

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:

- Lesson 1: Understanding LINQ **145**
- Lesson 2: Using LINQ Queries **205**

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 *IEnumerator* 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.

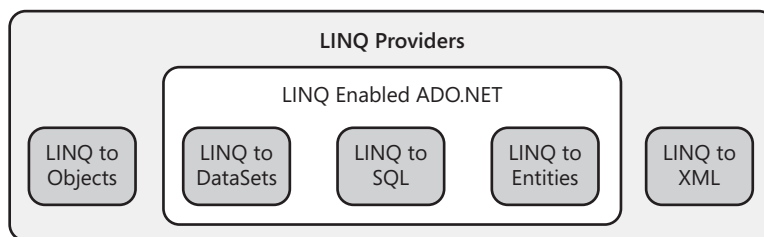


FIGURE 3-1 This figure shows the LINQ providers built into the .NET Framework.

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.

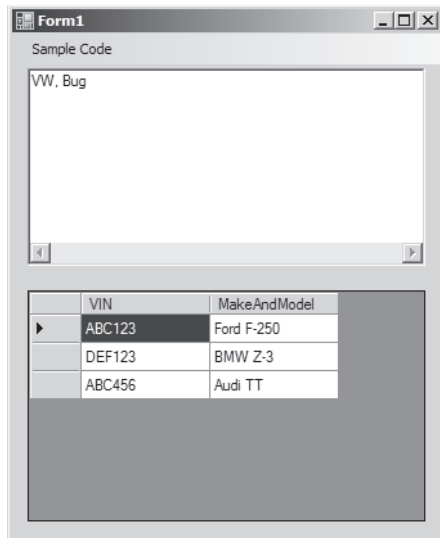


FIGURE 3-2 Anonymous types are displayed in the grid.

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 through 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 PredicateDelegateToolStripMenuItem_Click( _
    ByVal sender As System.Object, _
    ByVal e As System.EventArgs) _
    Handles PredicateDelegateToolStripMenuItem.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.

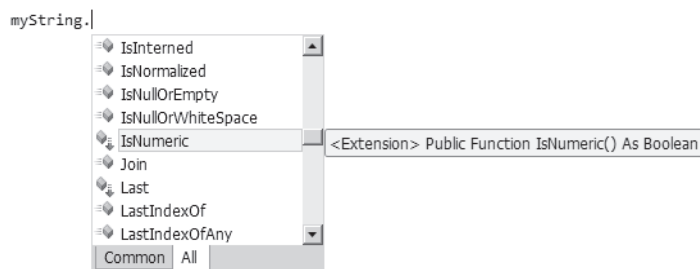


FIGURE 3-3 Extension methods for the *string* class are shown in the IntelliSense window.

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.

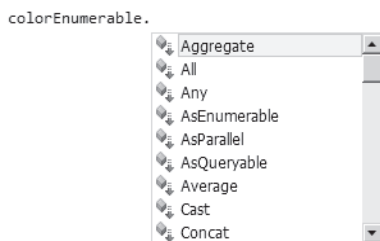


FIGURE 3-4 This figure shows *Extension* methods on the *IEnumerable* interface.

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 elementToolStripMenuItem_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) _
    Handles elementToolStripMenuItem.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 elementToolStripMenuItem_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

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
```

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
```

```
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, 23, 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, 23, 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.

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.

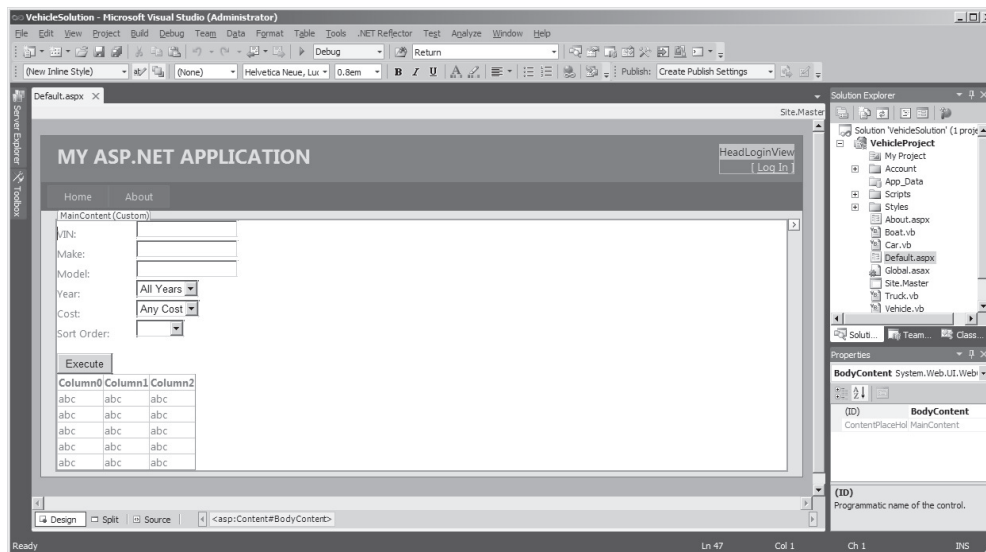


FIGURE 3-5 This is the rendered screen showing filter and sort settings.

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. Your 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)
    End Select
End Function

```

```

        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*. Your 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
<i>from</i>	Specifies a data source and a range variable
<i>where</i>	Filters source elements based on one or more Boolean expressions
<i>select</i>	Specifies the type and shape the elements in the returned sequence have when the query is executed

<i>group</i>	Groups query results according to a specified key value
<i>into</i>	Provides an identifier that can serve as a reference to the results of a <i>join</i> , <i>group</i> , or <i>select</i> clause
<i>orderby</i> (Visual Basic: <i>Order By</i>)	Sorts query results in ascending or descending order
<i>join</i>	Joins two data sources based on an equality comparison between two specified matching criteria
<i>let</i>	Introduces a range variable to store subexpression results in a query expression
<i>in</i>	Contextual keyword in a <i>from</i> or <i>join</i> clause to specify the data source
<i>on</i>	Contextual keyword in a <i>join</i> clause to specify the join criteria
<i>equals</i>	Contextual keyword in a <i>join</i> clause to join two sources
<i>by</i>	Contextual keyword in a <i>group</i> clause to specify the grouping criteria
<i>ascending</i>	Contextual keyword in an <i>orderby</i> clause
<i>descending</i>	Contextual keyword in an <i>orderby</i> 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
<i>Distinct</i>	Filters duplicate elements
<i>Skip/Skip While</i>	Jumps over elements before returning results
<i>Take/Take While</i>	Provides a means to limit how many elements will be retrieved
<i>Aggregate</i>	Includes aggregate functions in your queries
<i>Into</i>	Contextual keyword in the <i>Aggregate</i> clause that specifies what to do with the result of the aggregate
<i>All</i>	Contextual keyword in the <i>Aggregate</i> clause that determines whether all elements meet the specified criterion
<i>Any</i>	Contextual keyword in the <i>Aggregate</i> clause that determines whether any of the elements meet the specified criterion
<i>Average</i>	Contextual keyword in the <i>Aggregate</i> clause that calculates the average value

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

```
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
    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 or the `&&` C# operator. 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 r in repairs
                        on c.VIN equals r.VIN into g
                        from r in g.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

```

```
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 marshal 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.

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 a detailed overview of the ADO.NET disconnected classes.

- 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* ($\text{UnitPrice} * \text{Quantity} * \text{Discount}$) 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 $\text{Quantity} * \text{Price} * \text{Discount}$. 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
<i>CustomerID</i>	<i>OrderID</i>	<i>OrderItemID</i>
<i>Name</i>	<i>OrderDate</i>	<i>ProductID</i>
<i>Address</i>	<i>RequiredDate</i>	<i>Quantity</i>
<i>City</i>	<i>ShippedDate</i>	<i>Price</i>
<i>State</i>		<i>Discount</i>

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.

CHAPTER 4

LINQ to SQL

In the past, one of the biggest problems developers have had with ADO.NET is that it forced everyone to create data-centric applications. This meant it was difficult to write an object-centric application that was focused on business objects because you had to think about the ADO.NET data-centric objects, such as *DataSet* and *DataTable*, and how you would use these objects to get proper persistence. These objects also caused problems when working with null values.

LINQ to SQL was released with Visual Studio 2008 as the first solution by Microsoft to the impedance mismatch between applications and data. LINQ to SQL enables you to access SQL Server by LINQ queries. In this chapter, you see how LINQ to SQL can put the fun back into programming data access.

Exam objectives in this chapter:

- Map entities and relationships by using LINQ to SQL.
- Create disconnected objects.
- Manage the *DataContext* and *ObjectContext*.
- Cache data.
- Create, update, or delete data by using *DataContext*.
- Create a LINQ query.

Lessons in this chapter:

- Lesson 1: What Is LINQ to SQL? **239**
- Lesson 2: Executing Queries Using LINQ to SQL **260**
- Lesson 3: Submitting Changes to the Database **277**

Before You Begin

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



REAL WORLD

Glenn Johnson

Working with the classic ADO.NET classes such as *DataSet* and *DataTable* can be somewhat painful, especially when you have to deal with null values from the database. The first time I used LINQ to SQL was on a small project in which I needed to access a database that had several tables, and I needed to decide which technology to use for the data access. I decided to try LINQ to SQL. I was pleasantly surprised at how easy it was to set up and use.

Lesson 1: What Is LINQ to SQL?

LINQ to SQL provides a framework for managing relational data as objects, but you can still query the data. LINQ to SQL is an object-relational mapping (ORM) tool that enables you not only to query the data but also to insert, update, or delete the data. You can use an object-centric approach to manipulate the objects in your application while LINQ to SQL is in the background, tracking your changes.

In this lesson, you learn about modeling data.

After this lesson, you will be able to:

- Generate a LINQ to SQL model from an existing database.
- Use the LINQ to SQL model to map stored procedures to methods.
- Use a *DataContext* object to manage your database connection and context.
- Understand how LINQ to SQL connects to your database.
- Store information about objects and their state.
- Understand object lifetime and how objects are cached.
- Understand eager loading versus lazy loading.

Estimated lesson time: 45 minutes

Modeling Your Data

Probably the best way to help you gain an understanding of LINQ to SQL is to start with some data modeling to help you see the big picture of the LINQ to SQL capabilities. This also helps by providing a visual model of your classes and how they relate to each other.

Generating a LINQ to SQL Model from an Existing Database

The easiest way to get started with LINQ to SQL is to generate a model from an existing database. This can be accomplished by right-clicking your project node in Solution Explorer, and choosing Add | New Item | LINQ to SQL Classes. Name the file **Northwind.dbml**, as shown in Figure 4-1.

