also produces a generic *IEnumerable* object but contains logic to filter, and it is passed to the next part of the LINQ statement, the *order by* clause.

The *order by* clause accepts a generic *IEnumerable* object and sorts based on the criteria. Its output is also a generic *IOrderedEnumerable* object but contains the logic to sort and is passed to the last part of the LINQ statement, the *select* clause.

The *select* clause must always be the last part of any LINQ expression. This is when you can decide to return (select) the entire object with which you started (in this case, *c*) or something different. When selecting *c*, you are returning the whole string. In a traditional SQL statement, you might select * or select just the columns you need to get a subset of the data. You might select *c.SubString(0,2)* to return the first two characters of each of the colors to get a subset of the data or create a totally different object that is based on the string *c*.

Deferred Execution

A LINQ query is a generic *IEnumerable* object of what you select. The variable to which this result is assigned is known as the *range variable*. This is not a populated collection; it's merely a query object that can retrieve data. LINQ doesn't access the source data until you try to use the query object to work with the results. This is known as *deferred execution*.

EXAM TIP

For the exam, understand what deferred execution is because you can expect LINQ questions related to this topic.

The generic *IEnumerable* interface has only one method, *GetEnumerator*, that returns an object that implements the generic *IEnumerator* interface. The generic *IEnumerator* interface has a *Current* property, which references the current item in the collection, and two methods, *MoveNext* and *Reset*. *MoveNext* moves to the next element in the collection. *Reset* moves the iterator to its initial position—that is, before the first element in the collection.

The data source is not touched until you iterate on the query object, but if you iterate on the query object multiple times, you access the data source each time. For example, in the LINQ code example, a variable called *results* was created to retrieve all colors that start with B, sorted. In this example, code is added to loop over the *results* variable and display each color. Black in the original source data is changed to Slate. Then code is added to loop over the results again and display each color.

Sample of Visual Basic Code

```
For Each Color As String In results
    txtLog.AppendText(Color + Environment.NewLine)
Next
```

```
colors(4) = "Slate"

txtLog.AppendText("---------" + Environment.NewLine)
For Each Color As String In results
    txtLog.AppendText(Color + Environment.NewLine)
Next
```

Sample of C# Code

```
foreach (var color in results)
{
    txtLog.AppendText(color + Environment.NewLine);
}

colors[4] = "Slate";

txtLog.AppendText("---------" + Environment.NewLine);
foreach (var color in results)
{
    txtLog.AppendText(color + Environment.NewLine);
}
```

The second time the *results* variable was used, it displayed only Blue and Brown, not the original three matching colors (Black, Blue, and Brown), because the query was re-executed for the second loop on the updated collection in which Black was replaced by a color that does not match the query criteria. Whenever you use the *results* variable to loop over the results, the query re-executes on the same data source that might have been changed and, therefore, might return updated data.

You might be thinking of how that effects the performance of your application. Certainly, performance is something that must be considered when developing an application. However, you might be seeing the benefit of retrieving up-to-date data, which is the purpose of deferred execution. For users who don't want to re-run the query every time, use the resulting query object to produce a generic list immediately that you can use repeatedly afterward. The following code sample shows how to create a frozen list.

Sample of Visual Basic Code

```
Dim results As List(Of String) = (From c In colors _
            Where c.StartsWith("B") _
            Order By c _
            Select c).ToList()

For Each Color As String In results
    txtLog.AppendText(Color + Environment.NewLine)
Next

colors(4) = "Slate"

txtLog.AppendText("---------" + Environment.NewLine)
For Each Color As String In results
    txtLog.AppendText(Color + Environment.NewLine)
Next
```

Sample of C# Code

```
List<string> results = (from string c in colors
                        where c.StartsWith("B")
                        orderby c
                        select c).ToList();

foreach (var color in results)
{
    txtLog.AppendText(color + Environment.NewLine);
}

colors[4] = "Slate";

txtLog.AppendText("---------" + Environment.NewLine);
foreach (var color in results)
{
    txtLog.AppendText(color + Environment.NewLine);
}
```

In this code example, the result of the LINQ query is frozen by wrapping the expression in parentheses and adding the *ToList()* call to the end. The *results* variable now has a type of *List Of String*. The *ToList* method causes the query to execute and put the results into the variable called *results*; then the *results* collection can be used again without re-executing the LINQ expression.

LINQ Providers

In the previous examples, you can see that LINQ works with .NET Framework objects because the .NET Framework comes with a *LINQ to Objects* provider. Figure 3-1 shows the LINQ providers that are built into the .NET Framework. Each LINQ provider implements features focused on the data source.

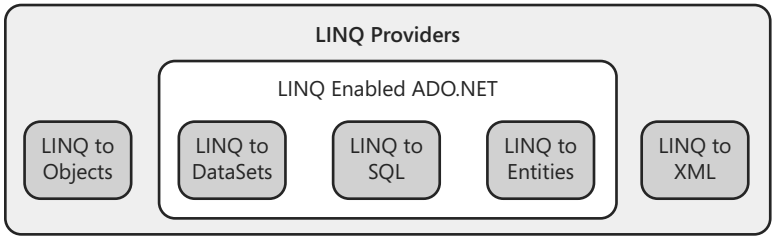

FIGURE 3-1 This figure shows the LINQ providers built into the .NET Framework.

The LINQ provider works as a middle tier between the data store and the language environment. In addition to the LINQ providers included in the .NET Framework, there are many third-party LINQ providers. To create a LINQ provider, you must implement the *IQueryable* interface. This has a *Provider* property whose type is *IQueryProvider*, which is called to initialize and execute LINQ expressions.

Features That Make Up LINQ

Now that you've seen a LINQ expression, it's time to see what was added to the .NET Framework to create LINQ. Each of these features can be used by itself, but all of them are required to create LINQ.

Object Initializers

You can use object initializers to initialize any or all of an object's properties in the same statement that instantiates the object, but you're not forced to write custom constructors.

You might have seen classes that have many constructors because the developer was trying to provide a simple way to instantiate and initialize the object. To understand this, consider the following code example of a *Car* class that has five automatic properties but doesn't have any custom constructors.

Sample of Visual Basic Code

```
Public Class Car
    Public Property VIN() As String
    Public Property Make() As String
    Public Property Model() As String
    Public Property Year() As Integer
    Public Property Color() As String
End Class
```

Sample of C# Code

```
public class Car
{
    public string VIN { get; set; }
    public string Make { get; set; }
    public string Model { get; set; }
    public int Year { get; set; }
    public string Color { get; set; }
}
```

To instantiate a *Car* object and populate the properties with data, you might do something like the following:

Sample of Visual Basic Code

```
Dim c As New Car()
c.VIN = "ABC123"
c.Make = "Ford"
c.Model = "F-250"
c.Year = 2000
```

Sample of C# Code

```
Car c = new Car();
c.VIN = "ABC123";
c.Make = "Ford";
c.Model = "F-250";
c.Year = 2000;
```

It took five statements to create and initialize the object, and *Color* wasn't initialized. If a constructor was provided, you could instantiate the object and implicitly initialize the properties with one statement, but what would you do if someone wanted to pass only three parameters? How about passing five parameters? Do you create constructors for every combination of parameters you might want to pass? The answer is to use object initializers.

By using object initializers, you can instantiate the object and initialize any combination of properties with one statement, as shown in the following example:

Sample of Visual Basic Code

```
Dim c As New Car() With {.VIN = "ABC123", .Make = "Ford", _
                         .Model = "F-250", .Year = 2000}
```

Sample of C# Code

```
Car c = new Car() { VIN = "ABC123", Make = "Ford",
                    Model = "F-250", Year = 2000 };
```

Conceptually, the *Car* object is being instantiated, and the default constructor generated by the compiler is executed. Each of the properties will be initialized with its default value. If you are using a parameterless constructor, as shown, the parentheses are optional, but if you are executing a constructor that requires parameters, the parentheses are mandatory.

Collection initializers are another form of object initializer, but for collections. Collection initializers have existed for arrays since the first release of the .NET Framework, but you can now initialize collections such as *ArrayList* and generic *List* using the same syntax. The following example populates a collection of cars, using both object initializers and collection initializers.

Sample of Visual Basic Code

```
Private Function GetCars() As List(Of Car)
    Return New List(Of Car) From
    {
        New Car() With {.VIN = "ABC123", .Make = "Ford",
                        .Model = "F-250", .Year = 2000},
        New Car() With {.VIN = "DEF123", .Make = "BMW",
                        .Model = "Z-3", .Year = 2005},
        New Car() With {.VIN = "ABC456", .Make = "Audi",
                        .Model = "TT", .Year = 2008},
        New Car() With {.VIN = "HIJ123", .Make = "VW",
                        .Model = "Bug", .Year = 1956},
        New Car() With {.VIN = "DEF456", .Make = "Ford",
                        .Model = "F-150", .Year = 1998}
    }
End Function
```

Sample of C# Code

```
private List<Car> GetCars()
{
    return new List<Car>
    {
        new Car {VIN = "ABC123",Make = "Ford",
```

```
                Model = "F-250", Year = 2000},
        new Car {VIN = "DEF123",Make = "BMW",
                Model = "Z-3",   Year = 2005},
        new Car {VIN = "ABC456",Make = "Audi",
                Model = "TT",    Year = 2008},
        new Car {VIN = "HIJ123",Make = "VW",
                Model = "Bug",   Year = 1956},
        new Car {VIN = "DEF456",Make = "Ford",
                Model = "F-150", Year = 1998}
    };
}
```

The code example creates a generic *List* object and populates the list with five cars, all in one statement. No variables are needed to set the properties of each car because it's being initialized.

How are object initializers used in LINQ? They enable you to create some types of projections in LINQ. A *projection* is a shaping or transformation of the data in a LINQ query to produce what you need in the output with the *select* statement instead of including just the whole source object(s). For example, if you want to write a LINQ query that will search a color list for all the color names that are five characters long, sorted by the matching color, instead of returning an *IEnumerable* object of *string*, you might use object initializers to return an *IEnumerable* object of *Car* in which the car's color is set to the matching color, as shown here:

Sample of Visual Basic Code

```
Dim colors() =
    {
        "Red",
        "Brown",
        "Orange",
        "Yellow",
        "Black",
        "Green",
        "White",
        "Violet",
        "Blue"
    }

Dim fords As IEnumerable(Of Car) = From c In colors
                                   Where c.Length = 5
                                   Order By c
                                   Select New Car() With
                                          {.Make = "Ford",
                                           .Color = c}

For Each car As Car In fords
    txtLog.AppendText(String.Format("Car: Make:{0} Color:{1}" _
                    & Environment.NewLine, car.Make, car.Color))
Next
```

Sample of C# Code

```
string[] colors =
{
```

```
    "Red",
    "Brown",
    "Orange",
    "Yellow",
    "Black",
    "Green",
    "White",
    "Violet",
    "Blue"
};

IEnumerable<Car> fords = from c in colors
                         where c.Length == 5
                         orderby c
                         select new Car()
                               {
                                   Make = "Ford",
                                   Color = c
                               };

foreach (Car car in fords)
{
    txtLog.AppendText(String.Format("Car: Make:{0} Color:{1}"
                    + Environment.NewLine, car.Make, car.Color));
}
```

The whole idea behind this example is that you want to construct a collection of cars, but each car will have a color from the collection of colors that matches the five-letter-long criterion. The *select* clause creates the *Car* object and initializes its properties. The *select* clause cannot contain multiple statements. Without object initializers, you wouldn't be able to instantiate and initialize the *Car* object without first writing a constructor for the *Car* class that takes *Make* and *Color* parameters.

Implicit Typed Local Variable Declarations

Doesn't it seem like a chore to declare a variable as a specific type and then instantiate that type in one statement? You have to specify the type twice, as shown in the following example:

Sample of Visual Basic Code

```
Dim c as Car = New Car( )
```

Sample of C# Code

```
Car c = new Car( )
```

Visual Basic users might shout, "Hey, Visual Basic doesn't require me to specify the type twice!" But what about the example in which you make a call to a method that returns a generic *List Of Car*, but you still have to specify the type for the variable that receives the collection, as shown here?

Sample of Visual Basic Code

```
Dim cars As List(Of Car) = GetCars()
```

Sample of C# Code

```
List<Car> cars = GetCars();
```

In this example, would it be better if you could ask the compiler what the type is for this variable called *cars*? You can, as shown in this code example:

Sample of Visual Basic Code

```
Dim cars = GetCars()
```

Sample of C# Code

```
var cars = GetCars();
```

Instead of providing the type for your variable, you're asking the compiler what the type is, based on whatever is on the right side of the equals sign. That means there must be an equals sign in the declaration statement, and the right side must evaluate to a typed expression. You cannot have a null constant on the right side of the equals sign because the compiler would not know what the type should be.

If you're wondering whether this is the same as the older variant type that existed in earlier Visual Basic, the answer is no. As soon as the compiler figures out what the type should be, you can't change it. Therefore, you get IntelliSense as though you explicitly declared the variable's type.

Here are the rules for implicitly typed local variables:

- They can be implemented on local variables only.
- The declaration statement must have an equals sign with a non-null assignment.
- They cannot be implemented on method parameters.

Implicitly typed local variables can be passed to methods, but the method's parameter type must match the actual type that was inferred by the compiler.

Do you really need this feature? Based on this explanation, you can see that this feature is simply an optional way to save a bit of typing. You might also be wondering why implicitly typed local variables are required for LINQ: This feature is required to support anonymous types, which are used in LINQ.

Anonymous Types

Often, you want to group together some data in a somewhat temporary fashion. That is, you want to have a grouping of data, but you don't want to create a new type just for something that might be used in one method. To understand the problem that anonymous types solves, imagine that you are writing the graphical user interface (GUI) for your application, and a collection of cars is passed to you in which each car has many properties. If you bind the collection directly to a data grid, you'll see all the properties, but you needed to display only two of the properties, so that automatic binding is displaying more data than you want. This is when anonymous types can be used.

In the following code sample, you want to create an anonymous type that contains only *Make* and *Model* properties because these are the only properties you need.

Sample of Visual Basic Code

```
Dim x = New With {.Make = "VW", .Model = "Bug"}
txtLog.AppendText(x.Make & ", " & x.Model)
```

Sample of C# Code

```
var x = new {Make = "VW", Model = "Bug"};
txtLog.AppendText(x.Make + ", " + x.Model);
```

If you type in this code, you see that in the second statement, when x is typed and you press the period key, the IntelliSense window is displayed and you see *Make* and *Model* as available selections. The variable called *x* is implicitly typed because you simply don't know what the name of the anonymous type is. The compiler, however, does know the name of the anonymous type. This is the benefit of getting IntelliSense for scenarios in which implicitly typed local variables are required.

Anonymous types are used in LINQ to provide projections. You might make a call to a method that returns a list of cars, but from that list, you want to run a LINQ query that retrieves the VIN as one property and make and year combined into a different property for displaying in a grid. Anonymous types help, as shown in this example:

Sample of Visual Basic Code

```
Dim carData = From c In GetCars()
    Where c.Year >= 2000
    Order By c.Year
    Select New With
                {
                    c.VIN,
                    .MakeAndModel = c.Make + " " + c.Model
                }

dgResults.DataSource = carData.ToList()
```

Sample of C# Code

```
var carData = from c in GetCars()
              where c.Year >= 2000
              orderby c.Year
              select new
              {
                  c.VIN,
                  MakeAndModel = c.Make + " " + c.Model
              };

dgResults.DataSource = carData.ToList();
```

When this example is executed, the LINQ query will locate all the cars that have a *Year* property equal to or greater than 2000. This will result in finding three cars that are sorted by year. That result is then projected to an anonymous type that grabs the VIN and combines the

make and model into a new property called *MakeAndModel*. Finally, the result is displayed in the grid, as shown in Figure 3-2.

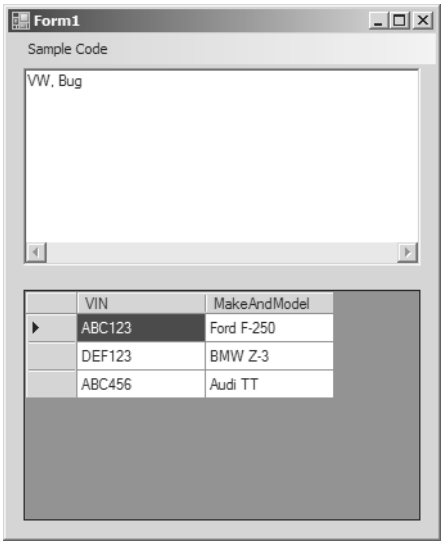

FIGURE 3-2 Anonymous types are displayed in the grid.

Lambda Expressions

Lambda expressions can be used anywhere delegates are required. They are very much like anonymous methods but have a much abbreviated syntax that makes them easy to use inline.

Consider the generic *List* class that has a *Find* method. This method accepts a generic *Predicate* delegate as a parameter. If you look up the generic *Predicate* delegate type, you find that this delegate is a reference to a method that accepts a single parameter of type *T* and returns a Boolean value. The *Find* method has code that loops though the list and, for each item, the code executes the method referenced through the *Predicate* parameter, passing the item to the method and receiving a response that indicates found or not found. Here is an example of using the *Find* method with a *Predicate* delegate:

Sample of Visual Basic Code

```
Dim yearToFind As Integer

Private Sub PredecateDelegateToolStripMenuItem_Click( _
            ByVal sender As System.Object, _
            ByVal e As System.EventArgs) _
        Handles PredecateDelegateToolStripMenuItem.Click

    yearToFind = 2000
    Dim cars = GetCars()
    Dim found = cars.Find(AddressOf ByYear)
    txtLog.AppendText(String.Format( _
            "Car VIN:{0} Make:{1} Year:{2}" & Environment.NewLine, _
```

```
            found.VIN, found.Make, found.Year))
End Sub

Private Function ByYear(ByVal c As Car) As Boolean
    Return c.Year = yearToFind
End Function
```

Sample of C# Code

```
int yearToFind = 2000;

private void predecateDelegateToolStripMenuItem_Click(
    object sender, EventArgs e)
{
    var cars = GetCars();
    yearToFind = 2000;
    var found = cars.Find(ByYear);
    txtLog.AppendText(string.Format(
        "Car VIN:{0} Make:{1} Year{2}" + Environment.NewLine,
        found.VIN, found.Make, found.Year));
}

private bool ByYear(Car c)
{
    return c.Year == yearToFind;
}
```

In this example, the *yearToFind* variable is defined at the class level to make it accessible to both methods. That's typically not desirable because *yearToFind* is more like a parameter that needs to be passed to the *ByYear* method. The problem is that the *Predicate* delegate accepts only one parameter, and that parameter has to be the same type as the list's type. Another problem with this code is that a separate method was created just to do the search. It would be better if a method wasn't required.

The previous example can be rewritten to use a lambda expression, as shown in the following code sample:

Sample of Visual Basic Code

```
Private Sub LambdaExpressionsToolStripMenuItem_Click( _
        ByVal sender As System.Object, _
        ByVal e As System.EventArgs) _
        Handles LambdaExpressionsToolStripMenuItem.Click
    Dim cars = GetCars()
    Dim theYear = 2000
    Dim found = cars.Find(Function(c) c.Year = theYear)
    txtLog.AppendText(String.Format( _
            "Car VIN:{0} Make:{1} Year:{2}" & Environment.NewLine, _
            found.VIN, found.Make, found.Year))
End Sub
```

Sample of C# Code

```
private void lambdaExpressionsToolStripMenuItem_Click(
    object sender, EventArgs e)
{
```

```
    var cars = GetCars();
    var theYear = 2000;
    var found = cars.Find(c => c.Year== theYear);
    txtLog.AppendText(string.Format(
        "Car VIN:{0} Make:{1} Year{2}" + Environment.NewLine,
        found.VIN, found.Make, found.Year));
}
```

You can think of the lambda expression as inline method. The left part declares the parameters, comma delimited. After the => sign, you have the expression. In this example, the lambda expression is supplied inline with the *Find* method. This eliminates the need for a separate method. Also, the variable called *theYear* is defined as a local variable in the enclosing method, so it's accessible to the lambda expression.

Formally, a lambda expression is an expression that has one input and contains only a single statement that is evaluated to provide a single return value; however, in the .NET Framework, multiple parameters can be passed to a lambda expression, and multiple statements can be placed into a lambda expression. The following example shows the multi-statement lambda expression syntax.

Sample of Visual Basic Code

```
Dim found = cars.Find(Function(c As Car)
                          Dim x As Integer
                          x = theYear
                          Return c.Year = x
                      End Function)
```

Sample of C# Code

```
var found = cars.Find(c =>
{
    int x;
    x = theYear;
    return c.Year == x;
});
```

In C#, if the lambda expression takes multiple parameters, the parameters must be surrounded by parentheses, and if the lambda expression takes no parameters, you must provide an empty set of parentheses where the parameter would go.

How are lambda expressions used with LINQ? When you type your LINQ query, behind the scenes, parts of it will be converted into a tree of lambda expressions. Also, you must use lambda expressions with query extension methods, which are covered right after extension methods, which follows.

Extension Methods

Extension methods enable you to add methods to a type, even if you don't have the source code for the type.

To understand why you might want this, think of this simple scenario: You want to add an *IsNumeric* method to the *string* class, but you don't have the source code for the *string* class. What would you do?