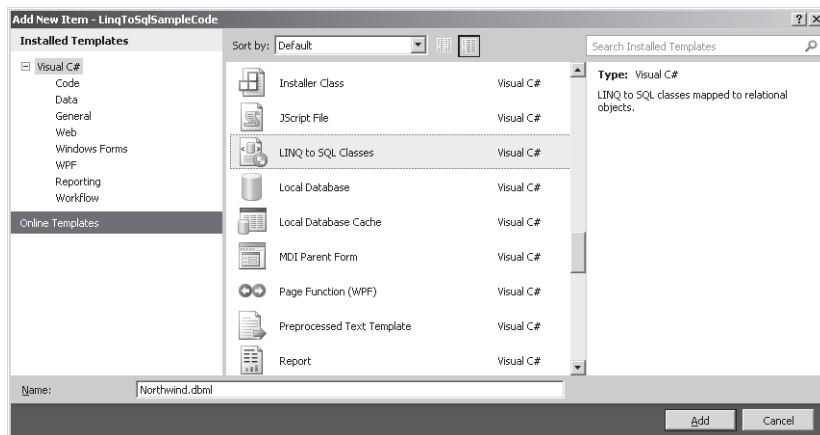


FIGURE 4-1 Select LINQ to SQL classes and name the file Northwind.dbml.

The file extension is .dbml (database markup language), which is an XML file that contains the model settings. After naming the file, click Add. The file will be rendered to your screen as a two-paneled window in which the left side displays table entities and the right side displays stored procedures.

NOTE BE CAREFUL WHEN NAMING THE .DBML FILE

The name you assign to the file will also be used to create a *DataContext* object called *nameDataContext*. To ensure Pascal casing on your data context object, be sure to use Pascal casing on this file name. For example, if you name this file nOrThWiNd.dbml, the data context class that is created will be called nOrThWiNdDataContext. You can go to the *DataContext* properties to change the name if you make a mistake when naming the file, but being careful when naming the file will save you time.

From Server Explorer, you can drag tables to the left pane and drop them. This requires you to have a configured connection to Microsoft SQL Server. If you don't have a connection to the Northwind database, you can right-click the *Data Connections* node, click Add Connection, select Microsoft SQL Server, and click OK. In the Add Connection window, type your server name (for example, \SQLExpress for your local SQL Server Express instance) and, in the Select Or Enter A Database Name drop-down list, select the Northwind database and click OK.

You can also drag stored procedures to the right pane and drop them. Figure 4-2 shows the model diagram after dragging and dropping the Customers, Orders, Order Details, and Employees tables and the CustOrderHist and CustOrdersDetail stored procedures.

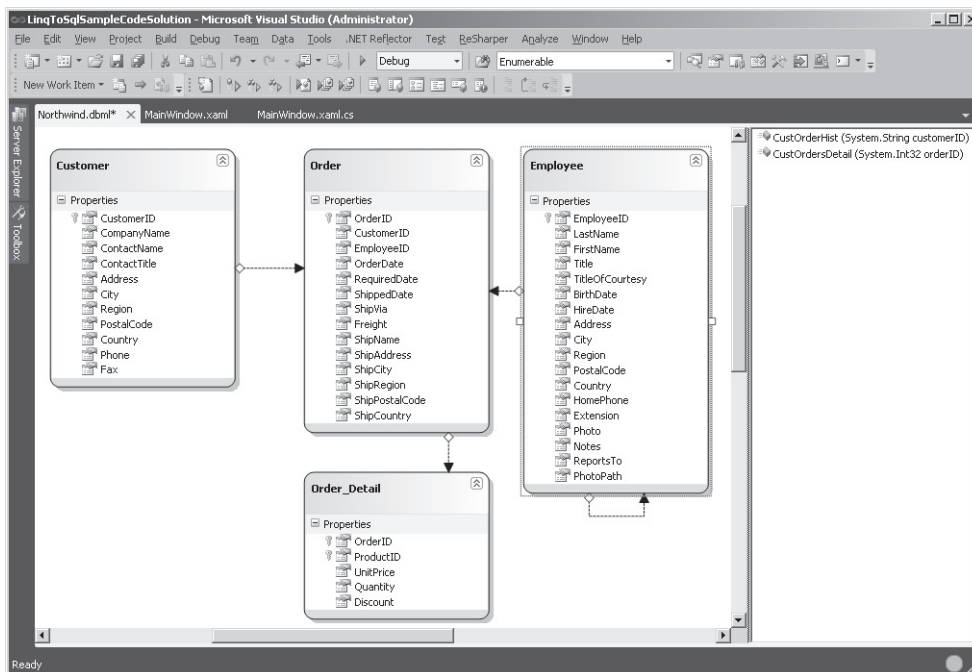


FIGURE 4-2 The model diagram shows tables as classes and stored procedures as methods.

Examining the Model

In Figure 4-2, the Customers table was added, but a class, *Customer* (singular), is shown. An instance of the *Customer* class represents a row in the Customers table. The LINQ to SQL designer automatically attempts to singularize plural table names. Most of the time, this works as expected, but it's not that smart. For example, a movies table will produce a *Movy* class instead of a *Movie* class. You can override the proposed name of any class by clicking the class in the design window and opening the Properties window to change the *Name* property to any valid class name.

In the Properties window, other properties can be configured. The *Insert*, *Update*, and *Delete* properties are defaulted to use the run time to generate the appropriate logic, but this can be changed to execute a stored procedure instead.

The primary key is also highlighted in the diagram by displaying a key beside all properties that make up the primary key. For example, notice that the *Order_Detail* class has two primary key properties to indicate that these properties are combined to produce a unique key.

The LINQ to SQL designer also imported the relationships into your diagram. For example, customers place orders, so you can see that an association line is drawn between the *Customer* class and the *Order* class. The association line shows a one-to-many relationship between the *Customer* class and the *Order* class. This also can be stated as "a customer has orders." You can use the Properties window to change the configuration of the associations.

Mapping Stored Procedures

With LINQ to SQL, you can easily access stored procedures as regular methods in your code, as shown in Figure 4-2, in which two stored procedures were added to the model by dragging and dropping them to the designer surface. The icon displayed in the designer is the standard method icon. If you click the *CustOrderHist* stored procedure, you'll see its properties, as shown in Figure 4-3.

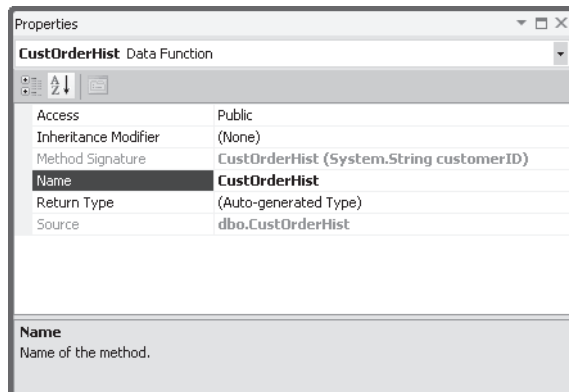


FIGURE 4-3 This figure displays the properties of the *CustOrderHist* stored procedure.

In Figure 4-3, the method signature is defined as *CustOrderHist* (*System.String* *customerID*), which means that a method called *CustOrderHist* will be created that accepts a string argument representing the customer ID.

NOTE WHAT DOES THE STORED PROCEDURE METHOD RETURN?

The designer will make an attempt to auto-define a new type that represents the output, but this works in simple scenarios only. If you have a stored procedure with conditional code that will return different result types based on a condition, the designer won't be smart enough to return the correct type. The design simply tries to execute the stored procedure with the SET FMTONLY ON option set, and it passes default values into the parameters to see what is returned. In the Properties window, you can specify the return type, but you can see that in some scenarios this will not be useful. Your solution will be either to rewrite the stored procedure or revert to traditional ADO.NET to get the returned result into a data table.

If you have a stored procedure that returns an entity type, for example, a stored procedure that returns a filtered list of customers, you can drag the stored procedure from Server Explorer and drop it on to the *Customer* entity. This will tell the designer that you want to return a list of *Customer* objects. If you've already added the stored procedure to the designer, you can set the Return Type in the Properties window, which informs the designer that you are returning an *IEnumerable* of the type you select.

Another example of using stored procedures with LINQ to SQL is when you want the *insert*, *update*, and *delete* statements to be executed as stored procedures instead of as dynamic SQL statements. This can be configured by clicking the appropriate entity class, for example, the *Customer* class, and setting *Insert*, *Update*, and *Delete* properties to the appropriate stored procedure, as shown in Figure 4-4.

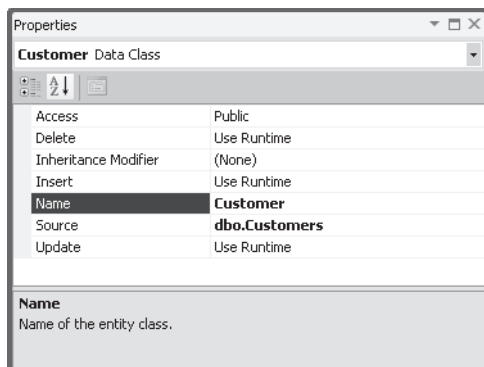


FIGURE 4-4 Each entity class has properties for *Insert*, *Update*, and *Delete*.

Figure 4-4 shows the default settings for *Insert*, *Update*, and *Delete*, but you can assign a stored procedure to these commands. In Figure 4-4, there is no property for *Select*. If you want to use a stored procedure for selecting customers, you can create the stored procedure and drag it to the design surface to create a method for selecting and then change the return type of the stored procedure to the *Customer* type.

Examining the Designer Output

When you save and close the LINQ to SQL designer, it creates types in your application that can access the entities and stored procedures in your model diagram. These types can be viewed by clicking the plus sign beside the Northwind.dbml file. If you don't have a plus sign beside the Northwind.dbml file, click the Show All Files button at the top of the Solution Explorer window. Under the Northwind.dbml file, you will see a Northwind.dbml.layout file, which is an XML file that contains layout information describing where the elements are on the design surface. The Northwind.dbml.vb (or Northwind.dbml.cs) file also contains the generated types. Open this file to see its contents.

The following classes are defined in this file: *Customer*, *CustOrderHistResult*, *CustOrderDetailsResult*, *Employee*, *NorthwindDataContext*, *Order*, and *Order_Detail*. The classes that have the "Result" suffix are auto-created to represent the results from the stored procedures.