One solution is to create a new class that inherits from *string*, maybe called *MyString*, and then add your *IsNumeric* method to this class. This solution has two problems. First, the *string* class is *sealed*, which means that you can't inherit from *string*. Even if you could inherit from *string* to create your custom class, you would need to make sure that everyone uses it and not the *string* class that's built in. You would also need to write code to convert strings you might receive when you make calls outside your application into your *MyString* class.

Another possible, and more viable, solution is to create a helper class, maybe called *StringHelper*, that contains all the methods you would like to add to the *string* class but can't. These methods would typically be static methods and take *string* as the first parameter. Here is an example of a *StringHelper* class that has the *IsNumeric* method:

Sample of Visual Basic Code

```
Public Module StringHelper
    Public Function IsNumeric(ByVal str As String) As Boolean
        Dim val As Double
        Return Double.TryParse(str, val)
    End Function
End Module
```

Sample of C# Code

```
public static class StringHelper
{
    public static bool IsNumeric(string str)
    {
        double val;
        return double.TryParse(str, out val);
    }
}
```

The following code uses the helper class to test a couple of strings to see whether they are numeric. The output will display *false* for the first call and *true* for the second call.

Sample of Visual Basic Code

```
Dim s As String = "abc123"
txtLog.AppendText(StringHelper.IsNumeric(s) & Environment.NewLine)
s = "123"
txtLog.AppendText(StringHelper.IsNumeric(s) & Environment.NewLine)
```

Sample of C# Code

```
string s = "abc123";
txtLog.AppendText(StringHelper.IsNumeric(s) + Environment.NewLine);
s = "123";
txtLog.AppendText(StringHelper.IsNumeric(s) + Environment.NewLine);
```

What's good about this solution is that the user doesn't need to instantiate a custom string class to use the *IsNumeric* method. What's bad about this solution is that the user needs to know that the helper class exists, and the syntax is clunky at best.

Prior to .NET Framework 3.5, the helper class solution was what most people implemented, so you will typically find lots of helper classes in an application, and, yes, you need to look for them and explore the helper classes so you know what's in them.

In .NET Framework 3.5, Microsoft introduced extension methods. By using extension methods, you can extend a type even when you don't have the source code for the type. In some respects, this is deceptive, but it works wonderfully, as you'll see.

In the previous scenario, another solution is to add the *IsNumeric* method to the *string* class by using an extension method, adding a public module (C# public *static* class) and creating public static methods in this class. In Visual Basic, you add the <*Extension()*> attribute before the method. In C#, you add the keyword *this* in front of the first parameter to indicate that you are extending the type of this parameter.

All your existing helper classes can be easily modified to become extension methods, but this doesn't break existing code. Here is the modified helper class, in which the *IsNumeric* method is now an extension method on *string*.

Sample of Visual Basic Code

```
Imports System.Runtime.CompilerServices

Public Module StringHelper
    <Extension()> _
    Public Function IsNumeric(ByVal str As String) As Boolean
        Dim val As Double
        Return Double.TryParse(str, val)
    End Function
End Module
```

Sample of C# Code

```
public static class StringHelper
{
    public static bool IsNumeric(this string str)
    {
        double val;
        return double.TryParse(str, out val);
    }
}
```

You can see in this code example that the changes to your helper class are minimal. Now that the *IsNumeric* method is on the *string* class, you can call the extension method as follows.

Sample of Visual Basic Code

```
Dim s As String = "abc123"
txtLog.AppendText(s.IsNumeric() & Environment.NewLine)
s = "123"
txtLog.AppendText(s.IsNumeric() & Environment.NewLine)
```

Sample of C# Code

```
string s = "abc123";
txtLog.AppendText(s.IsNumeric() + Environment.NewLine);
s = "123";
txtLog.AppendText(s.IsNumeric() + Environment.NewLine);
```

You can see that this is much cleaner syntax, but the helper class syntax still works, so you can convert your helper class methods to extension methods but you're not forced to call the helper methods explicitly. Because the compiler is not able to find *IsNumeric* in the *string* class, it is looking for the extension methods that extend *string* with the right name and the right signature. Behind the scenes, it is simply changing your nice syntax into calls to your helper methods when you build your application, so the clunky syntax is still there (in the compiled code), but you can't see it. Performance is exactly the same as well. The difference is that the IntelliSense window now shows you the extension methods on any *string*, as shown in Figure 3-3. The icon for the extension method is a bit different from the icon for a regular method. In fact, there are already extension methods on many framework types. In Figure 3-3, the method called *Last* is also an extension method.

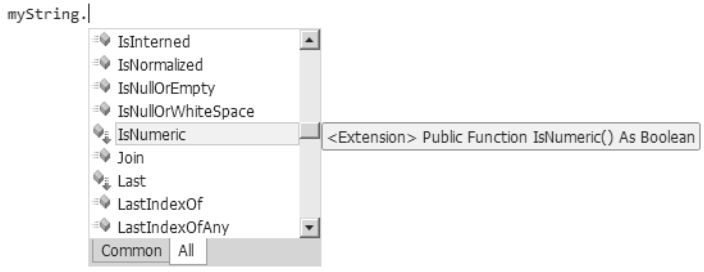

FIGURE 3-3 Extension methods for the *string* class are shown in the IntelliSense window.

In the previous code samples, did you notice that when a line needed to be appended in the *TextBox* class called *txtLog*, the string passed in is concatenated with *Environment.NewLine*? Wouldn't it be great if *TextBox* had a *WriteLine* method? Of course it would! In this example, a helper class called *TextBoxHelper* is added, as follows.

Sample of Visual Basic Code

```
Imports System.Runtime.CompilerServices

Public Module TextBoxHelper
    <Extension()> _
    Public Sub WriteLine(ByVal txt As TextBox, ByVal line As Object)
        txt.AppendText(line & Environment.NewLine)
    End Sub
End Module
```

Sample of C# Code

```
using System.Windows.Forms;

    public static class TextBoxHelper
```

```
{
    public static void WriteLine(this TextBox txt, object line)
    {
        txt.AppendText(line + Environment.NewLine);
    }
}
```

Using this new extension method on the *TextBox* class, you can change the code from the previous examples to use the *string* extension and the *TextBox* extension. This cleans up your code, as shown in the following example.

Sample of Visual Basic Code

```
Dim s As String = "abc123"
txtLog.WriteLine(s.IsNumeric())
s = "123"
txtLog.WriteLine(s.IsNumeric())
```

Sample of C# Code

```
string s = "abc123";
txtLog.WriteLine(s.IsNumeric());
s = "123";
txtLog.WriteLine(s.IsNumeric());
```

Here are some rules for working with extension methods:

- Extension methods must be defined in a Visual Basic module or C# static class.
- The Visual Basic module or C# static class must be *public*.
- If you define an extension method for a type, and the type already has the same method, the type's method is used and the extension method is ignored.
- In C#, the class and the extension methods must be *static*. Visual Basic modules and their methods are automatically static (Visual Basic *Shared*).
- The extension method works as long as it is in scope. This might require you to add an *imports* (C# *using*) statement for the namespace in which the extension method is to get access to the extension method.
- Although extension methods are implemented as *static* (Visual Basic *Shared*) methods, they are instance methods on the type you are extending. You cannot add static methods to a type with extension methods.

Query Extension Methods

Now that you've seen extension methods, you might be wondering why Microsoft needed extension methods to implement LINQ. To do so, Microsoft added extension methods to several types, but most important are the methods that were added to the generic *IEnumerable* interface. Extension methods can be added to any type, which is interesting when you think of adding concrete extension methods (methods that have code) to interfaces, which are abstract (can't have code).

Consider the following example code, in which an array of strings called *colors* is created and assigned to a variable whose type is *IEnumerable* of *string*.

Sample of Visual Basic Code

```
Dim colors() =
{
    "Red",
    "Brown",
    "Orange",
    "Yellow",
    "Black",
    "Green",
    "White",
    "Violet",
    "Blue"
}
Dim colorEnumerable As IEnumerable(Of String) = colors
```

Sample of C# Code

```
string[] colors =
{
    "Red",
    "Brown",
    "Orange",
    "Yellow",
    "Black",
    "Green",
    "White",
    "Violet",
    "Blue"
}
IEnumerable<string> colorEnumerable = colors;
```

Because the *colorEnumerable* variable's type is *IEnumerable* of *string*, when you type *colorEnumerable* and press the period key, the IntelliSense window is displayed and shows the list of methods available on the generic *IEnumerable* interface, as shown in Figure 3-4. These methods are known as *query extension methods*. In addition to the query extension methods that exist for *IEnumerable*, the generic *IEnumerable* interface also contains query extension methods.

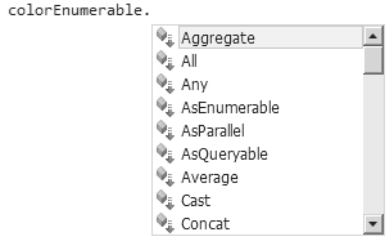

FIGURE 3-4 This figure shows *Extension* methods on the *IEnumerable* interface.

Some of these extension methods are mapped directly to Visual Basic and C# keywords used in LINQ (also known as LINQ operators). For example, you'll find *Where* and *OrderBy* extension methods that map directly to the *where* and *orderby* (Visual Basic *Order By*) keywords.

Microsoft added these extension methods to the *IEnumerable* interface by implementing the extension methods in a class called *Enumerable*, which is in the *System.Linq* namespace. This class is in an assembly called *System.Core.dll* to which most project templates already have a reference.

> ***NOTE*** **TO USE LINQ AND QUERY EXTENSION METHODS**
>
> **To use LINQ and query extension methods, your project must reference *System.Core.dll*, and, in your code, you must import (C# *using*) the *System.Linq* namespace.**

The *Enumerable* class also has three static methods you might find useful: *Empty*, *Range*, and *Repeat*. Here is a short description of each of these methods.

- **Empty** The generic *Empty* method produces a generic *IEnumerable* object with no elements.
- **Range** The *Range* method produces a counting sequence of integer elements. This can be useful when you want to join a counter to another element sequence that should be numbered. This method takes two parameters: the starting number and the count of how many times to increment. If you pass 100,5 to this method, it will produce 100, 101, 102, 103, 104.
- **Repeat** Use the generic *Repeat* method to produce an *IEnumerable<int>* object that has the same element repeated. This method accepts two parameters: *Element* and *Count*. The *element* parameter is the element to be repeated, and *count* specifies how many times to repeat the element.

The next section covers many of the query extension methods that are implemented on the *Enumerable* class to extend the generic *IEnumerable* interface.

ALL

The *All* method returns *true* when all the elements in the collection meet a specified criterion and returns *false* otherwise. Here is an example of checking whether all the cars returned from the *GetCars* method call have a year greater than 1960.

Sample of Visual Basic Code

```
txtLog.WriteLine(GetCars().All(Function(c) c.Year > 1960))
```

Sample of C# Code

```
txtLog.WriteLine(GetCars().All(c => c.Year > 1960));
```

ANY

The *Any* method returns *true* when at least one element in the collection meets the specified criterion and returns *false* otherwise. The following example checks whether there is a car that has a year of 1960 or earlier.

Sample of Visual Basic Code

```
txtLog.WriteLine(GetCars().Any(Function(c) c.Year <= 1960))
```

Sample of C# Code

```
txtLog.WriteLine(GetCars().Any(c => c.Year <= 1960));
```

ASENUMERABLE

To explain the *AsEnumerable* method, remember the rule mentioned earlier in this chapter for extension methods:

- If you define an extension method for a type, and the type already has the same method, the type's method will be used, and the extension method is ignored.

Use the *AsEnumerable* method when you want to convert a collection that implements the generic *IEnumerable* interface but is currently cast as a different type, such as *IQueryable*, to the generic *IEnumerable*. This can be desirable when the type you are currently casting has a concrete implementation of one of the extension methods you would prefer to get called.

For example, the *Table* class that represents a database table could have a *Where* method that takes the predicate argument and executes a SQL query to the remote database. If you don't want to execute a call to the database remotely, you can use the *AsEnumerable* method to cast to *IEnumerable* and execute the corresponding *Where* extension method.

In this example, a class called *MyStringList* inherits from *List Of String*. This class has a single *Where* method whose method signature matches the *Where* method on the generic *IEnumerable* interface. The code in the *Where* method of the *MyStringList* class is returning all elements but, based on the predicate that's passed in, is converting elements that match to uppercase.

Sample of Visual Basic Code

```
Public Class MyStringList
    Inherits List(Of String)
    Public Function Where(ByVal filter As Predicate(Of String)) As IEnumerable(Of
String)
        Return Me.Select(Of String)(Function(s) IIf(filter(s), s.ToUpper(), s))
    End Function
End Class
```

Sample of C# Code

```
public class MyStringList : List<string>
{
    public IEnumerable<string> Where(Predicate<string> filter)
    {
        return this.Select(s=>filter(s) ? s.ToUpper() : s);
```

```
    }
}
```

When the compiler looks for a *Where* method on a *MyStringList* object, it finds an implementation in the type itself and thus does not need to look for any extension method. The following code shows an example:

Sample of Visual Basic Code

```
Dim strings As New MyStringList From {"orange", "apple", "grape", "pear"}
For Each item In strings.Where(Function(s) s.Length = 5)
    txtLog.WriteLine(item)
Next
```

Sample of C# Code

```
var strings = new MyStringList{"orange","apple","grape","pear"};
foreach (var item in strings.Where(s => s.Length == 5))
{
    txtLog.WriteLine(item);
}
```

This produces four items in the output, but the apple and grape will be uppercase. To call the *Where* extension method, use *AsEnumerable* as follows:

Sample of Visual Basic Code

```
For Each item In strings.AsEnumerable().Where(Function(s) s.Length = 5)
    txtLog.WriteLine(item)
Next
```

Sample of C# Code

```
foreach (var item in strings.AsEnumerable().Where(s => s.Length == 5))
{
    txtLog.WriteLine(item);
}
```

This produces only two items, the apple and the grape, and they will not be uppercase.

ASPARALLEL

See "Parallel LINQ (PLINQ)" later in this chapter.

ASQUERYABLE

The *AsQueryable* extension method converts an *IEnumerable* object to an *IQueryable* object. This might be needed because the *IQueryable* interface is typically implemented by query providers to provide custom query capabilities such as passing a query back to a database for execution.

The *IQueryable* interface inherits the *IEnumerable* interface so the results of a query can be enumerated. Enumeration causes any code associated with an *IQueryable* object to be executed. In the following example, the *AsQueryable* method is executed, and information is now available for the provider.

Sample of Visual Basic Code

```
Private Sub asQueryableToolStripMenuItem_Click(ByVal sender As System.Object, _
        ByVal e As System.EventArgs) _
        Handles asQueryableToolStripMenuItem.Click
    Dim strings As New MyStringList From {"orange", "apple", "grape", "pear"}
    Dim querable = strings.AsQueryable()
    txtLog.WriteLine("Element Type:{0}", querable.ElementType)
    txtLog.WriteLine("Expression:{0}", querable.Expression)
    txtLog.WriteLine("Provider:{0}", querable.Provider)
End Sub
```

Sample of C# Code

```
private void asQueryableToolStripMenuItem_Click(object sender, EventArgs e)
{
    IEnumerable<string> strings = new MyStringList
        { "orange", "apple", "grape", "pear" };
    var querable = strings.AsQueryable();
    txtLog.WriteLine("Element Type:{0}", querable.ElementType);
    txtLog.WriteLine("Expression:{0}", querable.Expression);
    txtLog.WriteLine("Provider:{0}", querable.Provider);
}
```

You can think of the *AsQueryable* method as the opposite of the *AsEnumerable* method.

AVERAGE

The *Average* extension method is an aggregate extension method that can calculate an average of a numeric property that exists on the elements in your collection. In the following example, the *Average* method calculates the average year of the cars.

Sample of Visual Basic Code

```
Dim averageYear = GetCars().Average(Function(c) c.Year)
txtLog.WriteLine(averageYear)
```

Sample of C# Code

```
var averageYear = GetCars().Average(c => c.Year);
txtLog.WriteLine(averageYear);
```

CAST

Use the *Cast* extension method when you want to convert each of the elements in your source to a different type. The elements in the source must be coerced to the target type, or an *InvalidCastException* is thrown.

Note that the *Cast* method is not a filter. If you want to retrieve all the elements of a specific type, use the *OfType* extension method. The following example converts *IEnumerable* of *Car* to *IEnumerable* of *Object*.

Sample of Visual Basic Code

```
Private Sub castToolStripMenuItem_Click(ByVal sender As System.Object, _
        ByVal e As System.EventArgs) Handles castToolStripMenuItem.Click
    Dim cars As IEnumerable(Of Car) = GetCars()
```

```
    Dim objects As IEnumerable(Of Object) = cars.Cast(Of Object)()
End Sub
```

Sample of C# Code

```
private void castToolStripMenuItem_Click(object sender, EventArgs e)
{
    IEnumerable<Car> cars = GetCars();
    IEnumerable<Object> objects = cars.Cast<object>();
}
```

CONCAT

Use the *Concat* extension method to combine two sequences. This method is similar to the *Union* operator, but *Union* removes duplicates, whereas *Concat* does not remove duplicates.

The following example combines two collections to produce a result that contains all elements from both collections.

Sample of Visual Basic Code

```
Private Sub concatToolStripMenuItem_Click(ByVal sender As System.Object, _
       ByVal e As System.EventArgs) _
       Handles concatToolStripMenuItem.Click
   Dim lastYearScores As Integer() = New Integer() {88, 56, 23, 99, 65}
   Dim thisYearScores As Integer() = New Integer() {93, 78, 23, 99, 90}
   Dim item As Integer
   For Each item In lastYearScores.Concat(thisYearScores)
       Me.txtLog.WriteLine(item)
   Next
End Sub
```

Sample of C# Code

```
private void concatToolStripMenuItem_Click(object sender, EventArgs e)
{
    int[] lastYearScores = { 88, 56, 23, 99, 65 };
    int[] thisYearScores = { 93, 78, 23, 99, 90 };

    foreach (var item in lastYearScores.Concat(thisYearScores))
    {
        txtLog.WriteLine(item);
    }
}
```

The result:

```
88
56
23
99
65
93
78
23
99
90
```

CONTAINS

The *Contains* extension method determines whether an element exists in the source. If the element has the same values for all the properties as one item in the source, *Contains* returns *false* for a reference type but *true* for a value type. The comparison is done by reference for classes and by value for structures. The following example code gets a collection of cars, creates one variable that references one of the cars in the collection, and then creates another variable that references a new car.

Sample of Visual Basic Code

```
Private Sub containsToolStripMenuItem_Click(ByVal sender As System.Object, _
        ByVal e As System.EventArgs) _
        Handles containsToolStripMenuItem.Click
    Dim cars = Me.GetCars
    Dim c1 = cars.Item(2)
    Dim c2 As New Car
    txtLog.WriteLine(cars.Contains(c1))
    txtLog.WriteLine(cars.Contains(c2))
End Sub
```

Sample of C# Code

```
private void containsToolStripMenuItem_Click(object sender, EventArgs e)
{
    var cars = GetCars();
    Car c1 = cars[2];
    Car c2 = new Car();
    txtLog.WriteLine(cars.Contains(c1));
    txtLog.WriteLine(cars.Contains(c2));
}
```

The result:

```
True
False
```

COUNT

The *Count* extension method returns the count of the elements in the source. The following code example returns the count of cars in the source collection:

Sample of Visual Basic Code

```
Private Sub countToolStripMenuItem_Click(ByVal sender As System.Object, _
        ByVal e As System.EventArgs) _
        Handles countToolStripMenuItem.Click
    Dim cars = Me.GetCars
    txtLog.WriteLine(cars.Count())
End Sub
```

Sample of C# Code

```
private void countToolStripMenuItem_Click(object sender, EventArgs e)
{
    var cars = GetCars();
    txtLog.WriteLine(cars.Count());
}
```

The result:

```
5
```

DEFAULTIFEMPTY

Use the *DefaultIfEmpty* extension method when you suspect that the source collection might not have any elements, but you want at least one element corresponding to the default value of the type (*false* for Boolean, *0* for numeric, and *null* for a reference type). This method returns all elements in the source if there is at least one element in the source.

Sample of Visual Basic Code

```
Private Sub defaultIfEmptyToolStripMenuItem_Click(ByVal sender As System.Object, _
       ByVal e As System.EventArgs) _
       Handles defaultIfEmptyToolStripMenuItem.Click
   Dim cars As New List(Of Car)
   Dim oneNullCar = cars.DefaultIfEmpty()
   For Each car In oneNullCar
       txtLog.WriteLine(IIf((car Is Nothing), "Null Car", "Not Null Car"))
   Next
End Sub
```

Sample of C# Code

```
private void defaultIfEmptyToolStripMenuItem_Click(object sender, EventArgs e)
{
    List<Car> cars = new List<Car>();
    IEnumerable<Car> oneNullCar = cars.DefaultIfEmpty();
    foreach (var car in oneNullCar)
    {
        txtLog.WriteLine(car == null ? "Null Car" : "Not Null Car");
    }
}
```

The result:

```
Null Car
```

DISTINCT

The *Distinct* extension method removes duplicate values in the source. The following code sample shows how a collection that has duplicate values is filtered by using the *Distinct* method. Matching to detect duplication follows the same rules as for *Contains*.

Sample of Visual Basic Code

```
Private Sub distinctToolStripMenuItem_Click(ByVal sender As System.Object, _
        ByVal e As System.EventArgs) _
        Handles distinctToolStripMenuItem.Click
    Dim scores = New Integer() {88, 56, 23, 99, 65, 93, 78, 23, 99, 90}
    For Each score In scores.Distinct()
        txtLog.WriteLine(score)
    Next
End Sub
```

Sample of C# Code

```
private void distinctToolStripMenuItem_Click(object sender, EventArgs e)
{
    int[] scores = { 88, 56, 23, 99, 65, 93, 78, 23, 99, 90 };
    foreach (var score in scores.Distinct())
    {
        txtLog.WriteLine(score);
    }
}
```

The result:

```
88
56
23
99
65
93
78
90
```

ELEMENTAT

Use the *ElementAt* extension method when you know you want to retrieve the *n*th element in the source. If there is a valid element at that 0-based location, it's returned or an *ArgumentOutOfRangeException* is thrown.

Sample of Visual Basic Code

```
Private Sub elementAtToolStripMenuItem_Click(ByVal sender As System.Object, _
            ByVal e As System.EventArgs) _
            Handles elementAtToolStripMenuItem.Click
    Dim scores = New Integer() {88, 56, 23, 99, 65, 93, 78, 23, 99, 90}
    txtLog.WriteLine(scores.ElementAt(4))
End Sub
```

Sample of C# Code

```
private void elementAtToolStripMenuItem_Click(object sender, EventArgs e)
{
    int[] scores = { 88, 56, 23, 99, 65, 93, 78, 23, 99, 90 };
    txtLog.WriteLine(scores.ElementAt(4));
}
```

The result:

```
65
```

ELEMENTATORDEFAULT

The *ElementAtOrDefault* extension method is the same as the *ElementAt* extension method except that an exception is not thrown if the element doesn't exist. Instead, the default value for the type of the collection is returned. The sample code attempts to access an element that doesn't exist.

Sample of Visual Basic Code

```
Private Sub elementAtOrDefaultToolStripMenuItem_Click( _
        ByVal sender As System.Object, _
        ByVal e As System.EventArgs) _
        Handles elementAtOrDefaultToolStripMenuItem.Click
    Dim scores = New Integer() {88, 56, 23, 99, 65, 93, 78, 23, 99, 90}
    txtLog.WriteLine(scores.ElementAtOrDefault(15))
End Sub
```

Sample of C# Code

```
private void elementAtOrDefaultToolStripMenuItem_Click(
    object sender, EventArgs e)
{
    int[] scores = { 88, 56, 23, 99, 65, 93, 78, 23, 99, 90 };
    txtLog.WriteLine(scores.ElementAtOrDefault(15));
}
```

The result:

```
0
```

EXCEPT

When you have a sequence of elements and you want to find out which elements don't exist (as usual, by reference for classes and by value for structures) in a second sequence, use the *Except* extension method. The following code sample returns the differences between two collections of integers.

Sample of Visual Basic Code

```
Private Sub exceptToolStripMenuItem_Click(ByVal sender As System.Object, _
        ByVal e As System.EventArgs) _
    Handles exceptToolStripMenuItem.Click
    Dim lastYearScores As Integer() = New Integer() {88, 56, 23, 99, 65}
    Dim thisYearScores As Integer() = New Integer() {93, 78, 23, 99, 90}
    Dim item As Integer
    For Each item In lastYearScores.Except(thisYearScores)
        Me.txtLog.WriteLine(item)
    Next
End Sub
```

Sample of C# Code

```
private void exceptToolStripMenuItem_Click(object sender, EventArgs e)
{
    int[] lastYearScores = { 88, 56, 23, 99, 65 };
    int[] thisYearScores = { 93, 78, 23, 99, 90 };

    foreach (var item in lastYearScores.Except(thisYearScores))
    {
        txtLog.WriteLine(item);
    }
}
```

The result:

```
88
56
65
```

FIRST

When you have a sequence of elements and you just need the first element, use the *First* extension method. This method doesn't care how many elements are in the sequence as long as there is at least one element. If no elements exist, an *InvalidOperationException* is thrown.

Sample of Visual Basic Code

```
Private Sub firstToolStripMenuItem_Click(ByVal sender As System.Object, _
        ByVal e As System.EventArgs) _
        Handles firstToolStripMenuItem.Click
    Dim scores = New Integer() {88, 56, 23, 99, 65, 93, 78, 23, 99, 90}
    txtLog.WriteLine(scores.First())
End Sub
```

Sample of C# Code

```
private void firstToolStripMenuItem_Click(object sender, EventArgs e)
{
    int[] scores = { 88, 56, 23, 99, 65, 93, 78, 23, 99, 90 };
    txtLog.WriteLine(scores.First());
}
```

The result:

```
88
```

FIRSTORDEFAULT

The *FirstOrDefault* extension method is the same as the *First* extension method except that if no elements exist in the source sequence, the default value of the sequence type is returned. This example will attempt to get the first element when there are no elements.

Sample of Visual Basic Code

```
Private Sub firstOrDefaultToolStripMenuItem_Click(ByVal sender As System.Object, _
        ByVal e As System.EventArgs) _
        Handles firstOrDefaultToolStripMenuItem.Click
    Dim scores = New Integer() {}
    txtLog.WriteLine(scores.FirstOrDefault())
End Sub
```

Sample of C# Code

```
private void firstOrDefaultToolStripMenuItem_Click(object sender, EventArgs e)
{
    int[] scores = { };
    txtLog.WriteLine(scores.FirstOrDefault());
}
```

The result:

```
0
```

GROUPBY

The *GroupBy* extension method returns a sequence of *IGrouping<TKey, TElement>* objects. This interface implements *IEnumerable<TElement>* and exposes a single *Key* property that represents the grouping key value. The following code sample groups cars by the *Make* property.

Sample of Visual Basic Code

```
Private Sub groupByToolStripMenuItem_Click(ByVal sender As System.Object, _
        ByVal e As System.EventArgs) _
        Handles groupByToolStripMenuItem.Click
    Dim cars = GetCars()
    Dim query = cars.GroupBy(Function(c) c.Make)
    For Each group As IGrouping(Of String, Car) In query
        txtLog.WriteLine("Key:{0}", group.Key)
        For Each c In group
            txtLog.WriteLine("Car VIN:{0} Make:{1}", c.VIN, c.Make)
        Next
    Next
End Sub
```

Sample of C# Code

```
private void groupByToolStripMenuItem_Click(object sender, EventArgs e)
{
    var cars = GetCars();
    var query = cars.GroupBy(c => c.Make);
    foreach (IGrouping<string,Car> group in query)
    {
        txtLog.WriteLine("Key:{0}", group.Key);
        foreach (Car c in group)
        {
            txtLog.WriteLine("Car VIN:{0} Make:{1}", c.VIN, c.Make);
        }
    }
}
```

The result:

```
Key:Ford
Car VIN:ABC123 Make:Ford
Car VIN:DEF456 Make:Ford
Key:BMW
Car VIN:DEF123 Make:BMW
Key:Audi
Car VIN:ABC456 Make:Audi
Key:VW
Car VIN:HIJ123 Make:VW
```

Because there are two Fords, they are grouped together.

The *ToLookup* extension method provides the same result except that *GroupBy* returns a deferred query, whereas *ToLookup* executes the query immediately, and iterating on the result afterward will not change if the source changes. This is equivalent to the *ToList* extension method introduced earlier in this chapter, but for *IGrouping* instead of for *IEnumerable*.

GROUPJOIN

The *GroupJoin* extension method is similar to the SQL left outer join where it always produces one output for each input from the "outer" sequence. Any matching elements from the inner sequence are grouped into a collection that is associated with the outer element. In the following example code, a collection of *Makes* is provided and joined to the *cars* collection.

Sample of Visual Basic Code

```
Private Sub groupToolStripMenuItem_Click(ByVal sender As System.Object, _
        ByVal e As System.EventArgs) Handles groupToolStripMenuItem.Click
    Dim makes = New String() {"Audi", "BMW", "Ford", "Mazda", "VW"}
    Dim cars = GetCars()

    Dim query = makes.GroupJoin(cars, _
            Function(make) make, _
            Function(car) car.Make, _
            Function(make, innerCars) New With {.Make = make, .Cars = innerCars})

    For Each item In query
        txtLog.WriteLine("Make: {0}", item.Make)
        For Each car In item.Cars
            txtLog.WriteLine("Car VIN:{0}, Model:{1}", car.VIN, car.Model)
        Next
    Next
End Sub
```

Sample of C# Code

```
private void groupToolStripMenuItem_Click(object sender, EventArgs e)
{
    var makes = new string[] { "Audi", "BMW", "Ford", "Mazda", "VW" };
    var cars = GetCars();

    var query = makes.GroupJoin(cars,
        make => make, car => car.Make,
        (make, innerCars) => new { Make = make, Cars = innerCars });

    foreach (var item in query)
    {
        txtLog.WriteLine("Make: {0}", item.Make);
        foreach (var car in item.Cars)
        {
            txtLog.WriteLine("Car VIN:{0}, Model:{1}", car.VIN, car.Model);
        }
    }
}
```

The result:

```
Make: Audi
Car VIN:ABC456, Model:TT
Make: BMW
Car VIN:DEF123, Model:Z-3
Make: Ford
Car VIN:ABC123, Model:F-250
Car VIN:DEF456, Model:F-150
```

```
Make: Mazda
Make: VW
Car VIN:HIJ123, Model:Bug
```

INTERSECT

When you have a sequence of elements in which you want to find out which exist in a second sequence, use the *Intersect* extension method. The following code example returns the common elements that exist in two collections of integers.

Sample of Visual Basic Code

```
Private Sub intersectToolStripMenuItem_Click(ByVal sender As System.Object, _
        ByVal e As System.EventArgs) Handles intersectToolStripMenuItem.Click
    Dim lastYearScores As Integer() = New Integer() {88, 56, 23, 99, 65}
    Dim thisYearScores As Integer() = New Integer() {93, 78, 23, 99, 90}
    Dim item As Integer
    For Each item In lastYearScores.Intersect(thisYearScores)
        Me.txtLog.WriteLine(item)
    Next
End Sub
```

Sample of C# Code

```
private void intersectToolStripMenuItem_Click(object sender, EventArgs e)
{
    int[] lastYearScores = { 88, 56, 23, 99, 65 };
    int[] thisYearScores = { 93, 78, 23, 99, 90 };

    foreach (var item in lastYearScores.Intersect(thisYearScores))
    {
        txtLog.WriteLine(item);
    }
}
```

The result:

```
23
99
```

JOIN

The *Join* extension method is similar to the SQL inner join, by which it produces output only for each input from the outer sequence when there is a match to the inner sequence. For each matching element in the inner sequence, a resulting element is created. In the following sample code, a collection of *Makes* is provided and joined to the *cars* collection.

Sample of Visual Basic Code

```
Private Sub joinToolStripMenuItem_Click(ByVal sender As System.Object, _
        ByVal e As System.EventArgs) Handles joinToolStripMenuItem.Click
    Dim makes = New String() {"Audi", "BMW", "Ford", "Mazda", "VW"}
    Dim cars = GetCars()

    Dim query = makes.Join(cars, _
        Function(make) make, _
        Function(car) car.Make, _
```

```
        Function(make, innerCar) New With {.Make = make, .Car = innerCar})
    For Each item In query
        txtLog.WriteLine("Make: {0}, Car:{1} {2} {3}",
            item.Make, item.Car.VIN, item.Car.Make, item.Car.Model)
    Next
End Sub
```

Sample of C# Code

```
private void joinToolStripMenuItem_Click(object sender, EventArgs e)
{
    var makes = new string[] { "Audi", "BMW", "Ford", "Mazda", "VW" };
    var cars = GetCars();

    var query = makes.Join(cars,
        make => make, car => car.Make,
        (make, innerCar) => new { Make = make, Car = innerCar });

    foreach (var item in query)
    {
        txtLog.WriteLine("Make: {0}, Car:{1} {2} {3}",
            item.Make, item.Car.VIN, item.Car.Make, item.Car.Model);
    }
}
```

The result:

```
Make: Audi, Car:ABC456 Audi TT
Make: BMW, Car:DEF123 BMW Z-3
Make: Ford, Car:ABC123 Ford F-250
Make: Ford, Car:DEF456 Ford F-150
Make: VW, Car:HIJ123 VW Bug
```

EXAM TIP

For the exam, expect to be tested on the various ways to join sequences.

LAST

When you want to retrieve the last element in a sequence, use the *Last* extension method. This method throws an *InvalidOperationException* if there are no elements in the sequence. The following sample code retrieves the last element.

Sample of Visual Basic Code

```
Private Sub lastToolStripMenuItem_Click(ByVal sender As System.Object, _
        ByVal e As System.EventArgs) Handles lastToolStripMenuItem.Click
    Dim scores = New Integer() {88, 56, 23, 99, 65, 93, 78, 23, 99, 90}
    txtLog.WriteLine(scores.Last())
End Sub
```

Sample of C# Code

```
private void lastToolStripMenuItem_Click(object sender, EventArgs e)
{
    int[] scores = { 88, 56, 23, 99, 65, 93, 78, 23, 99, 90 };
```

```
    txtLog.WriteLine(scores.Last());
}
```

The result:

```
90
```

LASTORDEFAULT

The *LastOrDefault* extension method is the same as the *Last* extension method except that if no elements exist in the source sequence, the default value of the sequence type is returned. This example will attempt to get the last element when there are no elements.

Sample of Visual Basic Code

```
Private Sub lastOrDefaultToolStripMenuItem_Click(ByVal sender As System.Object, _
       ByVal e As System.EventArgs) Handles lastOrDefaultToolStripMenuItem.Click
   Dim scores = New Integer() {}
   txtLog.WriteLine(scores.LastOrDefault())
End Sub
```

Sample of C# Code

```
private void lastOrDefaultToolStripMenuItem_Click(object sender, EventArgs e)
{
    int[] scores = { };
    txtLog.WriteLine(scores.LastOrDefault());
}
```

The result:

```
0
```

LONGCOUNT

The *LongCount* extension method is the same as the *Count* extension method except that *Count* returns a 32-bit integer, and *LongCount* returns a 64-bit integer.

Sample of Visual Basic Code

```
Private Sub longCountToolStripMenuItem_Click(ByVal sender As System.Object, _
        ByVal e As System.EventArgs) Handles longCountToolStripMenuItem.Click
    Dim cars = Me.GetCars
    txtLog.WriteLine(cars.LongCount())
End Sub
```

Sample of C# Code

```
private void longCountToolStripMenuItem_Click(object sender, EventArgs e)
{
    var cars = GetCars();
    txtLog.WriteLine(cars.LongCount());
}
```

The result:

```
5
```

MAX

When you're working with a non-empty sequence of values and you want to determine which element is greatest, use the *Max* extension method. The *Max* extension has several overloads, but the following code sample shows two of the more common overloads that demonstrate the *Max* extension method's capabilities.

Sample of Visual Basic Code

```
Private Sub maxToolStripMenuItem_Click(ByVal sender As System.Object, _
        ByVal e As System.EventArgs) Handles maxToolStripMenuItem.Click
    Dim scores = New Integer() {88, 56, 23, 99, 65, 93, 78, 23, 99, 90}
    txtLog.WriteLine(scores.Max())

      Dim cars = GetCars()
      txtLog.WriteLine(cars.Max(Function(c) c.Year))
End Sub
```

Sample of C# Code

```
private void maxToolStripMenuItem_Click(object sender, EventArgs e)
{
    int[] scores = { 88, 56, 23, 99, 65, 93, 78, 23, 99, 90 };
    txtLog.WriteLine(scores.Max());

    var cars = GetCars();
    txtLog.WriteLine(cars.Max(c => c.Year));
}
```

The result:

```
99

2008
```

In this example, the parameterless overload is called on a collection of integers and returns the maximum value of 99. The next overload example enables you to provide a selector that specifies a property that finds the maximum value.

MIN

When you're working with a non-empty sequence of values and you want to determine which element is the smallest, use the *Min* extension method, as shown in the following code sample.

Sample of Visual Basic Code

```
Private Sub maxToolStripMenuItem_Click(ByVal sender As System.Object, _
        ByVal e As System.EventArgs) Handles maxToolStripMenuItem.Click
    Dim scores = New Integer() {88, 56, 23, 99, 65, 93, 78, 23, 99, 90}
    txtLog.WriteLine(scores.Min())
End Sub
```

Sample of C# Code

```
private void maxToolStripMenuItem_Click(object sender, EventArgs e)
{
    int[] scores = { 88, 56, 23, 99, 65, 93, 78, 23, 99, 90 };
```

```
    txtLog.WriteLine(scores.Min());
}
```

The result:

```
23
```

OFTYPE

The *OfType* extension method is a filtering method that returns only objects that can be type cast to a specific type. The following sample code retrieves just the integers from the object collection.

Sample of Visual Basic Code

```
Private Sub ofTypeToolStripMenuItem_Click(ByVal sender As System.Object, _
        ByVal e As System.EventArgs) Handles ofTypeToolStripMenuItem.Click
    Dim items = New Object() {55, "Hello", 22, "Goodbye"}
    For Each intItem In items.OfType(Of Integer)()
        txtLog.WriteLine(intItem)
    Next
End Sub
```

Sample of C# Code

```
private void ofTypeToolStripMenuItem_Click(object sender, EventArgs e)
{
    object[] items = new object[] { 55, "Hello", 22, "Goodbye" };
    foreach (var intItem in items.OfType<int>())
    {
        txtLog.WriteLine(intItem);
    }
}
```

The result:

```
55
22
```

ORDERBY, *ORDERBYDESCENDING*, *THENBY*, AND *THENBYDESCENDING*

When you want to sort the elements in a sequence, you can use the *OrderBy* or *OrderByDescending* extension methods, followed by the *ThenBy* and *ThenByDescending* extension methods. These extension methods are *nonstreaming*, which means that all elements in the sequence must be evaluated before any output can be produced. Most extension methods are *streaming*, which means that each element can be evaluated and potentially output without having to evaluate all elements.

All these extension methods return an *IOrderedEnumerable<T>* object, which inherits from *IEnumerable<T>* and enables the *ThenBy* and *ThenByDescending* operators. *ThenBy* and *ThenByDescending* are extension methods on *IOrderedEnumerable<T>* instead of on *IEnumerable<T>*, which the other extension methods extend. This can sometimes create unexpected errors when using the *var* keyword and type inference. The following code example creates a list of cars and then sorts them by make, model descending, and year.

Sample of Visual Basic Code

```
Private Sub orderByToolStripMenuItem_Click(ByVal sender As System.Object, _
        ByVal e As System.EventArgs) Handles orderByToolStripMenuItem.Click
    Dim cars = GetCars().OrderBy(Function(c) c.Make) _
                        .ThenByDescending(Function(c) c.Model) _
                        .ThenBy(Function(c) c.Year)
    For Each item In cars
        txtLog.WriteLine("Car VIN:{0} Make:{1} Model:{2} Year:{3}", _
        item.VIN, item.Make, item.Model, item.Year)
    Next
End Sub
```

Sample of C# Code

```
private void orderByToolStripMenuItem_Click(object sender, EventArgs e)
{
    var cars = GetCars().OrderBy(c=>c.Make)
                            .ThenByDescending(c=>c.Model)
                            .ThenBy(c=>c.Year);
    foreach (var item in cars)
    {
        txtLog.WriteLine("Car VIN:{0} Make:{1} Model:{2} Year:{3}",
            item.VIN, item.Make, item.Model, item.Year);
    }
}
```

The result:

```
Car VIN:ABC456 Make:Audi Model:TT Year:2008
Car VIN:DEF123 Make:BMW Model:Z-3 Year:2005
Car VIN:ABC123 Make:Ford Model:F-250 Year:2000
Car VIN:DEF456 Make:Ford Model:F-150 Year:1998
Car VIN:HIJ123 Make:VW Model:Bug Year:1956
```

REVERSE

The *Reverse* extension method is an ordering mechanism that reverses the order of the sequence elements. This code sample creates a collection of integers but displays them in reverse order.

Sample of Visual Basic Code

```
Private Sub reverseToolStripMenuItem_Click(ByVal sender As System.Object, _
        ByVal e As System.EventArgs) Handles reverseToolStripMenuItem.Click
    Dim scores = {88, 56, 23, 99, 65, 93, 78, 23, 99, 90}

    For Each item In scores.Reverse()
        txtLog.WriteLine(item)
    Next
End Sub
```

Sample of C# Code

```
private void reverseToolStripMenuItem_Click(object sender, EventArgs e)
{
    int[] scores = { 88, 56, 23, 99, 65, 93, 78, 23, 99, 90 };
    foreach (var item in scores.Reverse())
```

```
    {
        txtLog.WriteLine(item);
    }
}
```

The result:

```
90
99
23
78
93
65
99
23
56
88
```

SELECT

The *Select* extension method returns one output element for each input element. Although *Select* returns one output for each input, the *Select* operator also enables you to perform a projection to a new type of element. This conversion or mapping mechanism plays an important role in most LINQ queries.

In the following example, a collection of *Tuple* types is queried to retrieve all the elements whose make (*Tuple.item2*) is *Ford*, but the *Select* extension method transforms these *Tuple* types into *Car* objects.