Examining an *Entity Class*

The section focuses on one of the entity classes, the *Customer* class. All the other entity classes are implemented in a similar fashion. If you understand the *Customer* class, you should be able to understand the other entity classes.

When you locate the *Customer* class, you'll notice that this class is adorned with attributes, as shown in the following code sample:

Sample of Visual Basic Code

```
<Global.System.Data.Linq.Mapping.TableAttribute(Name="dbo.Customers"), _
Global.System.Runtime.Serialization.DataContractAttribute(> _
Partial Public Class Customer
    Implements System.ComponentModel.INotifyPropertyChanging,
                System.ComponentModel.INotifyPropertyChanged
    ' more code here
End Class
```

Sample of C# Code

```
[global::System.Data.Linq.Mapping.TableAttribute(Name = "dbo.Customers")]
[global::System.Runtime.Serialization.DataContractAttribute()]
public partial class Customer : INotifyPropertyChanging, INotifyPropertyChanged
{
    //mode code here
}
```

The first attribute is *TableAttribute*, which LINQ to SQL uses to identify the table in SQL Server that this class represents. This means that *TableAttribute* links the *Customer* class to the *dbo.Customers* table in the database because the *Name* property specifies the exact name of the database table. If no *Name* property is supplied, LINQ to SQL assumes the database table has the same name as the class. Only instances of classes declared as tables are stored in the database. Instances of these types of classes are known as *entities*. The classes themselves are known as *entity classes*.

The second attribute is *DataContractAttribute*, which enables serialization of the *Customer* class when used with Windows Communication Foundation (WCF) services. This attribute exists because the *Serialization* property on *NorthwindDataContext* was set to *Unidirectional*. If you didn't set the *Serialization Mode* property, you won't see this attribute. (Read more about this in the "Examining the DataContext Class" section of this chapter).

The *Customer* class implements the *INotifyPropertyChanging* interface, which defines a *PropertyChanging* event. The *Customer* entity uses this interface to tell the LINQ to SQL change tracker when it has changed. If you don't implement *INotifyPropertyChanging*, the LINQ to SQL change tracker assumes that all objects queried will change, and it automatically keeps a copy of all queried objects.

The *Customer* class also implements the *INotifyPropertyChanged* interface, which has a *PropertyChanged* event. This interface is implemented for use with data binding. If your object will not be data-bound, it will not need this interface implementation.

Next, the *Customer* class has private fields and public properties for each column in the database table. The following code sample shows the *CustomerID*.

Sample of Visual Basic Code

```
Private _CustomerID As String

<Global.System.Data.Linq.Mapping.ColumnAttribute(Storage:="_CustomerID", _
    DbType:"NChar(5) NOT NULL", CanBeNull:=False, IsPrimaryKey:=True), _
    Global.System.Runtime.Serialization.DataMemberAttribute(Order:=1)> _
Public Property CustomerID() As String
    Get
        Return Me._CustomerID
    End Get
    Set(ByVal value As String)
        If (String.Equals(Me._CustomerID, value) = False) Then
            Me.OnCustomerIDChanging(value)
            Me.SendPropertyChanging()
            Me._CustomerID = value
            Me.SendPropertyChanged("CustomerID")
            Me.OnCustomerIDChanged()
        End If
    End Set
End Property
```

Sample of C# Code

```
private string _CustomerID;

[global::System.Data.Linq.Mapping.ColumnAttribute(Storage="_CustomerID",
    DbType="NChar(5) NOT NULL", CanBeNull=false, IsPrimaryKey=true)]
[global::System.Runtime.Serialization.DataMemberAttribute(Order=1)]
public string CustomerID
{
    get
    {
        return this._CustomerID;
    }
    set
    {
        if ((this._CustomerID != value))
        {
            this.OnCustomerIDChanging(value);
            this.SendPropertyChanging();
            this._CustomerID = value;
            this.SendPropertyChanged("CustomerID");
            this.OnCustomerIDChanged();
        }
    }
}
```

In the code example, the *CustomerID* public property is adorned with *ColumnAttribute*. This attribute identifies each persistable property. Without this attribute, *CustomerID* will not be saved to the database. *ColumnAttribute* has several properties that can be set to change the persistence behavior slightly. In the code example, the *Storage* property identifies the

private field that has the data. The *Name* property on *ColumnAttribute* can be set if, for example, the field name in the table does not match the property name.

The *CustomerID* property is also decorated by *DataMemberAttribute* to indicate to WCF services that this property's data can be serialized.

The property getter isn't doing anything other than returning the value of the private field. The setter has code that first attempts to call the partial *OnCustomerIDChanging* and *OnCustomerChanged* methods. If you decide to implement these methods, they will be called automatically to notify you before and after the change. The setter also has code to trigger the *PropertyChanging* and *PropertyChanged* events to notify anyone who has subscribed to these events.

An additional private field and public property for each child table is also referenced. In the *Customer* class, there is a private field and public property for the related orders because a customer has orders. The following code sample shows the private field and public property that represent the orders related to a customer.

Sample of Visual Basic Code

```
Private _Orders As EntitySet(Of [Order])

<Global.System.Data.Linq.Mapping.AssociationAttribute(Name:="Customer_Order", _
    Storage:="_Orders", ThisKey:="CustomerID", OtherKey:="CustomerID"), _
    Global.System.Runtime.Serialization.DataMemberAttribute(Order:=12, _
        EmitDefaultValue:=False)> _
Public Property Orders() As EntitySet(Of [Order])
    Get
        If (Me.serializing _
            AndAlso (Me._Orders.HasLoadedOrAssignedValues = False)) Then
            Return Nothing
        End If
        Return Me._Orders
    End Get
    Set(ByVal value As EntitySet(Of [Order]))
        Me._Orders.Assign(value)
    End Set
End Property
```

Sample of C# Code

```
private EntitySet<Order> _Orders;

[global::System.Data.Linq.Mapping.AssociationAttribute(Name="Customer_Order", Storage="_
Orders", ThisKey="CustomerID", OtherKey="CustomerID")]
[global::System.Runtime.Serialization.DataMemberAttribute(Order=12,
EmitDefaultValue=false)]
public EntitySet<Order> Orders
{
    get
    {
        if ((this.serializing && (this._Orders.HasLoadedOrAssignedValues == false)))
        {
            return null;
        }
    }
}
```



```

        return this._Orders;
    }
    set
    {
        this._Orders.Assign(value);
    }
}

```

At first glance, this code looks similar to the code example for *CustomerID*, but *ColumnAttribute* has been replaced by *AssociationAttribute*. This attribute identifies *Customers_Order* as the relationship that navigates from the *Customers* table to the *Orders* table. The attribute also identifies the key(s) used on the *Customers* and *Orders* tables.

The data type for *Orders* is a generic entity set of *Order*. The generic *EntitySet* is a specialized collection that provides deferred loading and relationship maintenance for the collection side of one-to-many and one-to-one relationships.

The getter has code to return nothing (C# *null*) if *Customer* is currently being serialized to keep from also serializing *Orders*. The getter also returns nothing (C# *null*) if no value has been assigned to this property or if this property has not been loaded.

The setter has simple code to pass the incoming value to the *Assign* method of the private field. *EntitySet* has a *ListChanged* event to which you can subscribe if you want to be notified when an assignment is made to this collection.

Examining the *DataContext* Class

The *NorthwindDataContext* class was created by the LINQ to SQL designer. This class inherits from the *DataContext* class that is part of the .NET Framework. The *DataContext* class is the main object for moving data to and from the database. You must instantiate the *NorthwindDataContext* class and then use its properties and methods to provide access to the database. To see the *DataContext* properties, click an empty area of the LINQ to SQL designer surface. Figure 4-5 shows the *DataContext* properties.

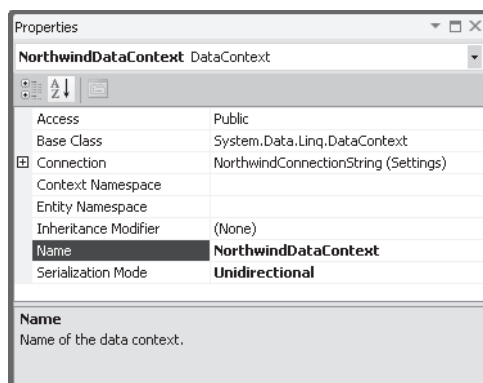


FIGURE 4-5 The *NorthwindDataContext* properties provide access to the connection string and other settings.

Of all the classes created by the LINQ to SQL designer, this is the only class that doesn't inherit from an object. The *Base Class* property provides the opportunity to create an intermediate class that inherits from *DataContext*, by which you add more functionality. You can then assign the intermediate class to the *Base Class* property.



EXAM TIP

You can expect to be tested on the *DataContext* class because it's explicitly called out in the exam objectives.

You also can set the namespace for the data context and entity classes so you can avoid naming collisions that could result if any of the created class names match the name of a class that already exists in your application.

If you are writing a WCF service, you might want to return instances of your entity classes from the service. This requires you to assign *DataContract* and *DataMember* attributes to the class and its properties by changing the *Serialization Mode* property from *None* to *Unidirectional*.

Looking at the *NorthwindDataContext* class that was produced by the LINQ to SQL designer, the following code example shows the class definition:

Sample of Visual Basic Code

```
<Global.System.Data.Linq.Mapping.DatabaseAttribute(Name:="Northwind")> _
Partial Public Class NorthwindDataContext
    Inherits System.Data.Linq.DataContext
    Private Shared mappingSource As System.Data.Linq.Mapping.MappingSource = _
        New AttributeMappingSource()
    'more members here
End Class
```

Sample of C# Code

```
[global::System.Data.Linq.Mapping.DatabaseAttribute(Name="Northwind")]
public partial class NorthwindDataContext : System.Data.Linq.DataContext
{
    private static System.Data.Linq.Mapping.MappingSource mappingSource = _
        new AttributeMappingSource();
    //more members here
}
```

This class is adorned with *DatabaseAttribute*, by which you specify the name of the database to which you will connect. This class inherits from *DataContext*.

This class also has a static field called *mappingSource*, which defaults to an instance of the *AttributeMappingSource* class. This field holds the mapping between the classes in the domain and the database as specified by attributes on the entity classes. You could opt to replace this object with an instance of *XmlMappingSource*, which would enable you to externalize the mappings to an XML file.

The *NorthwindDataContext* class contains a public property per type of entity class. The following code sample shows the *Customers* property:

Sample of Visual Basic Code

```
Public ReadOnly Property Customers() As System.Data.Linq.Table(Of Customer)
    Get
        Return Me.GetTable(Of Customer)()
    End Get
End Property
```

Sample of C# Code

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

Notice that the property type is the generic *Table* class of *Customer*. The *Table* class provides functionality for querying, inserting, updating, and deleting.