Sample of Visual Basic Code

```
Private Sub selectToolStripMenuItem_Click(ByVal sender As System.Object, _
        ByVal e As System.EventArgs) Handles selectToolStripMenuItem.Click
    Dim vehicles As New List(Of Tuple(Of String, String, Integer)) From { _
            Tuple.Create(Of String, String, Integer)("123", "VW", 1999), _
            Tuple.Create(Of String, String, Integer)("234", "Ford", 2009), _
            Tuple.Create(Of String, String, Integer)("567", "Audi", 2005), _
            Tuple.Create(Of String, String, Integer)("678", "Ford", 2003), _
            Tuple.Create(Of String, String, Integer)("789", "Mazda", 2003), _
            Tuple.Create(Of String, String, Integer)("999", "Ford", 1965) _
            }
    Dim fordCars = vehicles.Where(Function(v) v.Item2 = "Ford") _
            .Select(Function(v) New Car With { _
                                                .VIN = v.Item1, _
                                                .Make = v.Item2, _
                                                .Year = v.Item3 _
                                            })
    For Each item In fordCars
        txtLog.WriteLine("Car VIN:{0} Make:{1} Year:{2}", _
                         item.VIN, item.Make, item.Year)
    Next
End Sub
```

Sample of C# Code

```
private void selectToolStripMenuItem_Click(object sender, EventArgs e)
```

```
{
    var vehicles = new List<Tuple<string,string,int>>
    {
        Tuple.Create("123", "VW", 1999),
        Tuple.Create("234","Ford",2009),
        Tuple.Create("567","Audi", 2005),
        Tuple.Create("678","Ford", 2003),
        Tuple.Create("789","Mazda", 2003),
        Tuple.Create("999","Ford",1965)
    };

    var fordCars = vehicles
                        .Where(v=>v.Item2=="Ford")
                        .Select(v=>new Car
                        {
                            VIN=v.Item1,
                            Make=v.Item2,
                            Year=v.Item3
                        });
    foreach (var item in fordCars )
    {
        txtLog.WriteLine("Car VIN:{0} Make:{1} Year:{2}",
              item.VIN, item.Make, item.Year);
    }
}
```

The result:

```
Car VIN:234 Make:Ford Year:2009
Car VIN:678 Make:Ford Year:2003
Car VIN:999 Make:Ford Year:1965
```

SELECTMANY

When you use the *Select* extension method, each element from the input sequence can produce only one element in the output sequence. The *SelectMany* extension method projects a single output element into many output elements, so you can use the *SelectMany* method to perform a SQL inner join, but you can also use it when you are working with a collection of collections, and you are querying the outer collection but need to produce an output element for each element in the inner collection.

In the following code sample is a list of repairs in which each element in the *repairs* collection is a *Tuple* that contains the VIN of the vehicle as *item1* and a list of repairs as *item2*. *SelectMany* expands each *Tuple* into a sequence of repairs. *Select* projects each repair and the associated VIN into an anonymous type instance.

Sample of Visual Basic Code

```
Private Sub selectManyToolStripMenuItem_Click(ByVal sender As System.Object, _
        ByVal e As System.EventArgs) Handles selectManyToolStripMenuItem.Click
    Dim repairs = New List(Of Tuple(Of String, List(Of String))) From
            {
                Tuple.Create("ABC123",
                    New List(Of String) From {"Rotate Tires", "Change oil"}),
                Tuple.Create("DEF123",
```

```
                    New List(Of String) From {"Fix Flat", "Wash Vehicle"}),
                Tuple.Create("ABC456",
                    New List(Of String) From {"Alignment", "Vacuum", "Wax"}),
                Tuple.Create("HIJ123",
                    New List(Of String) From {"Spark plugs", "Air filter"}),
                Tuple.Create("DEF456",
                    New List(Of String) From {"Wiper blades", "PVC valve"})
            }
    Dim query = repairs.SelectMany(Function(t) _
        t.Item2.Select(Function(r) New With {.VIN = t.Item1, .Repair = r}))

    For Each item In query
        txtLog.WriteLine("VIN:{0} Repair:{1}", item.VIN, item.Repair)
    Next
End Sub
```

Sample of C# Code

```
private void selectManyToolStripMenuItem_Click(object sender, EventArgs e)
{
    var repairs = new List<Tuple<string, List<string>>>
                {
                    Tuple.Create("ABC123",
                        new List<string>{"Rotate Tires","Change oil"}),
                    Tuple.Create("DEF123",
                        new List<string>{"Fix Flat","Wash Vehicle"}),
                    Tuple.Create("ABC456",
                        new List<string>{"Alignment","Vacuum", "Wax"}),
                    Tuple.Create("HIJ123",
                        new List<string>{"Spark plugs","Air filter"}),
                    Tuple.Create("DEF456",
                        new List<string>{"Wiper blades","PVC valve"}),
                };
    var query = repairs.SelectMany(t =>
        t.Item2.Select(r => new { VIN = t.Item1, Repair = r }));

    foreach (var item in query)
    {
        txtLog.WriteLine("VIN:{0} Repair:{1}", item.VIN, item.Repair);
    }
}
```

The result:

```
VIN:ABC123 Repair:Rotate Tires
VIN:ABC123 Repair:Change oil
VIN:DEF123 Repair:Fix Flat
VIN:DEF123 Repair:Wash Vehicle
VIN:ABC456 Repair:Alignment
VIN:ABC456 Repair:Vacuum
VIN:ABC456 Repair:Wax
VIN:HIJ123 Repair:Spark plugs
VIN:HIJ123 Repair:Air filter
VIN:DEF456 Repair:Wiper blades
VIN:DEF456 Repair:PVC valve
```

SEQUENCEEQUAL

One scenario you might run into is when you have two sequences and want to see whether they contain the same elements in the same order. The *SequenceEqual* extension method can perform this task. It walks through two sequences and compares the elements inside for equality. You can also override the equality test by providing an *IEqualityComparer* object as a parameter. The following example code compares two sequences several times and displays the result.

Sample of Visual Basic Code

```
Private Sub sequenceEqualToolStripMenuItem_Click(ByVal sender As System.Object, _
        ByVal e As System.EventArgs) Handles sequenceEqualToolStripMenuItem.Click
    Dim lastYearScores = New List(Of Integer) From {93, 78, 23, 99, 91}
    Dim thisYearScores = New List(Of Integer) From {93, 78, 23, 99, 90}
    txtLog.WriteLine(lastYearScores.SequenceEqual(thisYearScores))
    lastYearScores(4) = 90
    txtLog.WriteLine(lastYearScores.SequenceEqual(thisYearScores))
    thisYearScores.Add(85)
    txtLog.WriteLine(lastYearScores.SequenceEqual(thisYearScores))
    lastYearScores.Add(85)
    txtLog.WriteLine(lastYearScores.SequenceEqual(thisYearScores))
    lastYearScores.Add(75)
    txtLog.WriteLine(lastYearScores.SequenceEqual(thisYearScores))
End Sub
```

Sample of C# Code

```
private void sequenceEqualToolStripMenuItem_Click(object sender, EventArgs e)
{
    var lastYearScores = new List<int>{ 93, 78, 23, 99, 91 };
    var thisYearScores = new List<int>{ 93, 78, 23, 99, 90 };
    txtLog.WriteLine(lastYearScores.SequenceEqual(thisYearScores));
    lastYearScores[4] = 90;
    txtLog.WriteLine(lastYearScores.SequenceEqual(thisYearScores));
    thisYearScores.Add(85);
    txtLog.WriteLine(lastYearScores.SequenceEqual(thisYearScores));
    lastYearScores.Add(85);
    txtLog.WriteLine(lastYearScores.SequenceEqual(thisYearScores));
    lastYearScores.Add(75);
    txtLog.WriteLine(lastYearScores.SequenceEqual(thisYearScores));
}
```

The result:

```
False
True
False
True
False
```

SINGLE

The *Single* extension method should be used when you have a collection of one element and want to convert the generic *IEnumerable* interface to the single element. If the sequence contains more than one element or no elements, an exception is thrown. The following example

code queries to retrieve the car with VIN HIJ123 and then uses the *Single* extension method to convert *IEnumerable* to the single *Car*.

Sample of Visual Basic Code

```
Private Sub singleToolStripMenuItem_Click(ByVal sender As System.Object, _
        ByVal e As System.EventArgs) Handles singleToolStripMenuItem.Click
    Dim cars = GetCars()
    Dim myCar As Car = cars.Where(Function(c) c.VIN = "HIJ123").Single()
    txtLog.WriteLine("Car VIN:{0}, Make:{1}, Model:{2}", _
        myCar.VIN, myCar.Make, myCar.Model)
End Sub
```

Sample of C# Code

```
private void singleToolStripMenuItem_Click(object sender, EventArgs e)
{
    var cars = GetCars();
    Car myCar = cars.Where(c => c.VIN == "HIJ123").Single();
    txtLog.WriteLine("Car VIN:{0}, Make:{1}, Model:{2}",
        myCar.VIN, myCar.Make, myCar.Model);
}
```

The result:

```
Car VIN:HIJ123, Make:VW, Model:Bug
```

SINGLEORDEFAULT

The *SingleOrDefault* extension method works like the *Single* extension method except that it doesn't throw an exception if no elements are in the sequence. It still throws an *InvalidOperationException* if more than one element exists in the sequence. The following sample code attempts to locate a car with an invalid VIN, so no elements exist in the sequence; therefore, the *myCar* variable will be *Nothing* (C# *null*).

Sample of Visual Basic Code

```
Private Sub singleOrDefaultToolStripMenuItem_Click(ByVal sender As System.Object, _
        ByVal e As System.EventArgs) Handles singleOrDefaultToolStripMenuItem.Click
    Dim cars = GetCars()
    Dim myCar = cars.Where(Function(c) c.VIN = "XXXXXX").SingleOrDefault()
    txtLog.WriteLine(myCar Is Nothing)
End Sub
```

Sample of C# Code

```
private void singleOrDefaultToolStripMenuItem_Click(object sender, EventArgs e)
{
    var cars = GetCars();
    Car myCar = cars.Where(c => c.VIN == "XXXXXX").SingleOrDefault();
    txtLog.WriteLine(myCar == null);
}
```

The result:

```
True
```

SKIP

The *Skip* extension method ignores, or jumps over, elements in the source sequence. This method, when combined with the *Take* extension method, typically produces paged result-sets to the GUI. The following sample code demonstrates the use of the *Skip* extension method when sorting scores and then skipping over the lowest score to display the rest of the scores.

Sample of Visual Basic Code

```
Private Sub skipToolStripMenuItem_Click(ByVal sender As System.Object, _
        ByVal e As System.EventArgs) Handles skipToolStripMenuItem.Click
    Dim scores = {88, 56, 23, 99, 65, 93, 78, 23, 99, 90}
    For Each score In scores.OrderBy(Function(i) i).Skip(1)
        txtLog.WriteLine(score)
    Next
End Sub
```

Sample of C# Code

```
private void skipToolStripMenuItem_Click(object sender, EventArgs e)
{
    int[] scores = { 88, 56, 23, 99, 65, 93, 78, 23, 99, 90 };
    foreach (var score in scores.OrderBy(i=>i).Skip(1))
    {
        txtLog.WriteLine(score);
    }
}
```

The result:

```
56
65
78
88
90
93
99
99
```

In this example, the score of 23 is missing because the *Skip* method jumped over that element.

SKIPWHILE

The *SkipWhile* extension method is similar to the *Skip* method except *SkipWhile* accepts a predicate that takes an element of the collection and returns a Boolean value to determine when to stop skipping over. The following example code skips over the scores as long as the score is less than 80.

Sample of Visual Basic Code

```
Private Sub skipWhileToolStripMenuItem_Click(ByVal sender As System.Object, _
        ByVal e As System.EventArgs) Handles skipWhileToolStripMenuItem.Click
    Dim scores = {88, 56, 23, 99, 65, 93, 78, 23, 99, 90}
    For Each score In scores.OrderBy(Function(i) i).SkipWhile(Function(s) s < 80)
```

```
        txtLog.WriteLine(score)
    Next
End Sub
```

Sample of C# Code

```
private void skipWhileToolStripMenuItem_Click(object sender, EventArgs e)
{
    int[] scores = { 88, 56, 23, 99, 65, 93, 78, 23, 99, 90 };
    foreach (var score in scores.OrderBy(i => i).SkipWhile(s => s < 80))
    {
        txtLog.WriteLine(score);
    }
}
```

The result:

```
88
90
93
99
99
```

Note that if the scores were not sorted, the *Skip* method would not skip over any elements because the first element (88) is greater than 80.

SUM

The *Sum* extension method is an aggregate function that can loop over the source sequence and calculate a total sum based on the lambda expression passed into this method to select the property to be summed. If the sequence is *IEnumerable* of a numeric type, *Sum* can be executed without a lambda expression. The following example code displays the sum of all the scores.

Sample of Visual Basic Code

```
Private Sub sumToolStripMenuItem_Click(ByVal sender As System.Object, _
        ByVal e As System.EventArgs) Handles sumToolStripMenuItem.Click
    Dim scores = {88, 56, 23, 99, 65, 93, 78, 99, 90}
    txtLog.WriteLine(scores.Sum())
End Sub
```

Sample of C# Code

```
private void sumToolStripMenuItem_Click(object sender, EventArgs e)
{
    int[] scores = { 88, 56, 23, 99, 65, 93, 78, 99, 90 };
    txtLog.WriteLine(scores.Sum());
}
```

The result:

```
714
```

TAKE

The *Take* extension method retrieves a portion of the sequence. You can specify how many elements you want with this method. It is commonly used with the *Skip* method to provide paging ability for data being displayed in the GUI. If you try to take more elements than are available, the *Take* method gracefully returns whatever it can without throwing an exception. The following code sample starts with a collection of integers called *scores*, sorts the collection, skips three elements, and then takes two elements.

Sample of Visual Basic Code

```
Private Sub takeToolStripMenuItem_Click(ByVal sender As System.Object, _
        ByVal e As System.EventArgs) Handles takeToolStripMenuItem.Click
    Dim scores = {88, 56, 23, 99, 65, 93, 78, 23, 99, 90}
    For Each item In scores.OrderBy(Function(i) i).Skip(3).Take(2)
        txtLog.WriteLine(item)
    Next
End Sub
```

Sample of C# Code

```
private void takeToolStripMenuItem_Click(object sender, EventArgs e)
{
    int[] scores = { 88, 56, 23, 99, 65, 93, 78, 23, 99, 90 };
    foreach (var item in scores.OrderBy(i => i).Skip(3).Take(2))
    {
        txtLog.WriteLine(item);
    }
}
```

The results:

```
65
78
```

TAKEWHILE

Just as the *SkipWhile* extension method enables you to skip while the provided predicate returns *true*, the *TakeWhile* extension method enables you to retrieve elements from your sequence as long as the provided predicate returns *true*.

Sample of Visual Basic Code

```
Private Sub takeWhileToolStripMenuItem_Click(ByVal sender As System.Object, _
        ByVal e As System.EventArgs) Handles takeWhileToolStripMenuItem.Click
    Dim scores = {88, 56, 23, 99, 65, 93, 78, 23, 99, 90}
    For Each item In scores.OrderBy(Function(i) i).TakeWhile(Function(s) s < 80)
        txtLog.WriteLine(item)
    Next
End Sub
```

Sample of C# Code

```
private void takeWhileToolStripMenuItem_Click(object sender, EventArgs e)
{
    int[] scores = { 88, 56, 23, 99, 65, 93, 78, 23, 99, 90 };
```

```
    foreach (var item in scores.OrderBy(i => i).TakeWhile(s => s < 80))
    {
        txtLog.WriteLine(item);
    }
}
```

The result:

```
23
23
56
65
78
```

TOARRAY

The *ToArray* extension method executes the deferred query and converts the result to a concrete array of the original sequence item's type. The following code creates a query to retrieve the even scores and converts the deferred query to an array of integers called *evenScores*. The third score is changed to two (even) and, when the even scores are displayed, the two is not in the array.

Sample of Visual Basic Code

```
Private Sub toArrayToolStripMenuItem_Click(ByVal sender As System.Object, _
        ByVal e As System.EventArgs) Handles toArrayToolStripMenuItem.Click
    Dim scores = {88, 56, 23, 99, 65, 93, 78, 23, 99, 90}
    Dim evenScores = scores.Where(Function(s) s Mod 2 = 0).ToArray()
    scores(2) = 2
    For Each item In evenScores
        txtLog.WriteLine(item)
    Next
End Sub
```

Sample of C# Code

```
private void toArrayToolStripMenuItem_Click(object sender, EventArgs e)
{
    int[] scores = { 88, 56, 23, 99, 65, 93, 78, 23, 99, 90 };
    var evenScores = scores.Where(s => s % 2 == 0).ToArray();
    scores[2] = 2;
    foreach (var item in evenScores)
    {
        txtLog.WriteLine(item);
    }
}
```

The result:

```
88
56
78
90
```

TODICTIONARY

The *ToDictionary* extension method executes the deferred query and converts the result to a dictionary with a key type inferred from the return type of the lambda passed as a parameter. The item associated with a dictionary entry is the value from the enumeration that computes the key.

The following code creates a query to retrieve the cars and converts them to a dictionary of cars with the *string* VIN used as the lookup key and assigns the dictionary to a *carsByVin* variable. The car with a VIN of HIJ123 is retrieved and displayed.

Sample of Visual Basic Code

```
Private Sub toDictionaryToolStripMenuItem_Click(ByVal sender As System.Object, _
        ByVal e As System.EventArgs) Handles toDictionaryToolStripMenuItem.Click
    Dim cars = GetCars()
    Dim carsByVin = cars.ToDictionary(Function(c) c.VIN)
    Dim myCar = carsByVin("HIJ123")
    txtLog.WriteLine("Car VIN:{0}, Make:{1}, Model:{2} Year:{3}", _
        myCar.VIN, myCar.Make, myCar.Model, myCar.Year)
End Sub
```

Sample of C# Code

```
private void toDictionaryToolStripMenuItem_Click(object sender, EventArgs e)
{
    var cars = GetCars();
    var carsByVin = cars.ToDictionary(c=>c.VIN);
    Car myCar = carsByVin["HIJ123"];
    txtLog.WriteLine("Car VIN:{0}, Make:{1}, Model:{2} Year:{3}",
        myCar.VIN, myCar.Make, myCar.Model, myCar.Year);
}
```

The result:

```
Car VIN:HIJ123, Make:VW, Model:Bug Year:1956
```

TOLIST

The *ToList* extension method executes the deferred query and stores each item in a *List<T>* where *T* is the same type as the original sequence. The following code creates a query to retrieve the even scores and converts the deferred query to a list of integers called *evenScores*. The third score is changed to two (even) and, when the even scores are displayed, the two is not in the list.

Sample of Visual Basic Code

```
Private Sub toListToolStripMenuItem_Click(ByVal sender As System.Object, _
        ByVal e As System.EventArgs) Handles toListToolStripMenuItem.Click
    Dim scores = {88, 56, 23, 99, 65, 93, 78, 23, 99, 90}
    Dim evenScores = scores.Where(Function(s) s Mod 2 = 0).ToList()
    scores(2) = 2
    For Each item In evenScores
        txtLog.WriteLine(item)
    Next
End Sub
```

Sample of C# Code

```
private void toListToolStripMenuItem_Click(object sender, EventArgs e)
{
    int[] scores = { 88, 56, 23, 99, 65, 93, 78, 23, 99, 90 };
    var evenScores = scores.Where(s => s % 2 == 0).ToList();
    scores[2] = 2;
    foreach (var item in evenScores)
    {
        txtLog.WriteLine(item);
    }
}
```

The result:

```
88
56
78
90
```

TOLOOKUP

The *ToLookup* extension method returns *ILookup<TKey, TElement>*—that is, a sequence of *IGrouping<TKey, TElement>* objects. This interface specifies that the grouping object exposes a *Key* property that represents the grouping value. This method creates a new collection object, thus providing a frozen view. Changing the original source collection will not affect this collection. The following code sample groups cars by the *Make* property. After *ToLookup* is called, the original collection is cleared, but it has no impact on the collection produced by the *ToLookup* method.

Sample of Visual Basic Code

```
Private Sub toLookupToolStripMenuItem_Click(ByVal sender As System.Object, _
        ByVal e As System.EventArgs) Handles toLookupToolStripMenuItem.Click
    Dim cars = GetCars()
    Dim query = cars.ToLookup(Function(c) c.Make)
    cars.Clear()
    For Each group As IGrouping(Of String, Car) In query
        txtLog.WriteLine("Key:{0}", group.Key)
        For Each c In group
            txtLog.WriteLine("Car VIN:{0} Make:{1}", c.VIN, c.Make)
        Next
    Next
End Sub
```

Sample of C# Code

```
private void toLookupToolStripMenuItem_Click(object sender, EventArgs e)
{
    var cars = GetCars();
    var query = cars.ToLookup(c => c.Make);
    cars.Clear();
    foreach (IGrouping<string, Car> group in query)
    {
        txtLog.WriteLine("Key:{0}", group.Key);
        foreach (Car c in group)
```

```
        {
            txtLog.WriteLine("Car VIN:{0} Make:{1}", c.VIN, c.Make);
        }
    }
}
```

The result:

```
Key:Ford
Car VIN:ABC123 Make:Ford
Car VIN:DEF456 Make:Ford
Key:BMW
Car VIN:DEF123 Make:BMW
Key:Audi
Car VIN:ABC456 Make:Audi
Key:VW
Car VIN:HIJ123 Make:VW
```

Because there are two Fords, they are grouped together. The *GroupBy* extension method provides the same result except that *GroupBy* is a deferred query, and *ToLookup* executes the query immediately to return a frozen sequence that won't change even if the original sequence is updated.

UNION

Sometimes, you want to combine two collections and work with the result. Be careful; this might not be the correct solution. You might want to use the *Concat* extension method, which fulfills the requirements of this scenario. The *Union* extension method combines the elements from two sequences but outputs the distinct elements. That is, it filters out duplicates. This is equivalent to executing *Concat* and then *Distinct*. The following code example combines two integer arrays by using the *Union* method and then sorts the result.

Sample of Visual Basic Code

```
Private Sub unionToolStripMenuItem_Click(ByVal sender As System.Object, _
        ByVal e As System.EventArgs) Handles unionToolStripMenuItem.Click
    Dim lastYearScores = {88, 56, 23, 99, 65, 56}
    Dim thisYearScores = {93, 78, 23, 99, 90, 99}
    Dim allScores = lastYearScores.Union(thisYearScores)
    For Each item In allScores.OrderBy(Function(s) s)
        txtLog.WriteLine(item)
    Next
End Sub
```

Sample of C# Code

```
private void unionToolStripMenuItem_Click(object sender, EventArgs e)
{
    int[] lastYearScores = { 88, 56, 23, 99, 65, 56 };
    int[] thisYearScores = { 93, 78, 23, 99, 90, 99 };
    var allScores = lastYearScores.Union(thisYearScores);
    foreach (var item in allScores.OrderBy(s=>s))
    {
        txtLog.WriteLine(item);
    }
}
```

The result:

```
23
56
65
78
88
90
93
99
```

WHERE

The *Where* extension method enables you to filter a source sequence. This method accepts a predicate lambda expression. When working with relational databases, this extension method typically translates to a SQL WHERE clause. The following sample code demonstrates the use of the *Where* method with a sequence of cars that are filtered on *Make* being equal to *Ford*.

Sample of Visual Basic Code

```
Private Sub whereToolStripMenuItem_Click(ByVal sender As System.Object, _
        ByVal e As System.EventArgs) Handles whereToolStripMenuItem.Click
    Dim cars = GetCars()
    For Each myCar In cars.Where(Function(c) c.Make = "Ford")
        txtLog.WriteLine("Car VIN:{0}, Make:{1}, Model:{2} Year:{3}", _
            myCar.VIN, myCar.Make, myCar.Model, myCar.Year)
    Next
End Sub
```

Sample of C# Code

```
private void whereToolStripMenuItem_Click(object sender, EventArgs e)
{
    var cars = GetCars();
    foreach (var myCar in cars.Where(c => c.Make == "Ford"))
    {
        txtLog.WriteLine("Car VIN:{0}, Make:{1}, Model:{2} Year:{3}",
            myCar.VIN, myCar.Make, myCar.Model, myCar.Year);
    }
}
```

The result:

```
Car VIN:ABC123, Make:Ford, Model:F-250 Year:2000
Car VIN:DEF456, Make:Ford, Model:F-150 Year:1998
```

ZIP

The *Zip* extension method merges two sequences. This is neither *Union* nor *Concat* because the resulting element count is equal to the minimum count of the two sequences. Element 1 of sequence 1 is mated to element 1 of sequence 2, and you provide a lambda expression to define which kind of output to create based on this mating. Element 2 of sequence 1 is then mated to element 2 of sequence 2 and so on until one of the sequences runs out of elements.

The following sample code starts with a number sequence, using a starting value of *1* and an ending value of *1000*. The second sequence is a collection of *Car* objects. The *Zip* extension method produces an output collection of anonymous objects that contain the index number and the car.

Sample of Visual Basic Code

```
Private Sub zipToolStripMenuItem_Click(ByVal sender As System.Object, _
        ByVal e As System.EventArgs) Handles zipToolStripMenuItem.Click
    Dim numbers = Enumerable.Range(1, 1000)
    Dim cars = GetCars()
    Dim zip = numbers.Zip(cars, _
            Function(i, c) New With {.Number = i, .CarMake = c.Make})
    For Each item In zip
        txtLog.WriteLine("Number:{0} CarMake:{1}", item.Number, item.CarMake)
    Next
End Sub
```

Sample of C# Code

```
private void zipToolStripMenuItem_Click(object sender, EventArgs e)
{
    var numbers = Enumerable.Range(1, 1000);
    var cars = GetCars();
    var zip = numbers.Zip(cars, (i, c) => new {
                Number = i, CarMake = c.Make });
    foreach (var item in zip)
    {
        txtLog.WriteLine("Number:{0} CarMake:{1}", item.Number, item.CarMake);
    }
}
```

The result:

```
Number:1 CarMake:Ford
Number:2 CarMake:BMW
Number:3 CarMake:Audi
Number:4 CarMake:VW
Number:5 CarMake:Ford
```

The ending range of *1000* on the first sequence was somewhat arbitrary but is noticeably higher than the quantity of *Car* objects in the second sequence. When the second sequence ran out of elements, the *Zip* method stopped producing output. If the ending range of the first sequence was set to 3, only three elements would be output because the first sequence would run out of elements.

PRACTICE Working with LINQ-Enabling Features

In this practice, you create a simple Vehicle Web application with a vehicles collection that is a generic list of *Vehicle*. This list will be populated with some vehicles to use object initializers, collection initializers, implicitly typed local variables, query extension methods, lambda expressions, and anonymous types.

This practice is intended to focus on the features that have been defined in this lesson, so the GUI will be minimal.

If you encounter a problem completing an exercise, the completed projects can be installed from the Code folder on the companion CD.

EXERCISE Create a Web Application with a GUI

In this exercise, you create a Web Application project and add controls to the main Web form to create the graphical user interface.

1. In Visual Studio .NET 2010, choose File | New | Project.
2. Select your desired programming language and then select the ASP.NET Web Application template. For the project name, enter **VehicleProject**. Be sure to select a desired location for this project.
3. For the solution name, enter **VehicleSolution**. Be sure Create Directory For Solution is selected and then click OK.

   After Visual Studio .NET creates the project, the home page, Default.aspx, will be displayed.

> ***NOTE*** **MISSING THE PROMPT FOR THE LOCATION**
>
> **If you don't see a prompt for the location, it's because your Visual Studio .NET settings, set up to enable you to abort the project, automatically remove all files from your hard drive. To select a location, simply choose File | Save All after the project has been created. To change this setting, choose Tools | Options | Projects And Solutions | Save New Projects When Created. When this option is selected, you are prompted for a location when you create the project.**

   There are two content tags, one called *HeaderContent* and one called *BodyContent*. The *BodyContent* tag currently has default markup to display a welcome message and help link.
4. If you haven't seen this Web Application template before, choose Debug | Start Debugging to build and run this Web application so that you can see the default template. After running the application, go back to the Default.aspx markup.
5. Delete the markup that's in the *BodyContent* tag.
6. Populate the *BodyContent* tag with the following markup, which will display filter and sort criteria and provide an execute button and a grid to display vehicles.

   ASPX Markup

```
<asp:Content ID="BodyContent" runat="server" ContentPlaceHolderID="MainContent">
    <asp:Label ID="lblVin" runat="server" Width="100px" Text="VIN: "></asp:Label>
    <asp:TextBox ID="txtVin" runat="server"></asp:TextBox>
    <br />
    <asp:Label ID="lblMake" runat="server" Width="100px" Text="Make: "></
```

```
asp:Label>
    <asp:TextBox ID="txtMake" runat="server"></asp:TextBox>
    <br />
    <asp:Label ID="lblModel" runat="server" Width="100px" Text="Model: "></
asp:Label>
    <asp:TextBox ID="txtModel" runat="server"></asp:TextBox>
    <br />
    <asp:Label ID="lblYear" runat="server" Width="100px" Text="Year: "></
asp:Label>
    <asp:DropDownList ID="ddlYear" runat="server">
        <asp:ListItem Text="All Years" Value="0" />
        <asp:ListItem Text="> 1995" Value="1995" />
        <asp:ListItem Text="> 2000" Value="2000" />
        <asp:ListItem Text="> 2005" Value="2005" />
    </asp:DropDownList>
    <br />
    <asp:Label ID="lblCost" runat="server" Width="100px" Text="Cost: "></
asp:Label>
    <asp:DropDownList ID="ddlCost" runat="server">
        <asp:ListItem Text="Any Cost" Value="0" />
        <asp:ListItem Text="> 5000" Value="5000" />
        <asp:ListItem Text="> 20000" Value="20000" />
    </asp:DropDownList>
    <br />
    <asp:Label ID="lblSort" runat="server" Width="100px" Text="Sort Order: "></
asp:Label>
    <asp:DropDownList ID="ddlSort" runat="server">
        <asp:ListItem Text="" />
        <asp:ListItem Text="VIN" />
        <asp:ListItem Text="Make" />
        <asp:ListItem Text="Model" />
        <asp:ListItem Text="Year" />
        <asp:ListItem Text="Cost" />
    </asp:DropDownList>
    <br />
    <br />
    <asp:Button ID="btnExecute" runat="server" Text="Execute" />
    <br />
    <asp:GridView ID="gvVehicles" runat="server">
    </asp:GridView>
</asp:Content>
```

If you click the Design tab (bottom left), you should see the rendered screen as shown in Figure 3-5.

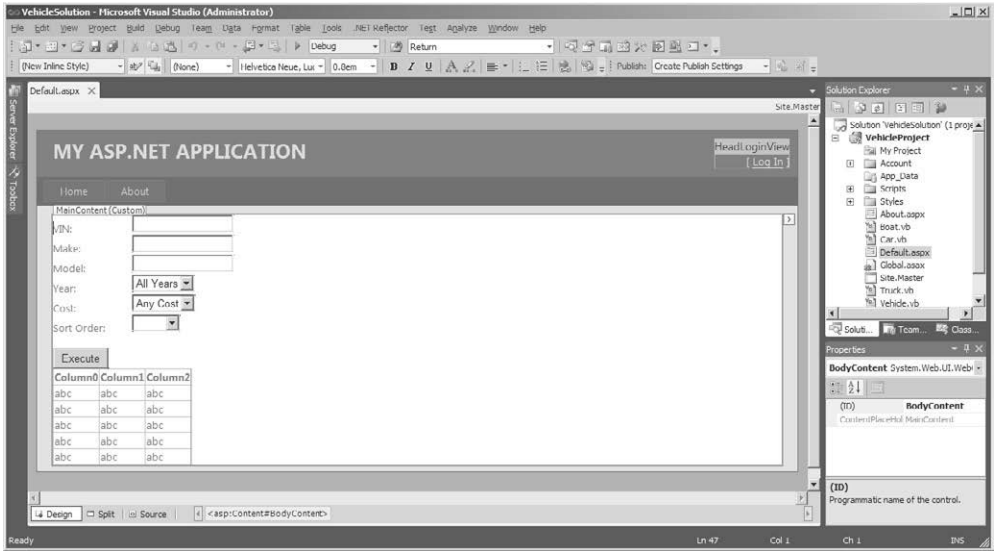

FIGURE 3-5 This is the rendered screen showing filter and sort settings.

7. Right-click the Design or Markup window and click View Code. This takes you to the code-behind page. There is already a *Page_Load* event handler method.

8. Before adding code to the *Page_Load* method, you must add some classes to the project. In Solution Explorer, right-click the VehicleProject icon, click Add, and then click Class. Name the class **Vehicle** and click Add. Add the following code to this class.

Sample of Visual Basic Code

```
Public Class Vehicle
    Public Property VIN As String
    Public Property Make As String
    Public Property Model As String
    Public Property Year As Integer
    Public Property Cost As Decimal
End Class
```

Sample of C# Code

```
public class Vehicle
{
    public string VIN { get; set; }
    public string Make { get; set; }
    public string Model { get; set; }
    public int Year { get; set; }
    public decimal Cost { get; set; }
}
```

9. Add another class, named **Car**, that inherits from *Vehicle*, as shown in the following code sample:

Sample of Visual Basic Code

```
Public Class Car
    Inherits Vehicle
End Class
```

Sample of C# Code

```
public class Car : Vehicle
{
}
```

10. Add another class, named **Truck**, that inherits from *Vehicle,* as shown in the following code sample:

Sample of Visual Basic Code

```
Public Class Truck
    Inherits Vehicle
End Class
```

Sample of C# Code

```
public class Truck : Vehicle
{
}
```

11. Add another class, named **Boat**, that inherits from *Vehicle*, as shown in the following code sample:

Sample of Visual Basic Code

```
Public Class Boat
    Inherits Vehicle
End Class
```

Sample of C# Code

```
public class Boat : Vehicle
{
}
```

12. Above *Page_Load* (at the class level), add code to declare a variable called **vehicles** and instantiate it as a new **List Of Vehicles**. Your code should look like the following sample.

Sample of Visual Basic Code

```
Private Shared Vehicles As New List(Of Vehicle)
```

Sample of C# Code

```
private List<Vehicle> vehicles = new List<Vehicle>();
```

13. In the *Page_Load* method, add code to create a generic list of *Vehicle*. Populate the list with ten vehicles, which will give you something with which to experiment in this practice. The *Page_Load* method should look like the following example:

Sample of Visual Basic Code

```
Protected Sub Page_Load(ByVal sender As Object, _
           ByVal e As System.EventArgs) Handles Me.Load
    If (Vehicles.Count = 0) Then
        Vehicles.Add(New Truck With {.VIN = "AAA123", .Make = "Ford", _
                        .Model = "F-250", .Cost = 2000, .Year = 1998})
        Vehicles.Add(New Truck With {.VIN = "ZZZ123", .Make = "Ford", _
                        .Model = "F-150", .Cost = 10000, .Year = 2005})
        Vehicles.Add(New Car With {.VIN = "FFF123", .Make = "VW", _
                        .Model = "Bug", .Cost = 2500, .Year = 1997})
        Vehicles.Add(New Boat With {.VIN = "LLL123", .Make = "SeaRay", _
                        .Model = "Signature", .Cost = 12000, .Year = 1995})
        Vehicles.Add(New Car With {.VIN = "CCC123", .Make = "BMW", _
                        .Model = "Z-3", .Cost = 21000, .Year = 2005})
        Vehicles.Add(New Car With {.VIN = "EEE123", .Make = "Ford", _
                        .Model = "Focus", .Cost = 15000, .Year = 2008})
        Vehicles.Add(New Boat With {.VIN = "QQQ123", .Make = "ChrisCraft", _
                        .Model = "BowRider", .Cost = 102000, .Year = 1945})
        Vehicles.Add(New Truck With {.VIN = "PPP123", .Make = "Ford", _
                        .Model = "F-250", .Cost = 1000, .Year = 1980})
        Vehicles.Add(New Car With {.VIN = "TTT123", .Make = "Dodge", _
                        .Model = "Viper", .Cost = 95000, .Year = 2007})
        Vehicles.Add(New Car With {.VIN = "DDD123", .Make = "Mazda", _
                        .Model = "Miata", .Cost = 20000, .Year = 2005})
    End If
End Sub
```

Sample of C# Code

```
protected void Page_Load(object sender, EventArgs e)
{
    if (vehicles.Count == 0)
    {
        vehicles.Add(new Truck  {VIN = "AAA123", Make = "Ford",
                        Model = "F-250", Cost = 2000, Year = 1998});
        vehicles.Add(new Truck  {VIN = "ZZZ123", Make = "Ford",
                        Model = "F-150", Cost = 10000, Year = 2005});
        vehicles.Add(new Car  {VIN = "FFF123", Make = "VW",
                        Model = "Bug", Cost = 2500, Year = 1997});
        vehicles.Add(new Boat  {VIN = "LLL123", Make = "SeaRay",
                        Model = "Signature", Cost = 12000, Year = 1995});
        vehicles.Add(new Car  {VIN = "CCC123", Make = "BMW",
                        Model = "Z-3", Cost = 21000, Year = 2005});
        vehicles.Add(new Car  {VIN = "EEE123", Make = "Ford",
                        Model = "Focus", Cost = 15000, Year = 2008});
        vehicles.Add(new Boat  {VIN = "QQQ123", Make = "ChrisCraft",
                        Model = "BowRider", Cost = 102000, Year = 1945});
        vehicles.Add(new Truck  {VIN = "PPP123", Make = "Ford",
                        Model = "F-250", Cost = 1000, Year = 1980});
        vehicles.Add(new Car  {VIN = "TTT123", Make = "Dodge",
                        Model = "Viper", Cost = 95000, Year = 2007});
```

```
        vehicles.Add(new Car  {VIN = "DDD123", Make = "Mazda",
                        Model = "Miata", Cost = 20000, Year = 2005});
    }
}
```

14. Under the code you just added into the *Page_Load* method, insert code to filter the list of vehicles based on the data input. Use *method chaining* to create one statement that puts together all the filtering, as shown in the following code sample:

Sample of Visual Basic Code

```
Dim result = Vehicles _
        .Where(Function(v) v.VIN.StartsWith(txtVin.Text)) _
        .Where(Function(v) v.Make.StartsWith(txtMake.Text)) _
        .Where(Function(v) v.Model.StartsWith(txtModel.Text)) _
        .Where(Function(v) v.Cost > Decimal.Parse(ddlCost.SelectedValue)) _
        .Where(Function(v) v.Year > Integer.Parse(ddlYear.SelectedValue))
```

Sample of C# Code

```
var result = vehicles
    .Where(v => v.VIN.StartsWith(txtVin.Text))
    .Where(v => v.Make.StartsWith(txtMake.Text))
    .Where(v => v.Model.StartsWith(txtModel.Text))
    .Where(v => v.Cost > Decimal.Parse(ddlCost.SelectedValue))
    .Where(v => v.Year > int.Parse(ddlYear.SelectedValue));
```

15. Under the code you just added into the *Page_Load* method, add code to perform a sort of the results. This code calls a *SetOrder* method that will be created in the next step. You code should look like the following:

Sample of Visual Basic Code

```
result = SetOrder(ddlSort.SelectedValue, result)
```

Sample of C# Code

```
result = SetOrder(ddlSort.SelectedValue, result);
```

16. Add the *SetOrder* method, which has code to add an *OrderBy* query extension method based on the selection passed into this method. Your code should look like the following:

Sample of Visual Basic Code

```
Private Function SetOrder(ByVal order As String, _
        ByVal query As IEnumerable(Of Vehicle)) As IEnumerable(Of Vehicle)
    Select Case order
        Case "VIN"
            Return query.OrderBy(Function(v) v.VIN)
        Case "Make"
            Return query.OrderBy(Function(v) v.Make)
        Case "Model"
            Return query.OrderBy(Function(v) v.Model)
        Case "Year"
            Return query.OrderBy(Function(v) v.Year)
        Case "Cost"
            Return query.OrderBy(Function(v) v.Cost)
```

```
        Case Else
            Return query
    End Select
End Function
```

Sample of C# Code

```
private IEnumerable<Vehicle> SetOrder(string order,
        IEnumerable<Vehicle> query)
{
    switch (order)
    {
        case "VIN":
            return query.OrderBy(v => v.VIN);
        case "Make":
            return query.OrderBy(v => v.Make);
        case "Model":
            return query.OrderBy(v => v.Model);
        case "Year":
            return query.OrderBy(v => v.Year);
        case "Cost":
            return query.OrderBy(v => v.Cost);
        default:
            return query;
    }
}
```

17. Finally, add code into the bottom of the *Page_Load* method to select an anonymous type that includes an index and all the properties in the *Vehicle* class and bind the result to *gvVehicles*. You code should look like the following example:

Sample of Visual Basic Code

```
gvVehicles.DataSource = result.Select(Function(v, i) New With
    {.Index = i, v.VIN, v.Make, v.Model, v.Year, v.Cost})
gvVehicles.DataBind()
```

Sample of C# Code

```
gvVehicles.DataSource = result.Select((v, i)=> new
    {Index = i, v.VIN, v.Make, v.Model, v.Year, v.Cost});
gvVehicles.DataBind();
```

18. Choose Build | Build Solution to build the application. If you have errors, you can double-click the error to go to the error line and correct.

19. Choose Debug | Start Debugging to run the application. When the application starts, you should see a Web page with your GUI controls that enables you to specify filter and sort criteria. If you type the letter **F** into the Make text box and click Execute, the grid will be populated only with items that begin with F. If you set the sort order and click the Execute button again, you will see the sorted results.

Lesson Summary

This lesson provided detailed information about the features that comprise LINQ.

- Object initializers enable you to initialize public properties and fields without creating an explicit constructor.
- Implicitly typed local variables enable you to declare a variable without specifying its type, and the compiler will infer the type for you.
- In many cases, using implicitly typed local variables is an option, but, when working with anonymous types, it's a requirement.
- Anonymous types enable you to create a type inline. This enables you to group data without creating a class.
- Lambda expressions provide a much more abbreviated syntax than a method or anonymous method and can be used wherever a delegate is expected.
- Extension methods enable you to add methods to a type even when you don't have the source code for the type.
- Extension methods enable you to create concrete methods on interfaces; that is, all types that implement the interface will get these methods.
- Query extension methods are extension methods primarily implemented on the generic *IEnumerable* interface.
- The *Enumerable* class contains the query extension methods and static methods called *Empty*, *Range*, and *Repeat*.