The *NorthwindDataContext* class also contains partial methods that you could implement for hooking into *insert*, *update*, and *delete* objects in any of the tables' properties on this class.

Managing Your Database Connection and Context Using *DataContext*

This section examines the connection string and how the *DataContext* object uses the connection string to connect to the database.

How LINQ to SQL Connects to Your Database

When you added items from Server Explorer, you automatically added the database connection string to your project as well. If you look in your config file, you will find the following connection string setting:

Config File

```
<connectionStrings>
  <add name="LinqToSqlSampleCode.Properties.Settings.NorthwindConnectionString"
        connectionString="Data Source=.;Initial Catalog=Northwind;Integrated
Security=True"
        providerName="System.Data.SqlClient" />
</connectionStrings>
```

LINQ to SQL uses the traditional ADO.NET *SqlConnection* class to open a connection to the SQL Server database. In this example, the data source property is set to a period, which means to connect to the local SQL Server instance. If you have only SqlExpress installed, your data source will be set to `.SQLEXPRESS`, which means you want to connect to the SqlExpress instance of SQL Server on your local machine.



EXAM TIP

For the exam, know that you must be on SQL Server 2000 or later to use LINQ to SQL because you will be tested on the requirements to use LINQ to SQL. You must be using .NET Framework 3.5 or later as well, and SQL Server 2000 has many limitations.

The *DataContext* object has a *Connection* property, and some of the constructors of the *NorthwindDataContext* accept a connection string. The following code sample shows the parameterless constructor for the *NorthwindDataContext* class:

Sample of Visual Basic Code

```
Public Sub New()  
    MyBase.New(Global.LinqToSqlSampleCode.MySettings.Default.NorthwindConnectionString, _  
        mappingSource)  
    OnCreated()  
End Sub
```

Sample of C# Code

```
public NorthwindDataContext() :  
    base(Global::LinqToSqlSampleCode.Properties.Settings.Default.  
        NorthwindConnectionString,  
        mappingSource)  
{  
    OnCreated();  
}
```

In this code example, the parameterless constructor makes a call to the base class (*DataContext*) constructor but is passing *NorthwindConnectionString*, which is in the configuration file. This means that you can instantiate the *NorthwindDataContext* without passing any parameter, and you automatically use the connection string that's in your config file. Also, you can easily change the connection string in the config file without requiring a rebuild of the application.

What's Sent to SQL Server, and When Is It Sent?

You might be wondering what kind of query is sent to SQL Server. Is the query efficient? When is the query sent to SQL Server? This section explores a simple LINQ to SQL query to answer these questions.

In the following code sample, a simple LINQ query is presented that retrieves a list of employees whose last names start with "D" and binds the result to a Windows Presentation Foundation (WPF) data grid.

Sample of Visual Basic Code

```
Private Sub mnuSimpleLinq_Click(ByVal sender As System.Object, _  
                                ByVal e As System.Windows.RoutedEventArgs)  
    Dim ctx = New NorthwindDataContext()  
    Dim employees = From emp In ctx.Employees  
                    Where emp.LastName.StartsWith("D")  
                    Select emp
```

```

        dg.ItemsSource = employees
    End Sub

```

Sample of C# Code

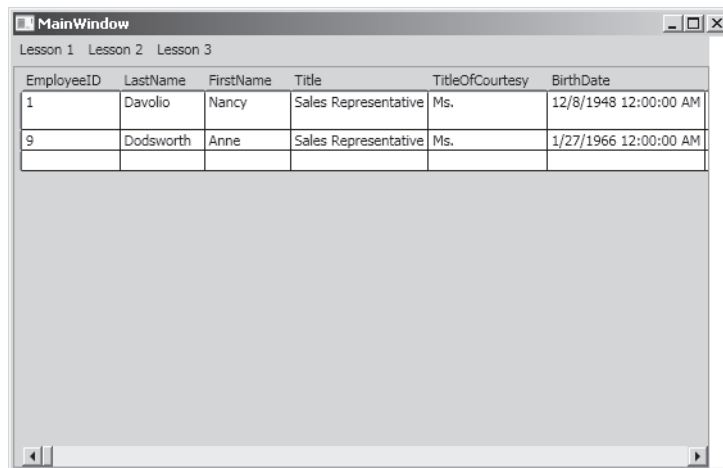
```

private void mnuSimpleLinq_Click(object sender, RoutedEventArgs e)
{
    var ctx = new NorthwindDataContext();
    var employees = from emp in ctx.Employees
                    where emp.LastName.StartsWith("D")
                    select emp;
    dg.ItemsSource = employees;
}

```

This example shows the use of the parameterless constructor to create the *NorthwindDataContext* object. There is no reference to a connection in this code. *NorthwindDataContext* has an *Employees* property that can be used in your LINQ query. The LINQ query that follows creates the *IQueryable<Employee>* query object; however, remember that LINQ query execution is deferred until the result of the query is enumerated. The last statement assigns the employees query object to the *ItemsSource* property on the WPF data grid. The data grid will enumerate the employees query object, which will cause the query to execute and retrieve the two employees whose last names start with "D."

When the LINQ to SQL query is created by initializing the *employees* variable, connection pooling is initialized, but nothing has executed yet. When the *employees* variable is assigned to the *ItemsSource* property of the data grid, the LINQ to SQL query is executed, and two employees' names are returned, as shown in Figure 4-6.



EmployeeID	LastName	FirstName	Title	TitleOfCourtesy	BirthDate
1	Davolio	Nancy	Sales Representative	Ms.	12/8/1948 12:00:00 AM
9	Dodsworth	Anne	Sales Representative	Ms.	1/27/1966 12:00:00 AM

FIGURE 4-6 Two employees are returned from the LINQ to SQL query. In this example, *AutoGenerateColumns* is set to *true*.

How did this query work? Did LINQ to SQL send a query to SQL Server to retrieve all the employees and then filter the employees within your application? How can you find the answers to these questions?

One way to find the answers is to set a breakpoint in your program on the statement that assigns the employees query to the data grid. Run the application and, when you reach the break point, hover over the *employees* variable, and you'll see a tool tip with the SQL query that will be sent to SQL Server; however, it's difficult to see the whole query within the small tool tip.

NOTE LINQ TO SQL DEBUG VISUALIZER

You can find various LINQ to SQL debug visualizers on the Internet. After installing one of these visualizers, you will see a magnifying glass when hovering over the variable. Clicking the magnifying glass typically displays a pop-up window with the query in a much more readable format.

Another way to find the answers is to use the *Log* property on *NorthwindDataContext*. This property accepts a *TextWriter* object and will write out all queries so you can create a *StreamWriter* object that references a file so you can write everything to a file. You can also assign a *StringWriter* to the *Log* property, which will send the SQL queries to a memory stream, and then you can display the contents. The following code sample shows the creation of a *StringWriter* that is assigned to the *Log* property, and its contents are displayed after the query is executed.

Sample of Visual Basic Code

```
Private Sub mnuSimpleLinq_Click(ByVal sender As System.Object, _
                                ByVal e As System.Windows.RoutedEventArgs)
    Dim ctx = New NorthwindDataContext()
    Dim sw = New StringWriter()
    ctx.Log = sw
    Dim employees = From emp In ctx.Employees
                    Where emp.LastName.StartsWith("D")
                    Select emp
    dg.ItemsSource = employees
    MessageBox.Show(sw.GetStringBuilder().ToString())
End Sub
```

Sample of C# Code

```
private void mnuSimpleLinq_Click(object sender, RoutedEventArgs e)
{
    var ctx = new NorthwindDataContext();
    var sw = new StringWriter();
    ctx.Log = sw;
    var employees = from emp in ctx.Employees
                    where emp.LastName.StartsWith("D")
                    select emp;
    dg.ItemsSource = employees;
    MessageBox.Show(sw.GetStringBuilder().ToString());
}
```

After running this code sample, a message box is displayed, as shown in Figure 4-7. This query is retrieving all columns from the Employees table, but the query includes a *where*

clause to provide the filtering at the database. SQL Server then performs the filtering and returns two rows to your application, thus providing efficient SQL for your LINQ to SQL query.

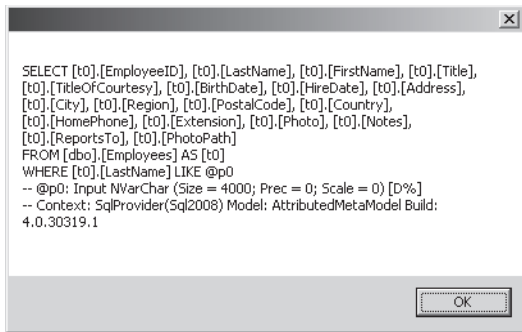


FIGURE 4-7 Here is the SQL query that is sent to SQL Server.

Finally, another way to see the queries sent to SQL Server is to use the SQL Server Profiler tool that comes with the Developer edition of SQL Server. The SQL Server Profiler can capture all SQL statements sent to SQL Server. Although this tool isn't included with SQL Server Express, it does work with that edition.

To use the SQL Server Profiler, you must have administrator privileges on SQL Server, or the SQL Server administrator can grant permissions for you to run the profiler tool. This tool can store the captured statements to a file or a database table. Figure 4-8 shows the output when running the sample LINQ to SQL code.

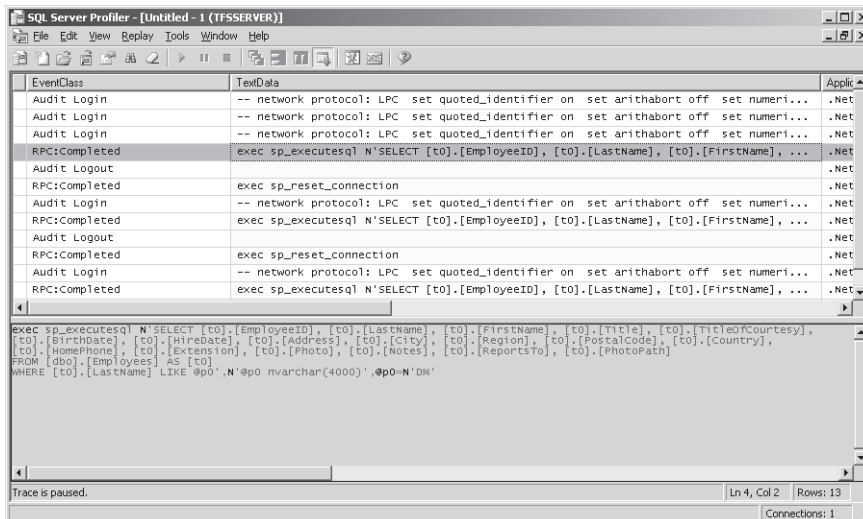


FIGURE 4-8 The SQL Server Profiler can capture all the traffic between the application and SQL Server.