Lesson Review

You can use the following questions to test your knowledge of the information in Lesson 1, "Understanding LINQ." The questions are also available on the companion CD if you prefer to review them in electronic form.

> ***NOTE*** **ANSWERS**
>
> **Answers to these questions and explanations of why each answer choice is correct or incorrect are located in the "Answers" section at the end of the book.**

1. To which of the following types can you add an extension method? (Each correct answer presents a complete solution. Choose five.)
   - **A.** *Class*
   - **B.** *Structure* (*C# struct*)
   - **C.** *Module* (*C# static class*)
   - **D.** *Enum*
   - **E.** *Interface*
   - **F.** *Delegate*

2. You want to page through an element sequence, displaying ten elements at a time, until you reach the end of the sequence. Which query extension method can you use to accomplish this? (Each correct answer presents part of a complete solution. Choose two.)

   A. *Skip*

   B. *Except*

   C. *SelectMany*

   D. *Take*

3. You have executed the *Where* query extension method on your collection, and it returned *IEnumerable* of *Car*, but you want to assign this to a variable whose type is *List Of Car*. How can you convert the *IEnumerable* of *Car* to *List Of Car*?

   A. Use *CType* (C# *cast*).

   B. It can't be done.

   C. Use the *ToList*() query extension method.

   D. Just make the assignment.

Lesson 2: Using LINQ Queries

The previous sections covered object initializers, implicitly typed local variables, anonymous types, lambda expressions, and extension methods. These features were created to support the implementation of LINQ. Now that you've seen all these, look at how LINQ is *language integrated*.

After this lesson, you will be able to:

- Identify the LINQ keywords.
- Create a LINQ query that provides filtering.
- Create a LINQ query that provides sorted results.
- Create a LINQ query to perform an inner join on two element sequences.
- Create a LINQ query to perform an outer join on two element sequences.
- Implement grouping and aggregation in a LINQ query.
- Create a LINQ query that defines addition loop variables using the *let* keyword.
- Create a LINQ query that implements paging.

Estimated lesson time: 60 minutes

Syntax-Based and Method-Based Queries

For basic queries, using LINQ in Visual Basic or C# is very easy and intuitive because both languages provide keywords that map directly to features that have been added through extension methods. The benefit is that you can write typed queries in a very SQL-like way, getting IntelliSense support all along the way.

In the following scenario, your schedule contains a list of days when you are busy, and you want to find out whether you are busy on a specific day. The following code demonstrates the implementation of a LINQ query to discover this.

Sample of Visual Basic Code

```
Private Function GetDates() As List(Of DateTime)
   Return New List(Of DateTime) From
                       {
                          New DateTime(11, 1, 1),
                          New DateTime(11, 2, 5),
                          New DateTime(11, 3, 3),
                          New DateTime(11, 1, 3),
                          New DateTime(11, 1, 2),
                          New DateTime(11, 5, 4),
                          New DateTime(11, 2, 2),
                          New DateTime(11, 7, 5),
                          New DateTime(11, 6, 30),
                          New DateTime(11, 10, 14),
                          New DateTime(11, 11, 22),
```

```
                        New DateTime(11, 12, 1),
                        New DateTime(11, 5, 22),
                        New DateTime(11, 6, 7),
                        New DateTime(11, 1, 4)
                    }
End Function
Private Sub BasicQueriesToolStripMenuItem_Click( _
        ByVal sender As System.Object, _
        ByVal e As System.EventArgs) _
        Handles BasicQueriesToolStripMenuItem.Click

    Dim schedule = GetDates()
    Dim areYouAvailable = new DateTime(11, 7, 10)

    Dim busy = From d In schedule
               Where d = areYouAvailable
               Select d

    For Each busyDate In busy
        txtLog.WriteLine("Sorry, but I am busy on {0:MM/dd/yy}", busyDate)
    Next
End Sub
```

Sample of C# Code

```
private List<DateTime> GetDates()
{
    return new List<DateTime>
                {
                 new DateTime(11, 1, 1),
                 new DateTime(11, 2, 5),
                 new DateTime(11, 3, 3),
                 new DateTime(11, 1, 3),
                 new DateTime(11, 1, 2),
                 new DateTime(11, 5, 4),
                 new DateTime(11, 2, 2),
                 new DateTime(11, 7, 5),
                 new DateTime(11, 6, 30),
                 new DateTime(11, 10, 14),
                 new DateTime(11, 11, 22),
                 new DateTime(11, 12, 1),
                 new DateTime(11, 5, 22),
                 new DateTime(11, 6, 7),
                 new DateTime(11, 1, 4)
                };
}
private void basicLINQToolStripMenuItem_Click(object sender, EventArgs e)
{
    var schedule = GetDates();
    var areYouAvailable = new DateTime(11,7, 5);

    var busy = from d in schedule
               where d == areYouAvailable
               select d;

    foreach(var busyDate in busy)
```

```
    {
        txtLog.WriteLine("Sorry, but I am busy on {0:MM/dd/yy}", busyDate);
    }
}
```

In the sample code, a LINQ query filtered the data, which returned an *IEnumerable<DateTime>* object as the result. Is there a simpler way to perform this query? You could argue that using the *Where* extension method would save some coding and would be simpler, as shown in this method-based code sample:

Sample of Visual Basic Code

```
Private Sub MethodbasedQueryToolStripMenuItem_Click( _
       ByVal sender As System.Object, _
       ByVal e As System.EventArgs) _
       Handles MethodbasedQueryToolStripMenuItem.Click

   Dim schedule = GetDates()
   Dim areYouAvailable = New DateTime(11,7,5)

   For Each busyDate In schedule.Where(Function(d) d = areYouAvailable)
       txtLog.WriteLine("Sorry, but I am busy on {0:MM/dd/yy}", busyDate)
   Next
End Sub
```

Sample of C# Code

```
private void methodbasedQueryToolStripMenuItem_Click(object sender, EventArgs e)
{
    var schedule = GetDates();
    var areYouAvailable = new DateTime(11,7,5);

    foreach (var busyDate in schedule.Where(d=>d==areYouAvailable))
    {
        txtLog.WriteLine("Sorry, but I am busy on {0:MM/dd/yy}", busyDate);
    }
}
```

This example eliminates the LINQ query and adds the *Where* extension method in the loop. This code block is smaller and more concise, but which is more readable? Decide for yourself. For a small query such as this, the extension method might be fine, but for larger queries, you probably will find it better to use the LINQ query. Performance is the same because both queries do the same thing.

Only a small subset of the query extension methods map to language keywords, so typically you will find yourself mixing LINQ queries with extension methods, as shown in the following rewrite of the previous examples:

Sample of Visual Basic Code

```
Private Sub MixingLINQAndMethodsToolStripMenuItem_Click( _
        ByVal sender As System.Object, _
        ByVal e As System.EventArgs) _
        Handles MixingLINQAndMethodsToolStripMenuItem.Click
    Dim schedule = GetDates()
```

```
    Dim areYouAvailable = New DateTime(11, 7, 5)

    Dim count = (From d In schedule
                 Where d = areYouAvailable
                 Select d).Count()

    If count > 0 Then
      txtLog.WriteLine("Sorry, but I am busy on {0:MM/dd/yy}", areYouAvailable)
    Else
      txtLog.WriteLine("Yay! I am available on {0:MM/dd/yy}", areYouAvailable)
    End If
End Sub
```

Sample of C# Code

```
private void mixingLINQAndMethodsToolStripMenuItem_Click(
    object sender, EventArgs e)
{
    var schedule = GetDates();
    var areYouAvailable = new DateTime(11, 7, 5);

    var count = (from d in schedule
                 where d == areYouAvailable
                 select d).Count();
    if (count > 0)
        txtLog.WriteLine("Sorry, but I am busy on {0:MM/dd/yy}",
            areYouAvailable);
    else
        txtLog.WriteLine("Yay! I am available on {0:MM/dd/yy}",
            areYouAvailable);
}
```

In the previous example, the *Count* extension method eliminates the *foreach* loop. In this example, an *if/then/else* statement is added to show availability. Also, parentheses are added to place the call to the *Count* method after the *select* clause.

LINQ Keywords

The LINQ-provided keywords can make your LINQ queries look clean and simple. Table 3-1 provides the list of available keywords, with a short description of each. Many of these keywords are covered in more detail in this section.

TABLE 3-1 Visual Basic and C# LINQ Keywords

KEYWORD	DESCRIPTION
from	Specifies a data source and a range variable
where	Filters source elements based on one or more Boolean expressions
select	Specifies the type and shape the elements in the returned sequence have when the query is executed

group	Groups query results according to a specified key value
into	Provides an identifier that can serve as a reference to the results of a *join*, *group*, or *select* clause
orderby (Visual Basic: *Order By*)	Sorts query results in ascending or descending order
join	Joins two data sources based on an equality comparison between two specified matching criteria
let	Introduces a range variable to store subexpression results in a query expression
in	Contextual keyword in a *from* or *join* clause to specify the data source
on	Contextual keyword in a *join* clause to specify the join criteria
equals	Contextual keyword in a *join* clause to join two sources
by	Contextual keyword in a *group* clause to specify the grouping criteria
ascending	Contextual keyword in an *orderby* clause
descending	Contextual keyword in an *orderby* clause

In addition to the keywords listed in Table 3-1, the Visual Basic team provided keywords that C# did not implement. These keywords are shown in Table 3-2 with a short description of each.

TABLE 3-2 Visual Basic Keywords That Are Not Implemented in C#

KEYWORD	DESCRIPTION
Distinct	Filters duplicate elements
Skip/Skip While	Jumps over elements before returning results
Take/Take While	Provides a means to limit how many elements will be retrieved
Aggregate	Includes aggregate functions in your queries
Into	Contextual keyword in the *Aggregate* clause that specifies what to do with the result of the aggregate
All	Contextual keyword in the *Aggregate* clause that determines whether all elements meet the specified criterion
Any	Contextual keyword in the *Aggregate* clause that determines whether any of the elements meet the specified criterion
Average	Contextual keyword in the *Aggregate* clause that calculates the average value

Count	Contextual keyword in the *Aggregate* clause that provides the count of elements that meet the specified criterion
Group	Contextual keyword in the *Aggregate* clause that provides access to the results of a *group by* or *group join* clause
LongCount	Contextual keyword in the *Aggregate* clause that provides the count (*as long*) of elements that meet the specified criterion
Max	Contextual keyword in the *Aggregate* clause that provides the maximum value
Min	Contextual keyword in the *Aggregate* clause that provides the minimum value
Sum	Contextual keyword in the *Aggregate* clause that provides the sum of the elements

All the query extension methods are available in both languages even if there isn't a language keyword mapping to the query extension method.

Projections

Projections enable you to transform the output of your LINQ query by using named or anonymouns types. The following code example demonstrates projections in a LINQ query by using anonymous types.

Sample of Visual Basic Code

```
Private Sub LINQProjectionsToolStripMenuItem_Click(ByVal sender As System.Object, _
        ByVal e As System.EventArgs) Handles LINQProjectionsToolStripMenuItem.Click

    Dim cars = GetCars()
    Dim vinsAndMakes = From c In cars
        Select New With
                       {
                           c.VIN,
                           .CarModel = c.Make
                       }
    For Each item In vinsAndMakes
        txtLog.WriteLine("VIN:{0} Make:{1}", item.VIN, item.CarModel)
    Next
End Sub
```

Sample of C# Code

```
private void lINQProjectionsToolStripMenuItem_Click(object sender, EventArgs e)
{
    var cars = GetCars();
    var vinsAndMakes = from c in cars
                       select new { c.VIN, CarModel = c.Model };
    foreach (var item in vinsAndMakes)
    {
        txtLog.WriteLine("VIN:{0} Make:{1}", item.VIN, item.CarModel);
```

```
    }
}
```

Using the *Let* Keyword to Help with Projections

You can use the *let* keyword to create a temporary variable within the LINQ query. Think of the *let* keyword as a variant of the *select* keyword used within the query. The following code sample shows how the *let* keyword can help with filtering and shaping the data.

Sample of Visual Basic Code

```
Private Sub LINQLetToolStripMenuItem_Click(ByVal sender As System.Object, _
        ByVal e As System.EventArgs) Handles LINQLetToolStripMenuItem.Click
    Dim cars = GetCars()
    Dim vinsAndMakes = From c In cars
                       Let makeModel = c.Make & " " & c.Model
                       Where makeModel.Contains("B")
                       Select New With
                                    {
                                        c.VIN,
                                        .MakeModel = makeModel
                                    }
For Each item In vinsAndMakes
    txtLog.WriteLine("VIN:{0} Make and Model:{1}", item.VIN, item.MakeModel)
    Next
End Sub
```

Sample of C# Code

```
private void lINQLetToolStripMenuItem_Click(object sender, EventArgs e)
{
    var cars = GetCars();
    var vinsAndMakes = from c in cars
                       let makeModel = c.Make + " " + c.Model
                       where makeModel.Contains('B')
                       select new { c.VIN, MakeModel=makeModel };
    foreach (var item in vinsAndMakes)
    {
      txtLog.WriteLine("VIN:{0} Make and Model:{1}", item.VIN, item.MakeModel);
    }
}
```

The result:

```
VIN:DEF123 Make and Model:BMW Z-3
VIN:HIJ123 Make and Model:VW Bug
```

Specifying a Filter

Both C# and Visual Basic have the *where* keyword that maps directly to the *Where* query extension method. You can specify a predicate (an expression that evaluates to a Boolean value) to determine the elements to be returned. The following code sample demonstrates the *where* clause with a *yearRange* variable being used as a parameter into the query.

Sample of Visual Basic Code

```
Private Sub LINQWhereToolStripMenuItem_Click(ByVal sender As System.Object, _
        ByVal e As System.EventArgs) Handles LINQWhereToolStripMenuItem.Click
    Dim yearRange = 2000
    Dim cars = GetCars()
    Dim oldCars = From c In cars
                  Where c.Year < yearRange
                  Select c

    For Each myCar In oldCars
        txtLog.WriteLine("Car VIN:{0}, Make:{1}, Model:{2} Year:{3}",
            myCar.VIN, myCar.Make, myCar.Model, myCar.Year)
    Next
End Sub
```

Sample of C# Code

```
private void lINQWhereToolStripMenuItem_Click(object sender, EventArgs e)
{
    int yearRange = 2000;
    var cars = GetCars();
    var oldCars = from c in cars
                  where c.Year < yearRange
                  select c;
    foreach (var myCar in oldCars)
    {
        txtLog.WriteLine("Car VIN:{0}, Make:{1}, Model:{2} Year:{3}",
            myCar.VIN, myCar.Make, myCar.Model, myCar.Year);
    }
}
```

The result:

```
Car VIN:HIJ123, Make:VW, Model:Bug Year:1956
Car VIN:DEF456, Make:Ford, Model:F-150 Year:1998
```

Specifying a Sort Order

It's very easy to sort using a LINQ query. The *orderby* keyword enables you to sort in ascending or descending order. In addition, you can sort on multiple properties to perform a compound sort. The following code sample shows the sorting of cars by *Make* ascending and then by *Model* descending.

Sample of Visual Basic Code

```
Private Sub LINQSortToolStripMenuItem_Click(ByVal sender As System.Object, _
        ByVal e As System.EventArgs) Handles LINQSortToolStripMenuItem.Click
    Dim cars = GetCars()
    Dim sorted = From c In cars
                 Order By c.Make Ascending, c.Model Descending
                 Select c

    For Each myCar In sorted
        txtLog.WriteLine("Car VIN:{0}, Make:{1}, Model:{2} Year:{3}",
            myCar.VIN, myCar.Make, myCar.Model, myCar.Year)
```

```
    Next
End Sub
```

Sample of C# Code

```
private void lINQSortToolStripMenuItem_Click(object sender, EventArgs e)
{
    var cars = GetCars();
    var sorted = from c in cars
                 orderby c.Make ascending, c.Model descending
                 select c;
    foreach (var myCar in sorted)
    {
        txtLog.WriteLine("Car VIN:{0}, Make:{1}, Model:{2} Year:{3}",
            myCar.VIN, myCar.Make, myCar.Model, myCar.Year);
    }
}
```

The result:

```
Car VIN:ABC456, Make:Audi, Model:TT Year:2008
Car VIN:DEF123, Make:BMW, Model:Z-3 Year:2005
Car VIN:ABC123, Make:Ford, Model:F-250 Year:2000
Car VIN:DEF456, Make:Ford, Model:F-150 Year:1998
Car VIN:HIJ123, Make:VW, Model:Bug Year:1956
```

Paging

The ability to look at data one page at a time is always a requirement when a large amount of data is being retrieved. LINQ simplifies this task with the *Skip* and *Take* extension methods. In addition, Visual Basic offers these query extension methods as keywords.

The following code example retrieves 25 rows of data and then provides paging capabilities to enable paging ten rows at a time.

EXAM TIP

For the exam, be sure that you fully understand how to perform paging.

Sample of Visual Basic Code

```
Private Sub LINQPagingToolStripMenuItem_Click(ByVal sender As System.Object, _
        ByVal e As System.EventArgs) Handles LINQPagingToolStripMenuItem.Click
    Dim pageSize = 10

    'create 5 copies of the cars - total 25 rows
    Dim cars = Enumerable.Range(1, 5) _
            .SelectMany(Function(i) GetCars() _
                .Select(Function(c) New With _
                    {.BatchNumber = i, c.VIN, c.Make, c.Model, c.Year}))

    'calculate page count
    Dim pageCount = (cars.Count() / pageSize)
    If (pageCount * pageSize < cars.Count()) Then pageCount += 1
```

```
    For i = 0 To pageCount - 1
        txtLog.WriteLine("-----Printing Page {0}------", i)
        'Dim currentPage = cars.Skip(i * pageSize).Take(pageSize)
        Dim currentPage = From c In cars
                          Skip (i * pageSize)
                          Take pageSize
                          Select c
        For Each myCar In currentPage
            txtLog.WriteLine("#{0} Car VIN:{1}, Make:{2}, Model:{3} Year:{4}", _
              myCar.BatchNumber, myCar.VIN, myCar.Make, myCar.Model, myCar.Year)
        Next
    Next
End Sub
```

Sample of C# Code

```
private void lINQPagingToolStripMenuItem_Click(object sender, EventArgs e)
{
    int pageSize = 10;

    //create 5 copies of the cars - total 25 rows
    var cars = Enumerable.Range(1,5)
        .SelectMany(i=>GetCars()
            .Select(c=>(new {BatchNumber=i, c.VIN, c.Make, c.Model, c.Year})));

    //calculate page count
    int pageCount = (cars.Count() / pageSize);
    if (pageCount * pageSize < cars.Count()) pageCount++;

    for(int i=0; i < pageCount; i++)
    {
        txtLog.WriteLine("-----Printing Page {0}------", i);
        var currentPage = cars.Skip(i * pageSize).Take(pageSize);

        foreach (var myCar in currentPage)
        {
            txtLog.WriteLine("#{0} Car VIN:{1}, Make:{2}, Model:{3} Year:{4}",
             myCar.BatchNumber, myCar.VIN, myCar.Make, myCar.Model, myCar.Year);
        }
    }
}
```

The result:

```
-----Printing Page 0------
#1 Car VIN:ABC123, Make:Ford, Model:F-250 Year:2000
#1 Car VIN:DEF123, Make:BMW, Model:Z-3 Year:2005
#1 Car VIN:ABC456, Make:Audi, Model:TT Year:2008
#1 Car VIN:HIJ123, Make:VW, Model:Bug Year:1956
#1 Car VIN:DEF456, Make:Ford, Model:F-150 Year:1998
#2 Car VIN:ABC123, Make:Ford, Model:F-250 Year:2000
#2 Car VIN:DEF123, Make:BMW, Model:Z-3 Year:2005
#2 Car VIN:ABC456, Make:Audi, Model:TT Year:2008
#2 Car VIN:HIJ123, Make:VW, Model:Bug Year:1956
#2 Car VIN:DEF456, Make:Ford, Model:F-150 Year:1998
-----Printing Page 1------
```

```
#3 Car VIN:ABC123, Make:Ford, Model:F-250 Year:2000
#3 Car VIN:DEF123, Make:BMW, Model:Z-3 Year:2005
#3 Car VIN:ABC456, Make:Audi, Model:TT Year:2008
#3 Car VIN:HIJ123, Make:VW, Model:Bug Year:1956
#3 Car VIN:DEF456, Make:Ford, Model:F-150 Year:1998
#4 Car VIN:ABC123, Make:Ford, Model:F-250 Year:2000
#4 Car VIN:DEF123, Make:BMW, Model:Z-3 Year:2005
#4 Car VIN:ABC456, Make:Audi, Model:TT Year:2008
#4 Car VIN:HIJ123, Make:VW, Model:Bug Year:1956
#4 Car VIN:DEF456, Make:Ford, Model:F-150 Year:1998
-----Printing Page 2------
#5 Car VIN:ABC123, Make:Ford, Model:F-250 Year:2000
#5 Car VIN:DEF123, Make:BMW, Model:Z-3 Year:2005
#5 Car VIN:ABC456, Make:Audi, Model:TT Year:2008
#5 Car VIN:HIJ123, Make:VW, Model:Bug Year:1956
#5 Car VIN:DEF456, Make:Ford, Model:F-150 Year:1998
```

This code sample starts by defining the page size as 10. Five copies of the cars are then created, which yields 25 cars. The five copies are created by using the *Enumerable* class to generate a range of values, 1 to 5. Each of these values is used with the *SelectMany* query extension method to create a copy of the cars. Calculating the page count is accomplished by dividing the count of the cars by the page size, but if there is a remainder, the page count is incremented. Finally, a *for* loop creates a query for each of the pages and then prints the current page.