The SQL Server Profiler can capture and display much more information than is shown in the *Log* property of *NorthwindDataContext*. In fact, in Figure 4-7, you can see that three select

statements were sent to SQL Server. The first select statement is highlighted, and it matches the statement that was shown when using the *Log* property. The second SQL statement has a *where* clause to return only EmployeeID=2, and the third SQL statement has a *where* clause to return EmployeeID=5. These two queries were caused by the data grid in an effort to retrieve the most recent value for the employees.

Eager Loading vs. Lazy Loading

When specifying properties or associations for which to query on your entity, you can perform *eager loading* or *lazy loading*. Lazy loading is also known as delay loading. Eager loading is also known as pre-fetch loading. The default behavior is to perform eager loading of the properties, which means that a property is loaded when a query is executed that references the property.

Lazy loading is configured in the LINQ to SQL designer by selecting an entity and then, in the Properties window, setting the *Delay Loaded* property to *true*. Figure 4-9 shows the *Delay Loaded* property.

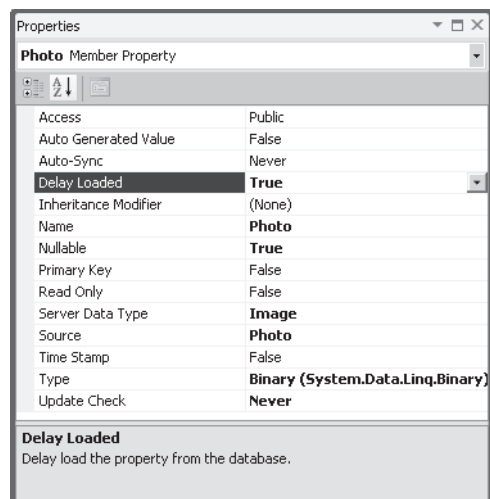


FIGURE 4-9 The *Delay Loaded* property can be set to *true* to perform lazy loading.

When using lazy loading, the property is not loaded until the property is accessed. When examining lazy loading, you need to think about performance and when you will take the performance hit. In one extreme, if every property were lazy loaded, there would be a cost associated with establishing the connection each time and transferring the data. To the user, this might make the application feel choppy or erratic. If you're fairly certain that you will use the data, why not pull the properties in one call? You take a big hit, maybe when a page is displayed to the user, but the page feels crisp afterward. The choice you make depends on how much data will be transferred and how certain you are that you will use the data. With lazy loading, you're making the decision to incur the performance cost to retrieve the data when you need it because you're fairly certain you won't need the data anyway. In

Figure 4-9, the Photo entry on the *Employee* entity is set to *Delay Loaded*. The following code example shows the effect of setting *Delay Loaded* to *true*:

Sample of Visual Basic Code

```
Private Sub mnuLazyLoading_Click(ByVal sender As System.Object, _
                                ByVal e As System.Windows.RoutedEventArgs)
    Dim ctx = New NorthwindDataContext()
    Dim sw = New StringWriter()
    ctx.Log = sw
    Dim employee = (From emp In ctx.Employees
                    Where emp.LastName.StartsWith("D")
                    Select emp).First()
    MessageBox.Show(sw.GetStringBuilder().ToString())

    sw = New StringWriter()
    ctx.Log = sw
    Dim photo = New MemoryStream(Employee.Photo.ToArray())
    MessageBox.Show(sw.GetStringBuilder().ToString())
End Sub
```

Sample of C# Code

```
private void mnuLazyLoading_Click(object sender, RoutedEventArgs e)
{
    var ctx = new NorthwindDataContext();
    var sw = new StringWriter();
    ctx.Log = sw;
    var employee = (from emp in ctx.Employees
                    where emp.LastName.StartsWith("Davolio")
                    select emp).First();
    MessageBox.Show(sw.GetStringBuilder().ToString());

    sw = new StringWriter();
    ctx.Log = sw;
    var photo = new MemoryStream(employee.Photo.ToArray());
    MessageBox.Show(sw.GetStringBuilder().ToString());
}
```

This code sample retrieves a single employee and displays the SQL statement that was generated. The code is then able to access the *Photo* property successfully, and the query that was run is displayed. Figure 4-10 shows the queries that were executed when this example code has been run.

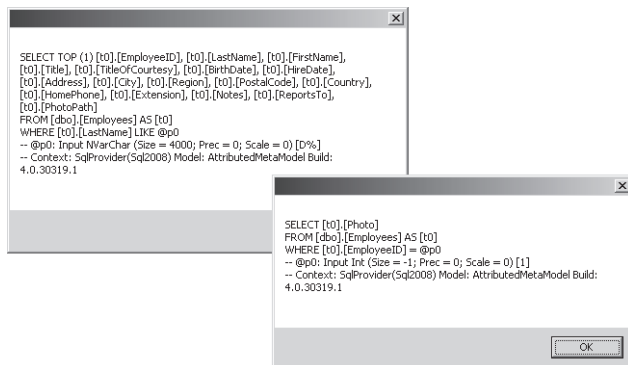


FIGURE 4-10 The SQL statements are shown that retrieve the employee name and then retrieve the employee's photo.

The first query that ran didn't include *Photo*, even though the LINQ expression requested the whole employee object. The *Photo* property was not included because the *Delay Loaded* property was set to *true*. When the *Photo* property was accessed, LINQ to SQL made a call to retrieve *Photo*. This is the second query displayed. When you use lazy loading, you're essentially betting that you're not going to need all columns' content, but if you do need the data, it will be automatically fetched for you.

Working with the LINQ to SQL Designer

In this practice, you create a new WPF application that accepts orders from customers, using the Northwind database. After creating the application, you use the LINQ to SQL designer to create an entity model for this application. In later exercises, you will add functionality to make the application operational.

This practice is intended to focus on the classes that have been defined in this lesson, so the graphical user interface (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 the Project and LINQ to SQL Entity Model

In this exercise, you create a WPF Application project and the entity model, using the Northwind database.

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

After Visual Studio .NET creates the project, the home page, *MainWindow.xaml*, is displayed.

4. In Solution Explorer, right-click the OrderEntryProject icon and choose Add | New Item. Select LINQ to SQL Classes and name the file **Northwind.dbml**. (Be sure to use the correct casing.)
5. Open Server Explorer by choosing View | Server Explorer.
6. Right-click the Data Connections icon and click Add Connection.
7. Depending on your Visual Studio configuration, you might be prompted with a window called Change Data Source. If so, select Microsoft SQL Server and click OK.
8. In the Add Connection window, at the Server Name prompt, type the name of the SQL Server instance. If you are using SQL Express on your local computer, type **./SqlExpress**.
9. At the Select Or Enter A Database Name prompt, select the Northwind database from the drop-down list and click OK. If you don't see the Northwind database in the drop-down list, install the Northwind database before going any further. If you don't have the Northwind database, a copy is included in the Chapter 4 sample code folder.
10. The Northwind database connection is now showing in the Server Explorer window. Beside the connection, click the plus sign to open the connection and then open the *Tables* node.
11. Drag the Customers, Orders, Order Details, and Products tables to the LINQ to SQL designer surface. Your window should look like the sample shown in Figure 4-11.

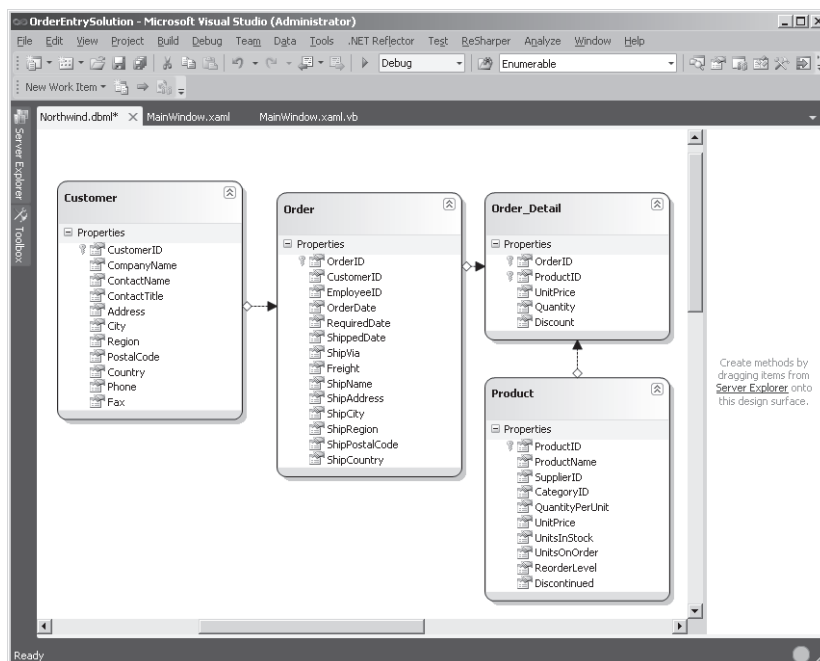


FIGURE 4-11 Here is the LINQ to SQL designer with the completed entity model.

12. Close and save the LINQ to SQL designer window.

Lesson Summary

This lesson provided detailed information about the LINQ to SQL designer.

- You can create an entity model easily by dragging and dropping database tables from Server Explorer.
- A table that is dropped on to the LINQ to SQL designer surface creates an entity class that represents each row in the table.
- You can drag and drop stored procedures to the LINQ to SQL designer surface, which creates methods that you can call from your application.
- Entity classes implement *INotifyPropertyChanging* and *INotifyPropertyChanged* to be tracked efficiently by the object tracking service.
- The *DataContext* object provides a property for each table and a method for each stored procedure.
- LINQ to SQL supports eager loading by default, but you can enable deferred loading, also known as lazy loading, to defer loading of the entity properties until you actually reference the property in your code.