In the Visual Basic example, the query for the page was written first to match the C# version, but that code is commented out and the query is rewritten using the Visual Basic *Skip* and *Take* keywords.

Joins

When working with databases, you commonly want to combine data from multiple tables to produce a merged result set. LINQ enables you to join two generic *IEnumerable* element sources, even if these sources are not from a database. There are three types of joins: inner joins, outer joins, and cross joins. Inner joins and outer joins typically match on a foreign key in a child source matching to a unique key in a parent source. This section examines these join types.

Inner Joins

Inner joins produce output only if there is a match between both join sources. In the following code sample, a collection of cars is joined to a collection of repairs, based on the VIN of the car. The resulting output combines some of the car information with some of the repair information.

Sample of Visual Basic Code

```
Public Class Repair
    Public Property VIN() As String
    Public Property Desc() As String
    Public Property Cost As Decimal
```

```
End Class

Private Function GetRepairs() As List(Of Repair)
    Return New List(Of Repair) From
    {
     New Repair With {.VIN = "ABC123", .Desc = "Change Oil", .Cost = 29.99},
     New Repair With {.VIN = "DEF123", .Desc = "Rotate Tires", .Cost = 19.99},
     New Repair With {.VIN = "HIJ123", .Desc = "Replace Brakes", .Cost = 200},
     New Repair With {.VIN = "DEF456", .Desc = "Alignment", .Cost = 30},
     New Repair With {.VIN = "ABC123", .Desc = "Fix Flat Tire", .Cost = 15},
     New Repair With {.VIN = "DEF123", .Desc = "Fix Windshield", .Cost = 420},
     New Repair With {.VIN = "ABC123", .Desc = "Replace Wipers", .Cost = 20},
     New Repair With {.VIN = "HIJ123", .Desc = "Replace Tires", .Cost = 1000},
     New Repair With {.VIN = "DEF456", .Desc = "Change Oil", .Cost = 30}
    }
End Function

Private Sub LINQInnerJoinToolStripMenuItem_Click( _
        ByVal sender As System.Object, _
        ByVal e As System.EventArgs) _
        Handles LINQInnerJoinToolStripMenuItem.Click
    Dim cars = GetCars()
    Dim repairs = GetRepairs()

    Dim carsWithRepairs = From c In cars
                          Join r In repairs
                          On c.VIN Equals r.VIN
                          Order By c.VIN, r.Cost
                          Select New With
                              {
                                  c.VIN,
                                  c.Make,
                                  r.Desc,
                                  r.Cost
                              }

    For Each item In carsWithRepairs
        txtLog.WriteLine("Car VIN:{0}, Make:{1}, Description:{2} Cost:{3:C}",
           item.VIN, item.Make, item.Desc, item.Cost)
    Next
End Sub
```

Sample of C# Code

```
public class Repair
{
    public string VIN { get; set; }
    public string Desc { get; set; }
    public decimal Cost { get; set; }
}

private List<Repair> GetRepairs()
{
    return new List<Repair>
    {
        new Repair {VIN = "ABC123", Desc = "Change Oil", Cost = 29.99m},
```

```
        new Repair {VIN = "DEF123", Desc = "Rotate Tires",  Cost =19.99m},
        new Repair {VIN = "HIJ123", Desc = "Replace Brakes",   Cost = 200},
        new Repair {VIN = "DEF456", Desc = "Alignment", Cost = 30},
        new Repair {VIN = "ABC123", Desc = "Fix Flat Tire", Cost = 15},
        new Repair {VIN = "DEF123", Desc = "Fix Windshield",  Cost =420},
        new Repair {VIN = "ABC123", Desc = "Replace Wipers", Cost = 20},
        new Repair {VIN = "HIJ123", Desc = "Replace Tires",   Cost = 1000},
        new Repair {VIN = "DEF456", Desc = "Change Oil", Cost = 30}
    };
}

private void lINQInnerJoinToolStripMenuItem_Click(object sender, EventArgs e)
{
    var cars = GetCars();
    var repairs = GetRepairs();

    var carsWithRepairs = from c in cars
                          join r in repairs
                          on c.VIN equals r.VIN
                          orderby c.VIN, r.Cost
                          select new
                          {
                              c.VIN,
                              c.Make,
                              r.Desc,
                              r.Cost
                          };
    foreach (var item in carsWithRepairs)
    {
        txtLog.WriteLine("Car VIN:{0}, Make:{1}, Description:{2} Cost:{3:C}",
           item.VIN, item.Make, item.Desc, item.Cost);
    }
}
```

The result:

```
Car VIN:ABC123, Make:Ford, Description:Fix Flat Tire Cost:$15.00
Car VIN:ABC123, Make:Ford, Description:Replace Wipers Cost:$20.00
Car VIN:ABC123, Make:Ford, Description:Change Oil Cost:$29.99
Car VIN:DEF123, Make:BMW, Description:Rotate Tires Cost:$19.99
Car VIN:DEF123, Make:BMW, Description:Fix Windshield Cost:$420.00
Car VIN:DEF456, Make:Ford, Description:Alignment Cost:$30.00
Car VIN:DEF456, Make:Ford, Description:Change Oil Cost:$30.00
Car VIN:HIJ123, Make:VW, Description:Replace Brakes Cost:$200.00
Car VIN:HIJ123, Make:VW, Description:Replace Tires Cost:$1,000.00
```

This example shows the creation of the *Repair* class and the creation of a *GetRepairs* method that returns a generic list of *Repair* objects. Next is the creation of a *cars* variable populated with *Car* objects and a *repairs* variable populated with *Repair* objects. A *carsWithRepairs* variable is created, and the LINQ query is assigned to it. The LINQ query defines an outer element source in the *from* clause and then defines an inner element source using the *join* clause. The *join* clause must be immediately followed by the *on* clause that defines the linking between the two sources. Also, when joining the two sources, you must use the *equals* keyword, not the equals sign. If you need to perform a join on multiple keys, use the Visual Basic

And keyword. In C#, you need to construct an anonymous type for each side of the equals. The LINQ query is sorting by the VIN of the car and the cost of the repair, and the returned elements are of an anonymous type that contains data from each element source.

When looking at the result of this query, the car with the VIN of ABC456 had no repairs, so there was no output for this car. If you want all cars to be in the output even if the car has no repairs, you must perform an outer join.

Another way to perform an inner join is to use the *Join* query extension method, which was covered earlier in this chapter.

Outer Joins

Outer joins produce output for every element in the outer source even if there is no match to the inner source. To perform an outer join by using a LINQ query, use the *into* clause with the *join* clause (Visual Basic *Group Join*). The *into* clause creates an identifier that can serve as a reference to the results of a *join*, *group*, or *select* clause. In this scenario, the *into* clause references the join and is assigned to the variable *temp*. The inner variable *rep* is out of scope, but a new *from* clause is provided to get the variable *r*, which references a repair, from *temp*. The *DefaultIfEmpty* method assigns *null* to *r* if no match can be made to a repair.

Sample of Visual Basic Code

```
Private Sub LINQOuterJoinToolStripMenuItem_Click( _
         ByVal sender As System.Object, _
         ByVal e As System.EventArgs) _
         Handles LINQOuterJoinToolStripMenuItem.Click
    Dim cars = GetCars()
    Dim repairs = GetRepairs()

    Dim carsWithRepairs = From c In cars
                          Group Join rep In repairs
                          On c.VIN Equals rep.VIN Into temp = Group
                             From r In temp.DefaultIfEmpty()
                          Order By c.VIN, If(r Is Nothing, 0, r.Cost)
                          Select New With
                              {
                                  c.VIN,
                                  c.Make,
                                  .Desc = If(r Is Nothing, _
                                             "***No Repairs***", r.Desc),
                                  .Cost = If(r Is Nothing, _
                                             0, r.Cost)
                              }
    For Each item In carsWithRepairs
    txtLog.WriteLine("Car VIN:{0}, Make:{1}, Description:{2} Cost:{3:C}",
           item.VIN, item.Make, item.Desc, item.Cost)
    Next
End Sub
```

Sample of C# Code

```
private void lINQOuterJoinToolStripMenuItem_Click(object sender, EventArgs e)
{
```

```
    var cars = GetCars();
    var repairs = GetRepairs();

    var carsWithRepairs = from c in cars
                          join rep in repairs
                          on c.VIN equals rep.VIN into temp
                             from r in temp.DefaultIfEmpty()
                          orderby c.VIN, r==null?0:r.Cost
                          select new
                          {
                              c.VIN,
                              c.Make,
                              Desc = r==null?"***No Repairs***":r.Desc,
                               Cost = r==null?0:r.Cost
                          };
    foreach (var item in carsWithRepairs)
    {
        txtLog.WriteLine("Car VIN:{0}, Make:{1}, Description:{2} Cost:{3:C}",
          item.VIN, item.Make, item.Desc, item.Cost);
    }
}
```

The result:

```
Car VIN:ABC123, Make:Ford, Description:Fix Flat Tire Cost:$15.00
Car VIN:ABC123, Make:Ford, Description:Replace Wipers Cost:$20.00
Car VIN:ABC123, Make:Ford, Description:Change Oil Cost:$29.99
Car VIN:ABC456, Make:Audi, Description:***No Repairs*** Cost:$0.00
Car VIN:DEF123, Make:BMW, Description:Rotate Tires Cost:$19.99
Car VIN:DEF123, Make:BMW, Description:Fix Windshield Cost:$420.00
Car VIN:DEF456, Make:Ford, Description:Alignment Cost:$30.00
Car VIN:DEF456, Make:Ford, Description:Change Oil Cost:$30.00
Car VIN:HIJ123, Make:VW, Description:Replace Brakes Cost:$200.00
Car VIN:HIJ123, Make:VW, Description:Replace Tires Cost:$1,000.00
```

The car with VIN = ABC456 is included in the result, even though it has no repairs. Another way to perform a left outer join is to use the *GroupJoin* query extension method, discussed earlier in this chapter.

Cross Joins

A cross join is a Cartesian product between two element sources. A Cartesian product will join each record in the outer element source with all elements in the inner source. No join keys are required with this type of join. Cross joins are accomplished by using the *from* clause multiple times without providing any link between element sources. This is often done by mistake.

In the following code sample, there is a *colors* element source and a *cars* element source. The *colors* source represents the available paint colors, and the *cars* source represents the cars that exist. The desired outcome is to combine the colors with the cars to show every combination of car and color available.

Sample of Visual Basic Code

```
Private Sub LINQCrossJoinToolStripMenuItem_Click( _
        ByVal sender As System.Object, _
```

```
        ByVal e As System.EventArgs) _
        Handles LINQCrossJoinToolStripMenuItem.Click
    Dim cars = GetCars()
    Dim colors() = {"Red", "Yellow", "Blue", "Green"}

    Dim carsWithRepairs = From car In cars
                          From color In colors
                          Order By car.VIN, color
                          Select New With
                              {
                                  car.VIN,
                                  car.Make,
                                  car.Model,
                                  .Color = color
                              }
    For Each item In carsWithRepairs
        txtLog.WriteLine("Car VIN:{0}, Make:{1}, Model:{2} Color:{3}",
           item.VIN, item.Make, item.Model, item.Color)
    Next
End Sub
```

Sample of C# Code

```
private void lINQCrossJoinToolStripMenuItem_Click(object sender, EventArgs e)
{
    var cars = GetCars();
    var colors = new string[]{"Red","Yellow","Blue","Green" };

    var carsWithRepairs = from car in cars
                          from color in colors
                          orderby car.VIN, color
                          select new
                          {
                              car.VIN,
                              car.Make,
                              car.Model,
                              Color=color
                          };
    foreach (var item in carsWithRepairs)
    {
        txtLog.WriteLine("Car VIN:{0}, Make:{1}, Model:{2} Color:{3}",
           item.VIN, item.Make, item.Model, item.Color);
    }
}
```

The result:

```
Car VIN:ABC123, Make:Ford, Model:F-250 Color:Blue
Car VIN:ABC123, Make:Ford, Model:F-250 Color:Green
Car VIN:ABC123, Make:Ford, Model:F-250 Color:Red
Car VIN:ABC123, Make:Ford, Model:F-250 Color:Yellow
Car VIN:ABC456, Make:Audi, Model:TT Color:Blue
Car VIN:ABC456, Make:Audi, Model:TT Color:Green
Car VIN:ABC456, Make:Audi, Model:TT Color:Red
Car VIN:ABC456, Make:Audi, Model:TT Color:Yellow
Car VIN:DEF123, Make:BMW, Model:Z-3 Color:Blue
Car VIN:DEF123, Make:BMW, Model:Z-3 Color:Green
```

```
Car VIN:DEF123, Make:BMW, Model:Z-3 Color:Red
Car VIN:DEF123, Make:BMW, Model:Z-3 Color:Yellow
Car VIN:DEF456, Make:Ford, Model:F-150 Color:Blue
Car VIN:DEF456, Make:Ford, Model:F-150 Color:Green
Car VIN:DEF456, Make:Ford, Model:F-150 Color:Red
Car VIN:DEF456, Make:Ford, Model:F-150 Color:Yellow
Car VIN:HIJ123, Make:VW, Model:Bug Color:Blue
Car VIN:HIJ123, Make:VW, Model:Bug Color:Green
Car VIN:HIJ123, Make:VW, Model:Bug Color:Red
Car VIN:HIJ123, Make:VW, Model:Bug Color:Yellow
```

The cross join produces an output for each combination of inputs, which means that the output count is the first input's count one multiplied by the second input's count.

Another way to implement a cross join is to use the *SelectMany* query extension method, covered earlier in this chapter.

Grouping and Aggregation

You will often want to calculate an aggregation such as the total cost of your repairs for each of your cars. LINQ enables you to calculate aggregates for each item by using the *group by* clause. The following code example demonstrates the use of the *group by* clause with the *Sum* aggregate function to output the VIN and the total cost of repairs.

Sample of Visual Basic Code

```
Private Sub LINQGroupByToolStripMenuItem_Click(ByVal sender As System.Object, _
        ByVal e As System.EventArgs) Handles LINQGroupByToolStripMenuItem.Click
    Dim repairs = From r In GetRepairs()
                  Group By VIN = r.VIN
                  Into grouped = Group, TotalCost = Sum(r.Cost)

    For Each item In repairs
        txtLog.WriteLine("Car VIN:{0}, TotalCost:{1:C}",
           item.VIN, item.TotalCost)
    Next
End Sub
```

Sample of C# Code

```
private void lINQGroupByToolStripMenuItem_Click(object sender, EventArgs e)
{
    var repairs = from r in GetRepairs()
                  group r by r.VIN into grouped
                  select new
                  {
                      VIN = grouped.Key,
                      TotalCost = grouped.Sum(c => c.Cost)
                  };
    foreach (var item in repairs)
    {
        txtLog.WriteLine("Car VIN:{0}, Total Cost:{1:C}",
           item.VIN, item.TotalCost);
    }
}
```

The result:

```
Car VIN:ABC123, Total Cost:$64.99
Car VIN:DEF123, Total Cost:$439.99
Car VIN:HIJ123, Total Cost:$1,200.00
Car VIN:DEF456, Total Cost:$60.00
```

This query produced the total cost for the repairs for each car that had repairs, but one car had no repairs, so it's not listed. To list all the cars, you must left join the cars to the repairs and then calculate the sum of the repairs. Also, you might want to add the make of the car to the output and include cars that have no repairs. This requires you to perform a join and group on multiple properties. The following example shows how you can achieve the result.

Sample of Visual Basic Code

```
Private Sub LINQGroupBy2ToolStripMenuItem_Click( _
        ByVal sender As System.Object, _
        ByVal e As System.EventArgs) _
        Handles LINQGroupBy2ToolStripMenuItem.Click
    Dim cars = GetCars()
    Dim repairs = GetRepairs()

    Dim carsWithRepairs = From c In cars
                          Group c By Key = New With {c.VIN, c.Make}
                          Into grouped = Group
                          Group Join r In repairs On Key.VIN Equals r.VIN
                          Into joined = Group
                          Select New With
                                {
                                 .VIN = Key.VIN,
                                 .Make = Key.Make,
                                 .TotalCost = joined.Sum(Function(x) x.Cost)
                                }
    For Each item In carsWithRepairs
        txtLog.WriteLine("Car VIN:{0}, Make:{1}, Total Cost:{2:C}", _
          item.VIN, item.Make, item.TotalCost)
    Next
End Sub
```

Sample of C# Code

```
private void lINQGroupBy2ToolStripMenuItem_Click(object sender, EventArgs e)
{
    var cars = GetCars();
    var repairs = GetRepairs();

    var carsWithRepairs = from c in cars
                          join rep in repairs
                          on c.VIN equals rep.VIN into temp
                          from r in temp.DefaultIfEmpty()
                          group r by new { c.VIN, c.Make } into grouped
                          select new
                          {
                              VIN = grouped.Key.VIN,
                              Make = grouped.Key.Make,
                              TotalCost =
```

```
                                grouped.Sum(c => c == null ? 0 : c.Cost)
                            };
    foreach (var item in carsWithRepairs)
    {
        txtLog.WriteLine("Car VIN:{0}, Make:{1}, Total Cost:{2:C}",
           item.VIN, item.Make, item.TotalCost);
    }
}
```

The result:

```
Car VIN:ABC123, Make:Ford, Total Cost:$64.99
Car VIN:DEF123, Make:BMW, Total Cost:$439.99
Car VIN:ABC456, Make:Audi, Total Cost:$0.00
Car VIN:HIJ123, Make:VW, Total Cost:$1,200.00
Car VIN:DEF456, Make:Ford, Total Cost:$60.00
```

Parallel LINQ (PLINQ)

Parallel LINQ, also known as PLINQ, is a parallel implementation of LINQ to objects. PLINQ implements all the LINQ query extension methods and has additional operators for parallel operations. The degree of concurrency for PLINQ queries is based on the capabilities of the computer running the query.

In many, but not all, scenarios, PLINQ can provide a significant increase in speed by using all available CPUs or CPU cores. A PLINQ query can provide performance gains when you have CPU-intensive operations that can be paralleled, or divided, across each CPU or CPU core. The more computationally expensive the work is, the greater the opportunity for performance gain. For example, if the workload takes 100 milliseconds to execute, a sequential query over 400 elements will take 40 seconds to complete the work, whereas a parallel query on a computer with eight cores might take only 5 seconds. This yields a speedup of 35 seconds.

One problem with Windows applications is that when you try to update a control on your form from a thread other than the thread that created the control, an *InvalidOperationException* is thrown with the message, "Cross-thread operation not valid: Control 'txtLog' accessed from a thread other than the thread it was created on." To work with threading, update in a thread-safe way the following extension method for *TextBox* to the *TextBoxHelper* class.

Sample of Visual Basic Code

```
<Extension()> _
Public Sub WriteLine(ByVal txt As TextBox, _
                     ByVal format As String, _
                     ByVal ParamArray parms As Object())
    Dim line As String = String.Format((format & Environment.NewLine), parms)
    If txt.InvokeRequired Then
        txt.BeginInvoke(New Action(Of String)(AddressOf txt.AppendText), _
                        New Object() {line})
    Else
        txt.AppendText(line)
```

```
    End If
End Sub
```

Sample of C# Code

```
public static void WriteLine(this TextBox txt,
               string format, params object[] parms)
{
    string line = string.Format(format + Environment.NewLine, parms);
    if (txt.InvokeRequired)
    {
        txt.BeginInvoke((Action<string>)txt.AppendText, line);
    }
    else
    {
        txt.AppendText(line);
    }
}
```

You use the *Invoke* or *BeginInvoke* method on the *TextBox* class to marshal the callback to the thread that was used to create the UI control. The *BeginInvoke* method posts an internal dedicated Windows message to the UI thread message queue and returns immediately, which helps avoid thread deadlock situations.

This extension method checks the *TextBox* object to see whether marshaling is required. If marshaling is required (i.e., when the calling thread is not the one used to create the *TextBox* object), the *BeginInvoke* method is executed. If marshaling is not required, the *AppendText* method is called directly on the *TextBox* object. The *BeginInvoke* method takes *Delegate* as a parameter, so *txt.AppendText* is cast to an action of *String*, a general-purpose delegate that exists in the framework, which represents a call to a method that takes a *string* parameter. Now that there is a thread-safe way to display information into the *TextBox* class, the *AsParallel* example can be performed without risking threading-related exceptions.

AsParallel Extension Method

The *AsParallel* extension method divides work onto each processor or processor core. The following code sample starts a stopwatch in the *System.Diagnostics* namespace to show you the elapsed time when completed, and then the *Enumerable* class produces a sequence of integers, from 1 to 10. The *AsParallel* method call is added to the source. This causes the iterations to be spread across the available processor and processor cores. Then a LINQ query retrieves all the even numbers, but in the LINQ query, the *where* clause is calling a *Compute* method, which has a one-second delay using the *Thread* class, which is in the *System.Threading* namespace. Finally, a *foreach* loop displays the results.