Lesson Review

You can use the following questions to test your knowledge of the information in Lesson 1, "What Is LINQ to SQL?" 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. When working with LINQ to SQL, what is the main object that moves data to and from the database?
 - A. *DataSet*
 - B. *SqlDataAdapter*
 - C. *DataContext*
 - D. *Entity*

2. You want to use LINQ to SQL to run queries on a table that contains a column that stores large photos. Most of the time, you won't need to view the photo, but occasionally you will need to see it. In the LINQ to SQL designer, which property can you set on the photo column to get the efficient loading of the data for most scenarios but still be able to retrieve the photo when needed?
- A. *Skip*
 - B. *Delay Loaded*
 - C. *Take*
 - D. *Auto Generated Value*

Lesson 2: Executing Queries Using LINQ to SQL

In the previous lesson, you were introduced to the LINQ to SQL designer, the *DataContext* class, and an example entity class. In addition, a couple of LINQ to SQL queries were presented to demonstrate the operation of the *DataContext* class and lazy loading. You also saw how the LINQ to SQL provider queries only for the required data when *where* clauses are provided.

This lesson examines some of the more common types of LINQ to SQL queries you might perform.

After this lesson, you will be able to:

- Perform LINQ to SQL queries with filtering and sorting.
- Write LINQ to SQL statements that implement projections.
- Perform inner joins with LINQ to SQL.
- Perform outer joins with LINQ to SQL.
- Create and execute grouping and aggregation with LINQ to SQL.

Estimated lesson time: 45 minutes

Basic Query with Filter and Sort

Basic queries using the LINQ to SQL classes are very clean and readable. In addition, remember that LINQ to SQL classes retrieve only the data you request. The following code sample queries for a list of customers that contain the word "Restaurant" in the company name, sorted on postal code.

Sample of Visual Basic Code

```
private void mnuBasicQuery_Click(object sender, RoutedEventArgs e)
{
    var ctx = new NorthwindDataContext();
    var sw = new StringWriter();
    ctx.Log = sw;
    var customers = from c in ctx.Customers
                    where c.CompanyName.Contains("Restaurant")
                    orderby c.PostalCode
                    select c;
    dg.ItemsSource = customers;
    MessageBox.Show(sw.GetStringBuilder().ToString());
}
```

Sample of C# Code

```
private void mnuBasicQuery_Click(object sender, RoutedEventArgs e)
{
    var ctx = new NorthwindDataContext();
    var sw = new StringWriter();
    ctx.Log = sw;
```

```

var customers = from c in ctx.Customers
                where c.CompanyName.Contains("Restaurant")
                orderby c.PostalCode
                select c;
dg.ItemsSource = customers;
MessageBox.Show(sw.GetStringBuilder().ToString());
}

```

SQL Query

```

SELECT [t0].[CustomerID], [t0].[CompanyName], [t0].[ContactName], [t0].[ContactTitle],
[t0].[Address], [t0].[City], [t0].[Region], [t0].[PostalCode], [t0].[Country], [t0].
[Phone], [t0].[Fax]
FROM [dbo].[Customers] AS [t0]
WHERE [t0].[CompanyName] LIKE @p0
ORDER BY [t0].[PostalCode]
-- @p0: Input NVarChar (Size = 4000; Prec = 0; Scale = 0) [%Restaurant%]

```

The LINQ to SQL query sends a query to SQL Server that includes a *where* clause and an *order by* clause. Looking at the sample code, this looks like a very basic LINQ query, using the *Customers* property on the *NorthwindDataContext* object. The key here is that the LINQ to SQL provider is capable of constructing an efficient query to send to SQL Server.

Projections

One of the potential problems with the previous query is that all the column values from the *Customers* table are returned, but you might have needed to retrieve only *CustomerID*, *CompanyName*, and *PostalCode*. You can use a projection to limit the column values returned from SQL Server. The following code example demonstrates the use of projections to limit the returned column values from the *Customers* table.

Sample of Visual Basic Code

```

Private Sub mnuProjection_Click(ByVal sender As System.Object, _
                                ByVal e As System.Windows.RoutedEventArgs)
    Dim ctx = New NorthwindDataContext()
    Dim sw = New StringWriter()
    ctx.Log = sw
    Dim customers = From c In ctx.Customers
                    Where c.CompanyName.Contains("Restaurant")
                    Order By c.PostalCode
                    Select New With {c.CustomerID, c.CompanyName, c.PostalCode}
    dg.ItemsSource = customers
    MessageBox.Show(sw.GetStringBuilder().ToString())
End Sub

```

Sample of C# Code

```

private void mnuProjection_Click(object sender, RoutedEventArgs e)
{
    var ctx = new NorthwindDataContext();
    var sw = new StringWriter();
    ctx.Log = sw;
    var customers = from c in ctx.Customers

```

```

        where c.CompanyName.Contains("Restaurant")
        orderby c.PostalCode
        select new
        {
            c.CustomerID,
            c.CompanyName,
            c.PostalCode
        };
    dg.ItemsSource = customers;
    MessageBox.Show(sw.GetStringBuilder().ToString());
}

```

SQL Query

```

SELECT [t0].[CustomerID], [t0].[CompanyName], [t0].[PostalCode]
FROM [dbo].[Customers] AS [t0]
WHERE [t0].[CompanyName] LIKE @p0
ORDER BY [t0].[PostalCode]
-- @p0: Input NVarChar (Size = 4000; Prec = 0; Scale = 0) [%Restaurant%]

```

This code instantiates an anonymous type that filters out columns. Chapter 3, “Introducing LINQ,” covers anonymous types and projections in more detail.

Inner Joins

An inner join produces output only when the two tables you are joining match on the unique key to foreign key. Inner joins can be implemented easily with LINQ to SQL by using the standard LINQ query syntax. The following LINQ query produces an inner join of the Customers table to the Orders table and retrieves CustomerID, CompanyName, OrderID, and OrderDate by using query extension methods.

Sample of Visual Basic Code

```

Private Sub mnuInnerJoin1_Click(ByVal sender As System.Object, _
                                ByVal e As System.Windows.RoutedEventArgs)
    Dim ctx = New NorthwindDataContext()
    Dim sw = New StringWriter()
    ctx.Log = sw
    Dim customers = ctx.Customers.Join(ctx.Orders,
                                       Function(c) c.CustomerID,
                                       Function(o) o.CustomerID,
                                       Function(c, o) New With
                                       {
                                           c.CustomerID,
                                           c.CompanyName,
                                           o.OrderID,
                                           o.OrderDate
                                       }) _
        .OrderBy(Function(r) r.CustomerID) _
        .ThenBy(Function(r) r.OrderID)

    dg.ItemsSource = customers
    MessageBox.Show(sw.GetStringBuilder().ToString())
End Sub

```


Sample of C# Code

```
private void mnuInnerJoin1_Click(object sender, RoutedEventArgs e)
{
    var ctx = new NorthwindDataContext();
    var sw = new StringWriter();
    ctx.Log = sw;
    var customers = ctx.Customers.Join(
        ctx.Orders,
        c => c.CustomerID,
        o => o.CustomerID,
        (c, o) => new
        {
            c.CustomerID,
            c.CompanyName,
            o.OrderID,
            o.OrderDate
        })
        .OrderBy(r=>r.CustomerID)
        .ThenBy((r=>r.OrderID));
    dg.ItemsSource = customers;
    MessageBox.Show(sw.GetStringBuilder().ToString());
}
```

SQL Query

```
SELECT [t0].[CustomerID], [t0].[CompanyName], [t1].[OrderID], [t1].[OrderDate]
FROM [dbo].[Customers] AS [t0]
INNER JOIN [dbo].[Orders] AS [t1] ON [t0].[CustomerID] = [t1].[CustomerID]
ORDER BY [t0].[CustomerID], [t1].[OrderID]
```

Using query extension methods to perform the join produced a nice, clean SQL query. Could this query be written as a LINQ query? It can, and the following code sample produces the same result.

Sample of Visual Basic Code

```
Private Sub mnuInnerJoin2_Click(ByVal sender As System.Object, _
                                ByVal e As System.Windows.RoutedEventArgs)
    Dim ctx = New NorthwindDataContext()
    Dim sw = New StringWriter()
    ctx.Log = sw
    Dim customers = From c In ctx.Customers
                    Join o In ctx.Orders
                    On c.CustomerID Equals o.CustomerID
                    Order By c.CustomerID, o.OrderID
                    Select New With
                    {
                        c.CustomerID,
                        c.CompanyName,
                        o.OrderID,
                        o.OrderDate
                    }
    dg.ItemsSource = customers
    MessageBox.Show(sw.GetStringBuilder().ToString())
End Sub
```

Sample of C# Code

```
private void mnuInnerJoin_Click(object sender, RoutedEventArgs e)
{
    var ctx = new NorthwindDataContext();
    var sw = new StringWriter();
    ctx.Log = sw;
    var customers = from c in ctx.Customers
                    join o in ctx.Orders
                      on c.CustomerID equals o.CustomerID
                    orderby c.CustomerID, o.OrderID
                    select new
                    {
                        c.CustomerID,
                        c.CompanyName,
                        o.OrderID,
                        o.OrderDate
                    };
    dg.ItemsSource = customers;
    MessageBox.Show(sw.GetStringBuilder().ToString());
}
```

SQL Query

```
SELECT [t0].[CustomerID], [t0].[CompanyName], [t1].[OrderID], [t1].[OrderDate]
FROM [dbo].[Customers] AS [t0]
INNER JOIN [dbo].[Orders] AS [t1] ON [t0].[CustomerID] = [t1].[CustomerID]
ORDER BY [t0].[CustomerID], [t1].[OrderID]
```

This is a clean-looking LINQ query, and it produced a nice, efficient SQL query. If you look through the results carefully, you might find that there are two customers who have not placed any orders. How would you know that? These two customers are missing from the output because they don't match up to any orders. The missing customer IDs are FISSA and PARIS. To see all customers, you need to write an outer join.

Outer Joins

An outer join produces output of the outer table, even if the outer table element doesn't match the inner table. To perform an outer join, you must provide code to indicate that you still want the outer table row, even if there is no match to the inner table. You can perform outer joins by using the *GroupJoin* extension method, as shown in the following sample code:

Sample of Visual Basic Code

```
Private Sub mnuOuterJoin1_Click(ByVal sender As System.Object, _
                                ByVal e As System.Windows.RoutedEventArgs)
    Dim ctx = New NorthwindDataContext()
    Dim sw = New StringWriter()
    ctx.Log = sw
    Dim customers = ctx.Customers.GroupJoin(ctx.Orders, _
        Function(c) c.CustomerID, _
        Function(o) o.CustomerID, _
        Function(c, o) New With
        {
            c.CustomerID,
```

```

        c.CompanyName,
        .Orders = o
    }) _
    .SelectMany(Function(t) t.Orders.DefaultIfEmpty().Select( _
        Function(ord) New With
        {
            t.CompanyName,
            t.CustomerID,
            .OrderID = CType(ord.OrderID, Nullable(Of Integer)),
            .OrderDate = CType(ord.OrderDate, Nullable(Of DateTime))
        }) _
    .OrderBy(Function(r) r.CustomerID) _
    .ThenBy(Function(r) r.OrderID)
    dg.ItemsSource = customers
    MessageBox.Show(sw.GetStringBuilder().ToString())
End Sub

```

Sample of C# Code

```

private void mnuOuterJoin1_Click(object sender, RoutedEventArgs e)
{
    var ctx = new NorthwindDataContext();
    var sw = new StringWriter();
    ctx.Log = sw;
    var customers = ctx.Customers.GroupJoin(
        ctx.Orders,
        c => c.CustomerID,
        o => o.CustomerID,
        (c, o) => new
        {
            c.CustomerID,
            c.CompanyName,
            Orders = o
        })
    .SelectMany(t=>t.Orders.DefaultIfEmpty().Select(ord=>
        new
        {
            t.CompanyName,
            t.CustomerID,
            OrderID=(int?)ord.OrderID,
            OrderDate=(DateTime?) ord.OrderDate}))
    .OrderBy(r => r.CustomerID).ThenBy((r => r.OrderID));

    dg.ItemsSource = customers;
    MessageBox.Show(sw.GetStringBuilder().ToString());
}

```

SQL Query

```

SELECT [t2].[CompanyName], [t2].[CustomerID],
[t2].[value] AS [OrderID2], [t2].[value2] AS [OrderDate]
FROM (
SELECT [t1].[OrderID] AS [value], [t1].[OrderDate] AS [value2],
[t0].[CompanyName], [t0].[CustomerID]
FROM [dbo].[Customers] AS [t0]
LEFT OUTER JOIN [dbo].[Orders] AS [t1] ON [t0].[CustomerID] = [t1].[CustomerID]

```

```

    ) AS [t2]
ORDER BY [t2].[CustomerID], [t2].[value]

```

This code sample turned out to be ugly, primarily because the goal was to bind this to the data grid and see the same results as the inner join but with an extra row for FISSA and PARIS. In the SQL query, although a left outer join was performed, it was nested in a subquery, and the only result the outer query provides is a reordering of the fields.

You can also perform an outer join by using a LINQ query with the *into* keyword with the join. The following is a rewrite of the previous query, done as a LINQ query.

Sample of Visual Basic Code

```

Private Sub mnuOuterJoin2_Click(ByVal sender As System.Object, _
                                ByVal e As System.Windows.RoutedEventArgs)
    Dim ctx = New NorthwindDataContext()
    Dim sw = New StringWriter()
    ctx.Log = sw
    Dim customers = From c In ctx.Customers
                    Group Join o In ctx.Orders
                    On c.CustomerID Equals o.CustomerID Into InJoin = Group
                    From outJoin In InJoin.DefaultIfEmpty()
                    Order By c.CustomerID, outJoin.OrderID
                    Select New With
                    {
                        c.CustomerID,
                        c.CompanyName,
                        .OrderID = CType(outJoin.OrderID, Nullable(Of Integer)),
                        .OrderDate = CType(outJoin.OrderDate, Nullable(Of DateTime))
                    }
    dg.ItemsSource = customers
    MessageBox.Show(sw.GetStringBuilder().ToString())
End Sub

```

Sample of C# Code

```

private void mnuOuterJoin2_Click(object sender, RoutedEventArgs e)
{
    var ctx = new NorthwindDataContext();
    var sw = new StringWriter();
    ctx.Log = sw;
    var customers = from c in ctx.Customers
                    join o in ctx.Orders
                    on c.CustomerID equals o.CustomerID into inJoin
                    from outJoin in inJoin.DefaultIfEmpty()
                    orderby c.CustomerID, outJoin.OrderID
                    select new
                    {
                        c.CustomerID,
                        c.CompanyName,
                        OrderID = (int?)outJoin.OrderID,
                        OrderDate = (DateTime?)outJoin.OrderDate
                    };
    dg.ItemsSource = customers;
    MessageBox.Show(sw.GetStringBuilder().ToString());
}

```

SQL Query

```
SELECT [t0].[CustomerID], [t0].[CompanyName],  
[t1].[OrderID] AS [OrderID2], [t1].[OrderDate] AS [OrderDate]  
FROM [dbo].[Customers] AS [t0]  
LEFT OUTER JOIN [dbo].[Orders] AS [t1] ON [t0].[CustomerID] = [t1].[CustomerID]  
ORDER BY [t0].[CustomerID], [t1].[OrderID]
```

The LINQ query is much neater than the previous code example, which was implemented by extension methods. In addition, the SQL query is a nice, clean left outer join.

Grouping and Aggregation

LINQ to SQL also enables you to perform grouping operations to retrieve aggregate results. For example, you might want to retrieve the total amount of each order. To get the total of each order, get the sum of each order item in the Order_Details table. The following code sample shows how grouping and aggregation can solve this problem.

Sample of Visual Basic Code

```
Private Sub mnuAggregates_Click(ByVal sender As System.Object, _  
                                ByVal e As System.Windows.RoutedEventArgs)  
    Dim ctx = New NorthwindDataContext()  
    Dim sw = New StringWriter()  
    ctx.Log = sw  
    Dim orders = From o In ctx.Order_Details  
                  Group o By OrderID = o.OrderID Into grouped = Group  
                  Select New With  
                  {  
                      .OrderID = OrderID,  
                      .Total = grouped.Sum(Function(line) _  
                                             line.Quantity * line.UnitPrice * _  
                                             (1 - CType(line.Discount, Decimal)))  
                  }  
    dg.ItemsSource = orders  
    MessageBox.Show(sw.GetStringBuilder().ToString())  
End Sub
```

Sample of C# Code

```
private void mnuAggregates_Click(object sender, RoutedEventArgs e)  
{  
    var ctx = new NorthwindDataContext();  
    var sw = new StringWriter();  
    ctx.Log = sw;  
    var orders = from o in ctx.Order_Details  
                  group o by o.OrderID  
                  into grouped  
                  select new  
                  {  
                      OrderID = grouped.Key,  
                      Total = grouped.Sum(  
                          line=>line.Quantity * line.UnitPrice *  
                          (1 - (decimal)line.Discount))  
                  };  
}
```

```

        dg.ItemsSource = orders;
        MessageBox.Show(sw.GetStringBuilder().ToString());
    }

```

SQL Query

```

SELECT SUM([t1].[value]) AS [Total], [t1].[OrderID]
FROM (
    SELECT (CONVERT(Decimal(29,4),[t0].[Quantity])) * [t0].[UnitPrice] *
        (@p0 - (CONVERT(Decimal(33,4),[t0].[Discount]))) AS [value], [t0].[OrderID]
    FROM [dbo].[Order Details] AS [t0]
    ) AS [t1]
GROUP BY [t1].[OrderID]
-- @p0: Input Decimal (Size = -1; Prec = 33; Scale = 4) [1]

```

This code sample grouped the *Order_Details* rows by *OrderID* and then calculated the total of each order by calculating the sum of the line items of the order. To calculate the sum, each line had to be calculated by multiplying the quantity by the unit price and then multiplying by one minus the discount.

Paging

When writing an application that queries thousands or millions of rows of data, you will often run into problems when a query returns many more rows of data than you could possibly display. Having said that, what's the sense of waiting for all that data to be shipped from SQL Server to your application? For example, maybe you queried for the customers whose names begin with the letter A, but you didn't realize that this would return ten thousand rows of data.

Paging can be a useful way to minimize the amount of data returned from a query so you can see the first part of the data quickly and decide whether you want to continue viewing more data. To implement paging, you can use the *Skip* and *Take* extension methods with your LINQ query. In the following sample code, a scrollbar has been added to the WPF form, and its settings are configured to match the quantity of pages of customers. When scrolling, the *Scroll* event is triggered, and you see the previous or next page of customers. This code sample uses the *Skip* and *Take* methods:

Sample of Visual Basic Code

```

Private Const pageSize As Integer = 25
Private pageCount As Integer
Private customerCount As Integer
Private customers As IQueryable(Of Tuple(Of String, String))
Private sw As New StringWriter()

Private Sub mnuPaging_Click(ByVal sender As System.Object, _
    ByVal e As System.Windows.RoutedEventArgs)
    Dim ctx = New NorthwindDataContext()
    ctx.Log = sw
    customers = From c In ctx.Customers
                Order By c.CompanyName

```

```

        Select New Tuple(Of String, String)(c.CustomerID, c.CompanyName)

customerCount = customers.Count()
pageCount = customerCount / pageSize
If (pageCount * pageSize < customerCount) Then pageCount += 1

scrData.Minimum = 0
scrData.Maximum = pageCount
scrData.Visibility = Visibility.Visible
scrData.SmallChange = 1
scrData.Scroll(Nothing, Nothing)
End Sub

Private Sub scrData_Scroll(ByVal sender As System.Object, _
    ByVal e As System.Windows.Controls.Primitives.ScrollEventArgs)
    Dim customersDisplay = From c In customers
        Select New With {.ID = c.Item1, .Name = c.Item2}
    dg.ItemsSource = customersDisplay.Skip(CInt(scrData.Value) * pageSize).Take(pageSize)
End Sub

```

Sample of C# Code

```

private const int pageSize = 25;
private int pageCount;
private int customerCount;
private IQueryable<Tuple<string,string>> customers;
StringWriter sw = new StringWriter();

private void mnuPaging_Click(object sender, RoutedEventArgs e)
{
    var ctx = new NorthwindDataContext();
    ctx.Log = sw;
    customers = from c in ctx.Customers
        orderby c.CompanyName
        select
            new Tuple<string,string>(c.CustomerID,c.CompanyName);

    customerCount = customers.Count();
    pageCount = customerCount / pageSize;
    if (pageCount * pageSize < customerCount) pageCount++;

    scrData.Minimum = 0;
    scrData.Maximum = pageCount;
    scrData.Visibility = Visibility.Visible;
    scrData.SmallChange = 1;
    scrData.Scroll(null, null);
}

private void scrData_Scroll(object sender,
    System.Windows.Controls.Primitives.ScrollEventArgs e)
{
    var customersDisplay = from c in customers
        select new {ID = c.Item1, Name = c.Item2};
    dg.ItemsSource = customersDisplay.Skip((int)scrData.Value * pageSize).Take(pageSize);
}

```

SQL Query

```
SELECT COUNT(*) AS [value]
FROM [dbo].[Customers] AS [t0]
-- Context: SqlProvider(Sql2008) Model: AttributedMetaModel Build: 4.0.30319.1

SELECT [t1].[CustomerID] AS [item1], [t1].[CompanyName] AS [item2]
FROM (
    SELECT ROW_NUMBER() OVER (
        ORDER BY [t0].[CompanyName]) AS [ROW_NUMBER],
        [t0].[CustomerID], [t0].[CompanyName]
    FROM [dbo].[Customers] AS [t0]
    ) AS [t1]
WHERE [t1].[ROW_NUMBER] BETWEEN @p0 + 1 AND @p0 + @p1
ORDER BY [t1].[ROW_NUMBER]
-- @p0: Input Int (Size = -1; Prec = 0; Scale = 0) [0]
-- @p1: Input Int (Size = -1; Prec = 0; Scale = 0) [25]
-- Context: SqlProvider(Sql2008) Model: AttributedMetaModel Build: 4.0.30319.1
```

Of the SQL queries that were run, the first query was to get the count of customers, which was executed when the *customers.Count()* call was made. The next SQL query was to retrieve the first page of customers. In this query, the two parameters are *@p0*, the current page, and *@p1*, the page size. These parameters are set to 0 and 25, respectively. The query itself uses the *ROW_NUMBER* function available in SQL Server 2005 and later, which is why using LINQ to SQL on SQL Server 2000 has limitations. As you page up and down, the second query executes but will substitute a different value for the current page (*@p0*).