Sample of Visual Basic Code

```
Private Sub AsParallelToolStripMenuItem_Click(ByVal sender As System.Object, _
        ByVal e As System.EventArgs) _
        Handles AsParallelToolStripMenuItem.Click
    Dim sw As New Stopwatch
    sw.Start()
```

```
    Dim source = Enumerable.Range(1, 10).AsParallel()
    Dim evenNums = From num In source
                   Where Compute(num) Mod 2 = 0
                   Select num
    For Each ev In evenNums
        txtLog.WriteLine("{0} on Thread {1}", _
            New Object() {ev, Thread.CurrentThread.GetHashCode})
    Next
    sw.Stop()
    txtLog.WriteLine("Done {0}", New Object() {sw.Elapsed})
End Sub

Public Function Compute(ByVal num As Integer) As Integer
    txtLog.WriteLine("Computing {0} on Thread {1}", _
        New Object() {num, Thread.CurrentThread.GetHashCode})
    Thread.Sleep(1000)
    Return num
End Function
```

Sample of C# Code

```
private void asParallelToolStripMenuItem_Click(
    object sender, EventArgs e)
{
    Stopwatch sw = new Stopwatch();
    sw.Start();
    var source = Enumerable.Range(1, 10).AsParallel();
    var evenNums = from num in source
                   where Compute(num) % 2 == 0
                   select num;
    foreach (var ev in evenNums)
    {
        txtLog.WriteLine("{0} on Thread {1}", ev,
            Thread.CurrentThread.GetHashCode());
    }
    sw.Stop();
    txtLog.WriteLine("Done {0}", sw.Elapsed);
}

public int Compute(int num)
{
    txtLog.WriteLine("Computing {0} on Thread {1}", num,
        Thread.CurrentThread.GetHashCode());
    Thread.Sleep(1000);
    return num;
}
```

AsEnumerable **results, showing even numbers, total time, and computing method:**

```
6 on Thread 10
2 on Thread 10
4 on Thread 10
8 on Thread 10
10 on Thread 10
Done 00:00:05.0393262
Computing 1 on Thread 12
```

```
Computing 2 on Thread 11
Computing 3 on Thread 12
Computing 4 on Thread 11
Computing 5 on Thread 11
Computing 6 on Thread 12
Computing 7 on Thread 12
Computing 8 on Thread 11
Computing 9 on Thread 12
Computing 10 on Thread 11
```

The output from the *Compute* calls always shows after the *foreach* (Visual Basic *For Each*) loop output because *BeginInvoke* marshalls calls to the UI thread for execution when the UI thread is available. The *foreach* loop is running on the UI thread, so the thread is busy until the loop completes. The results are not ordered. Your result will vary as well, and, in some cases, the results might be ordered. In the example, you can see that the *foreach* loop displayed the even numbers, using the main thread of the application, which was thread 10 on this computer. The *Compute* method was executed on a different thread, but the thread is either 11 or 12 because this is a two-core processor. Although the *Compute* method has a one-second delay, it took five seconds to execute because only two threads were allocated, one for each core.

In an effort to get a clearer picture of PLINQ, the writing to a *TextBox* has been replaced in the following code. Instead of using *TextBox*, *Debug.WriteLine* is used, which removes the requirement to marshall calls back to the UI thread.

Sample of Visual Basic Code

```
Private Sub AsParallel2ToolStripMenuItem_Click( _
   ByVal sender As System.Object, ByVal e As System.EventArgs) _
   Handles AsParallel2ToolStripMenuItem.Click
   Dim sw As New Stopwatch
   sw.Start()
   Dim source = Enumerable.Range(1, 10).AsParallel()
   Dim evenNums = From num In source
                  Where Compute2(num) Mod 2 = 0
                  Select num
   For Each ev In evenNums
      Debug.WriteLine(String.Format("{0} on Thread {1}", _
          New Object() {ev, Thread.CurrentThread.GetHashCode}))
   Next
   sw.Stop()
   Debug.WriteLine(String.Format("Done {0}", New Object() {sw.Elapsed}))
End Sub
```

Sample of C# Code

```
private void asParallel2ToolStripMenuItem_Click(
   object sender, EventArgs e)
{
   Stopwatch sw = new Stopwatch();
   sw.Start();
   var source = Enumerable.Range(1, 10).AsParallel();
   var evenNums = from num in source
```

```
                where Compute2(num) % 2 == 0
                select num;
   foreach (var ev in evenNums)
   {
      Debug.WriteLine(string.Format("{0} on Thread {1}", ev,
         Thread.CurrentThread.GetHashCode()));
   }
   sw.Stop();
   Debug.WriteLine(string.Format("Done {0}", sw.Elapsed));
}

public int Compute2(int num)
{
   Debug.WriteLine(string.Format("Computing {0} on Thread {1}", num,
      Thread.CurrentThread.GetHashCode()));
   Thread.Sleep(1000);
   return num;
}
```

The result:

```
Computing 2 on Thread 10
Computing 1 on Thread 6
Computing 3 on Thread 10
Computing 4 on Thread 6
Computing 5 on Thread 10
Computing 6 on Thread 6
Computing 7 on Thread 10
Computing 8 on Thread 6
Computing 9 on Thread 10
Computing 10 on Thread 6
2 on Thread 9
4 on Thread 9
6 on Thread 9
8 on Thread 9
10 on Thread 9
Done 00:00:05.0632071
```

The result, which is in the Visual Studio .NET Output window, shows that there is no waiting for the UI thread. Once again, your result will vary based on your hardware configuration.

ForAll Extension Method

When the query is iterated by using a *foreach* (Visual Basic *For Each*) loop, each iteration is synchronized in the same thread, to be treated one after the other in the order of the sequence. If you just want to perform each iteration in parallel, without any specific order, use the *ForAll* method. It has the same effect as performing each iteration in a different thread. Analyze this technique to verify that you get the performance gain you expect. The following example shows the use of the *ForAll* method instead of the *For Each* (C# *foreach*) loop.

Sample of Visual Basic Code

```
Private Sub ForAllToolStripMenuItem_Click(ByVal sender As System.Object, _
        ByVal e As System.EventArgs) _
```

```
        Handles ForAllToolStripMenuItem.Click
    Dim sw As New Stopwatch
    sw.Start()
    Dim source = Enumerable.Range(1, 10).AsParallel()
    Dim evenNums = From num In source
                   Where Compute2(num) Mod 2 = 0
                   Select num
    evenNums.ForAll(Sub(ev) Debug.WriteLine(string.Format(
                            "{0} on Thread {1}", ev, _
                           Thread.CurrentThread.GetHashCode())))
    sw.Stop()
    Debug.WriteLine((string.Format("Done {0}", New Object() {sw.Elapsed}))
End Sub
```

Sample of C# Code

```
private void forAllToolStripMenuItem_Click(object sender, EventArgs e)
{
    Stopwatch sw = new Stopwatch();
    sw.Start();
    var source = Enumerable.Range(1, 10).AsParallel();
    var evenNums = from num in source
                   where Compute(num) % 2 == 0
                   select num;
    evenNums.ForAll(ev => Debug.WriteLine(string.Format(
                      "{0} on Thread {1}", ev,
                      Thread.CurrentThread.GetHashCode())));
    sw.Stop();
    Debug.WriteLine(string.Format("Done {0}", sw.Elapsed));
}
```

***ForAll* result, showing even numbers, total time, and computing method:**

```
Computing 1 on Thread 9
Computing 2 on Thread 10
Computing 3 on Thread 9
2 on Thread 10
Computing 4 on Thread 10
Computing 5 on Thread 9
4 on Thread 10
Computing 6 on Thread 10
Computing 7 on Thread 9
6 on Thread 10
Computing 8 on Thread 10
Computing 9 on Thread 9
8 on Thread 10
Computing 10 on Thread 10
10 on Thread 10
Done 00:00:05.0556551
```

Like the previous example, the results are not guaranteed to be ordered, and there is no attempt to put the results in a particular order. This technique can give you better performance as long as this behavior is acceptable.

AsOrdered Extension Method

Sometimes, you must maintain the order in your query, but you still want parallel execution. Although this will come at a cost, it's doable by using the *AsOrdered* extension method. The following example shows how you can add this method call right after the *AsParallel* method to maintain order.

Sample of Visual Basic Code

```
Private Sub AsOrderedToolStripMenuItem_Click(ByVal sender As System.Object, _
        ByVal e As System.EventArgs) _
        Handles AsOrderedToolStripMenuItem.Click
    Dim sw As New Stopwatch
    sw.Start()
    Dim source = Enumerable.Range(1, 10).AsParallel().AsOrdered()
    Dim evenNums = From num In source
                   Where Compute2(num) Mod 2 = 0
                   Select num
    evenNums.ForAll(Sub(ev) Debug.WriteLine(string.Format(
                          "{0} on Thread {1}", ev, _
                          Thread.CurrentThread.GetHashCode())))
    sw.Stop()
    Debug.WriteLine(string.Format("Done {0}", New Object() {sw.Elapsed}))
End Sub
```

Sample of C# Code

```
private void asOrderedToolStripMenuItem_Click(object sender, EventArgs e)
{
    Stopwatch sw = new Stopwatch();
    sw.Start();
    var source = Enumerable.Range(1, 10).AsParallel().AsOrdered();
    var evenNums = from num in source
                   where Compute2(num) % 2 == 0
                   select num;

    evenNums.ForAll(ev => Debug.WriteLine(string.Format(
                      "{0} on Thread {1}", ev,
                      Thread.CurrentThread.GetHashCode())));
    sw.Stop();
    Debug.WriteLine(string.Format("Done {0}", sw.Elapsed));
}
```

***AsOrdered* result, showing even numbers, total time, and computing method:**

```
Computing 2 on Thread 11
Computing 1 on Thread 10
2 on Thread 11
Computing 4 on Thread 11
Computing 3 on Thread 10
4 on Thread 11
Computing 6 on Thread 11
Computing 5 on Thread 10
6 on Thread 11
Computing 8 on Thread 11
Computing 7 on Thread 10
8 on Thread 11
```

```
Computing 9 on Thread 11
Computing 10 on Thread 10
10 on Thread 10
Done 00:00:05.2374586
```

The results are ordered, at least for the even numbers, which is what the *AsOrdered* extension method is guaranteeing.

PRACTICE Working with Disconnected Data Classes

In this practice, you convert the Web application from Lesson 1 to use LINQ queries instead of query extension methods. The result of this practice functions the same way, but you will see how using LINQ queries can improve readability.

If you encounter a problem completing an exercise, the completed projects can be installed from the Code folder on the companion CD.

EXERCISE 1 Converting from Query Extension Methods to LINQ Queries

In this exercise, you modify the Web application you created in Lesson 1 to use LINQ queries.

1. In Visual Studio .NET 2010, choose File | Open | Project. Open the project from Lesson 1 or locate and open the solution in the Begin folder for this lesson.
2. In Solution Explorer, right-click the Default.aspx file and select View Code to open the code-behind file containing the code from Lesson 1.
3. In the *Page_Load* method, locate the statement that contains all the *Where* method calls as follows:

Sample of Visual Basic Code

```
Dim result = Vehicles _
        .Where(Function(v) v.VIN.StartsWith(txtVin.Text)) _
        .Where(Function(v) v.Make.StartsWith(txtMake.Text)) _
        .Where(Function(v) v.Model.StartsWith(txtModel.Text)) _
        .Where(Function(v) v.Cost > Decimal.Parse(ddlCost.SelectedValue)) _
        .Where(Function(v) v.Year > Integer.Parse(ddlYear.SelectedValue))
```

Sample of C# Code

```
var result = vehicles
    .Where(v => v.VIN.StartsWith(txtVin.Text))
    .Where(v => v.Make.StartsWith(txtMake.Text))
    .Where(v => v.Model.StartsWith(txtModel.Text))
    .Where(v => v.Cost > Decimal.Parse(ddlCost.SelectedValue))
    .Where(v => v.Year > int.Parse(ddlYear.SelectedValue));
```

4. Convert the previous code to use a LINQ query. Your code should look like the following:

Sample of Visual Basic Code

```
Dim result = From v In Vehicles
             Where v.VIN.StartsWith(txtVin.Text) _
             And v.Make.StartsWith(txtMake.Text) _
```

```
            And v.Model.StartsWith(txtModel.Text) _
            And v.Cost > Decimal.Parse(ddlCost.SelectedValue) _
            And v.Year > Integer.Parse(ddlYear.SelectedValue) _
            Select v
```

Sample of C# Code

```
var result = from v in vehicles
             where v.VIN.StartsWith(txtVin.Text)
             &&    v.Make.StartsWith(txtMake.Text)
             &&    v.Model.StartsWith(txtModel.Text)
             &&    v.Cost > Decimal.Parse(ddlCost.SelectedValue)
             &&    v.Year > int.Parse(ddlYear.SelectedValue)
             select v;
```

Behind the scenes, these queries do the same thing as the previous code, which implemented many *Where* calls by using method chaining.

5. Locate the *SetOrder* method. Replace the code in this method to use LINQ expressions. Your code should look like the following:

Sample of Visual Basic Code

```
Private Function SetOrder(ByVal order As String, _
        ByVal query As IEnumerable(Of Vehicle)) As IEnumerable(Of Vehicle)
    Select Case order
        Case "VIN"
            Return From v In query Order By v.VIN Select v
        Case "Make"
            Return From v In query Order By v.Make Select v
        Case "Model"
            Return From v In query Order By v.Model Select v
        Case "Year"
            Return From v In query Order By v.Year Select v
        Case "Cost"
            Return From v In query Order By v.Cost Select v
        Case Else
            Return query
    End Select

End Function
```

Sample of C# Code

```
private IEnumerable<Vehicle> SetOrder(string order,
        IEnumerable<Vehicle> query)
{
    switch (order)
    {
        case "VIN":
            return from v in query orderby v.VIN select v;
        case "Make":
            return from v in query orderby v.Make select v;
        case "Model":
            return from v in query orderby v.Model select v;
        case "Year":
            return from v in query orderby v.Year select v;
        case "Cost":
```

```
            return from v in query orderby v.Cost select v;
        default:
            return query;
    }
}
```

6. Locate the data-binding code. This code uses the *Select* query extension method to instantiate an anonymous type, which is then bound to the grid as follows:

   Sample of Visual Basic Code

   ```
   gvVehicles.DataSource = result.Select(Function(v, i) New With
                   {.Index = i, v.VIN, v.Make, v.Model, v.Year, v.Cost})
   gvVehicles.DataBind()
   ```

   Sample of C# Code

   ```
   gvVehicles.DataSource = result.Select((v, i)=> new
       {Index = i, v.VIN, v.Make, v.Model, v.Year, v.Cost});
   gvVehicles.DataBind();
   ```

   Can you convert the previous code to a LINQ query? The LINQ *select* keyword doesn't support the index parameter value this code uses. You could spend time trying to find a way to convert this code, but it's better to leave this code as is.

7. Choose Build | Build Solution to build the application. If you have errors, you can double-click the error to go to the error line and correct.

8. Choose Debug | Start Debugging to run the application.

   When the application starts, you should see a Web page with your GUI controls that enables you to specify filter and sort criteria. If you type the letter **F** into the Make text box and click Execute, the grid will be populated only with items that begin with F. If you set the sort order and click the Execute button again, you will see the sorted results.

Lesson Summary

This lesson provided an introductions to LINQ.

- You can use LINQ queries to provide a typed method of querying any generic *IEnumerable* object.
- LINQ queries can be more readable than using query extension methods.
- Not all query extension methods map to LINQ keywords, so you might still be required to use query extension methods with your LINQ queries.
- Although the *Select* query extension method maps to the LINQ select keyword, the LINQ select keyword doesn't support the index parameter the *Select* query extension method has.
- LINQ queries enable you to filter, project, sort, join, group, and aggregate.
- PLINQ provides a parallel implementation of LINQ that can increase the performance of LINQ queries.

Lesson Review

You can use the following questions to test your knowledge of the information in Lesson 2, "Using LINQ Queries." The questions are also available on the companion CD if you prefer to review them in electronic form.

> ***NOTE*** **ANSWERS**
>
> **Answers to these questions and explanations of why each answer choice is correct or incorrect are located in the "Answers" section at the end of the book.**

1. Given the following LINQ query:

   ```
   from c in cars join r in repairs on c.VIN equals r.VIN …
   ```

   what kind of join does this perform?

   A. Cross join

   B. Left outer join

   C. Right outer join

   D. Inner join

2. In a LINQ query that starts with:

   ```
   from o in orderItems
   ```

   The *orderItems* collection is a collection of *OrderItem* with properties called *UnitPrice*, *Discount*, and *Quantity*. You want the query to filter out *OrderItem* objects whose *totalPrice* (UnitPrice * Quantity * Discount) result is less than 100. You want to sort by *totalPrice*, and you want to include the total price in your *select* clause. Which keyword can you use to create a *totalPrice* result within the LINQ query so you don't have to repeat the formula three times?

   A. *let*

   B. *on*

   C. *into*

   D. *by*

Case Scenarios

In the following case scenarios, you will apply what you've learned about LINQ as discussed in this chapter. You can find answers to these questions in the "Answers" section at the end of this book.

Case Scenario 1: Fibonacci Sequence

You were recently challenged to create an expression to produce the Fibonacci sequence for a predetermined quantity of iterations. An example of the Fibonacci sequence is:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55

The sequence starts with 0 and 1, known as the seed values. The next number is always the sum of the previous two numbers, so 0 + 1 = 1 to get the third element, 1 + 1 = 2 to get the fourth element, 2 + 1 = 3 for the fifth element, 3 + 2 = 5 for the sixth element, and so on.

Answer the following questions regarding the implementation of the Fibonacci sequence.

1. Can you write an expression using a LINQ query or query extension methods that will produce Fibonacci numbers for a predetermined quantity of iterations?
2. Instead of producing Fibonacci numbers for a predetermined quantity of iterations, how about producing Fibonacci numbers until you reach a desired maximum value?

Case Scenario 2: Sorting and Filtering Data

In your application, you are using a collection of *Customer*, a collection of *Order*, and a collection of *OrderItem*. Table 3-3 shows the properties of each of the classes. The total price for *OrderItem* is Quantity * Price * Discount. The *Order* amount is the sum of the total price of the order items. The *max Quantity* value is the maximum quantity of products purchased for a customer.

You must write a LINQ query that produces a generic *IEnumerable* result that contains *CustomerID*, *Name*, *OrderAmount*, and *MaxQuantity*. You produce this data only for orders whose amount is greater than $1,000. You want to sort by *OrderAmount* descending.

TABLE 3-3 Classes with Corresponding Properties

CUSTOMER	*ORDER*	*ORDERITEM*
CustomerID	*OrderID*	*OrderItemID*
Name	*CustomerID*	*OrderID*
Address	*OrderDate*	*ProductID*
City	*RequiredDate*	*Quantity*
State	*ShippedDate*	*Price*
		Discount

1. Can you produce a LINQ query that solves this problem?
2. Can you produce a solution to this problem by using query extension methods?

Suggested Practices

To help you successfully master the exam objectives presented in this chapter, complete the following tasks.

Create Query with Extension Methods

You should create at least one application that uses the LINQ and query extension methods. This can be accomplished by completing the practices at the end of Lesson 1 and Lesson 2 or by completing the following Practice 1.

- **Practice 1** Create an application that requires you to collect data into at least two generic collections in which the objects in these collections are related. This could be movies that have actors, artists who record music, or people who have vehicles. Add query extension methods to perform inner joins of these collections and retrieve results.
- **Practice 2** Complete Practice 1 and then add query extension methods to perform outer joins and *group by* with aggregations.

Create LINQ Queries

You should create at least one application that uses the LINQ and query extension methods. This can be accomplished by completing the practices at the end of Lesson 1 and Lesson 2 or by completing the following Practice 1.

- **Practice 1** Create an application that requires you to collect data into at least two generic collections in which the objects in these collections are related. This could be movies that have actors, artists who record music, or people who have vehicles. Add LINQ queries to perform inner joins of these collections and retrieve results.
- **Practice 2** Complete Practice 1 and then add query LINQ queries to perform outer joins and *group by* with aggregations.

Take a Practice Test

The practice tests on this book's companion CD offer many options. For example, you can test yourself on just the lesson review content, or you can test yourself on all the 70-516 certification exam content. You can set up the test so that it closely simulates the experience of taking

a certification exam, or you can set it up in study mode so that you can look at the correct answers and explanations after you answer each question.

MORE INFO **PRACTICE TESTS**

For details about all the practice test options available, see the "How to Use the Practice Tests" section in this book's introduction.

Index

Symbols and Numbers

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z

About the Author

GLENN JOHNSON is a professional trainer, consultant, and developer whose experience spans over 20 years. As a consultant and developer, he has worked on many large projects, most of them in the insurance industry. Glenn's strengths are with Microsoft products, such as ASP.NET, MVC, Silverlight, WPF, WCF, and Microsoft SQL Server, using C#, Visual Basic, and T-SQL. This is yet one more of many .NET books that Glenn has authored; he also develops courseware and teaches classes in many countries on Microsoft ASP.NET, Visual Basic 2010, C#, and the .NET Framework.

Glenn holds the following Microsoft Certifications: MCT, MCPD, MCTS, MCAD, MCSD, MCDBA, MCP + Site Building, MCSE + Internet, MCP + Internet, and MCSE. You can find Glenn's website at http://GJTT.com.