In the Visual Basic 2010 and C# code, variables and constants are being defined outside the method for these variables to be accessible in all methods. The first method, *mnuPaging_Click*, executes when you select the Paging menu option. This method retrieves the count of customers and sets the LINQ to SQL query; it also configures the scroll bar and sets it to be visible. The next method, *scrData_Scroll*, is executed each time you click the scroll bar. This method retrieves the current page value from the scroll bar and uses the *Skip* and *Take* methods to retrieve a single page of data.

Writing LINQ Queries to Display Data

In this practice, you continue the order entry application from Lesson 1, “What Is LINQ to SQL?” by adding a GUI and then LINQ queries to populate the GUI with data.

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

EXERCISE 1 Add the GUI

In this exercise, you modify the WPF application you created in Lesson 1 by creating the GUI. In the next exercise, you add the LINQ queries to populate the screen.

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, double-click the `MainWindow.xaml` file to open the file in the WPF Form Designer window.
3. On the Window tab, add a *Loaded* event handler. When prompted for a new *EventHandler*, double-click the New *EventHandler* option. Your XAML should look like the following:

Sample of Visual Basic XAML

```
<Window x:Class="MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Loaded="Window_Loaded"
        Title="MainWindow" Height="350" Width="525">
    <Grid>
</Grid>
</Window>
```

Sample of C# XAML

```
<Window x:Class="OrderEntryProject.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Loaded="Window_Loaded"
        Title="MainWindow" Height="350" Width="525">
    <Grid>
</Grid>
</Window>
```

4. The XAML code contains a *Grid* definition. Inside the grid, add three row definitions. The first two rows should have their *Height* property set to *Auto*, and the last row should have its *Height* property set to *"*"*. Regardless of your programming language, your XAML for the grid should look like the following:

XAML

```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto" />
        <RowDefinition Height="Auto" />
        <RowDefinition Height="*" />
    </Grid.RowDefinitions>
</Grid>
```

5. In the XAML, before the end of the *Grid*, add a *Menu*. Inside the menu, add *MenuItem* elements for Save called **mnuSave**, New Order called **mnuOrder**, and Exit called **mnuExit**. After adding these items, double-click each menu item to add the click event handler code. Your XAML should look like the following.

XAML

```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto" />
        <RowDefinition Height="Auto" />
        <RowDefinition Height="*" />
    </Grid.RowDefinitions>
```

```

</Grid.RowDefinitions>
<Menu>
  <MenuItem Header="Save" Name="mnuSave" Click="mnuSave_Click" />
  <MenuItem Header="New Order" Name="mnuOrder" Click="mnuOrder_Click" />
  <MenuItem Header="Exit" Name="mnuExit" Click="mnuExit_Click" />
</Menu>
</Grid>

```

6. In XAML, under *Menu*, add a combo box called **cmbCustomers**. Configure the combo box to be in *Grid.Row="1"*. Under that, add a list box called **lstOrders**. Configure *ListBox* to be in *Grid.Row="2"*. Set the *Margin* property of both items to 5. Double-click the combo box to add a *SelectionChanged* event handler to your code. Your XAML should look like the following:

XAML

```

<Grid>
  <Grid.RowDefinitions>
    <RowDefinition Height="Auto" />
    <RowDefinition Height="Auto" />
    <RowDefinition Height="*" />
  </Grid.RowDefinitions>
  <Menu>
    <MenuItem Header="Save" Name="mnuSave" Click="mnuSave_Click" />
    <MenuItem Header="New Order" Name="mnuOrder" Click="mnuOrder_Click"/>
    <MenuItem Header="Exit" Name="mnuExit" Click="mnuExit_Click" />
  </Menu>
  <ComboBox Grid.Row="1" Name="cmbCustomers" Margin="5"
    SelectionChanged="cmbCustomers_SelectionChanged"/>
  <ListBox Grid.Row="2" Name="lstOrders" Margin="5"/>
</Grid>

```

7. Extend markup in the list box element to configure a custom template to display *OrderID*, *OrderDate*, and *RequiredDate* from the *Orders* table. This will require the *ListBox* element to be converted to have separate start and end tags. Your XAML for the list box should look like the following, regardless of programming language:

XAML

```

<ListBox Grid.Row="2" Margin="5" Name="lstOrders">
  <ListBox.ItemTemplate>
    <DataTemplate>
      <Border CornerRadius="5" BorderThickness="2"
        BorderBrush="Blue" Margin="3">
        <StackPanel Orientation="Horizontal">
          <TextBlock Text="Order #"
            TextAlignment="Right" Width="40"/>
          <TextBlock Name="txtOrderID"
            Text="{Binding Path=OrderID}" Margin="5,0,10,0"
            Width="30"/>
          <TextBlock Text="Order Date:"
            TextAlignment="Right" Width="80"/>
          <TextBlock Name="txtOrderDate"
            Text="{Binding Path=OrderDate, StringFormat={}{0:MM/dd/yyyy}}"
            Margin="5,0,10,0" Width="75"/>

```

```

        <TextBlock Text="Required Date:"
            TextAlignment="Right" Width="80"/>
        <TextBlock Name="txtRequiredDate"
            Text="{Binding Path=RequiredDate, StringFormat={}{0:MM/dd/
yyyy}}}"
            Margin="5,0,10,0" Width="75"/>
    </StackPanel>
</Border>
</DataTemplate>
</ListBox.ItemTemplate>
</ListBox>

```

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

Although you haven't written any Visual Basic 2010 or C# code yet, you have entered enough XAML to produce a screen that can show a customer list in the combo box and the customer's orders in the data grid. Your main window should look like the sample in Figure 4-12. In the next exercise, you will add code to populate the window.

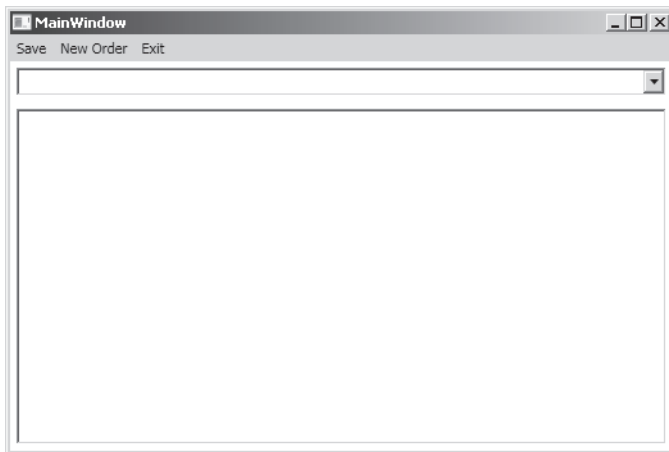


FIGURE 4-12 This is the completed GUI that will display customers and orders.

EXERCISE 2 Creating LINQ Queries to Display Data

In this exercise, you add code that includes LINQ queries to the WPF application you worked on in Exercise 1, "Add the GUI," of this lesson. This produces a running application for viewing the data, but in the next lesson, you'll add code to add an order to a customer and save it.

1. In Visual Studio .NET 2010, choose File | Open | Project. Open the project from the preceding exercise.
2. In Solution Explorer, double-click the *MainWindow.xaml.vb* or *MainWindow.xaml.cs* file to open the code-behind file for the main window.
3. Add a private field to the top of the *MainWindow* class, called *ctx*, and set its type to *NorthwindDataContext*; also, instantiate the class as shown in the following code sample:

Sample of Visual Basic Code

```
Private ctx As New NorthwindDataContext();
```

Sample of C# Code

```
private NorthwindDataContext ctx = new NorthwindDataContext();
```

4. In the *Window_Loaded* event handler method, add code to save *ctx* to an application property called *ctx*, which makes the *ctx* object accessible from other windows. Also, add a LINQ query to populate *cmbCustomers* with a *Tuple* of *string*, a *string* that contains *CustomerID* and *CompanyName* from the *Customers* table. Set the *DisplayMemberPath* to display *Item2* of the *Tuple*. The *Window_Loaded* event handler should look like the following:

Sample of Visual Basic Code

```
Private Sub Window_Loaded(ByVal sender As System.Object, _
                          ByVal e As System.Windows.RoutedEventArgs)
    Application.Current.Properties("ctx") = ctx
    cmbCustomers.ItemsSource = From c In ctx.Customers
                              Select New Tuple(Of String, String)( _
                                  c.CustomerID, _
                                  c.CompanyName)
    cmbCustomers.DisplayMemberPath = "Item2"
End Sub
```

Sample of C# Code

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
    App.Current.Properties["ctx"] = ctx;
    cmbCustomers.ItemsSource = from c in ctx.Customers
                              select new Tuple<string,string>
                                  (
                                      c.CustomerID,
                                      c.CompanyName
                                  );
    cmbCustomers.DisplayMemberPath = "Item2";
}
```

5. Add code to the *SelectionChanged* event handler method to retrieve the select customer information and use it to write a LINQ query for a list of *OrderID*, *OrderDate*, and *RequestDate* from the *Orders* table. Your code should look like the following.

Sample of Visual Basic Code

```
Private Sub cmbCustomers_SelectionChanged(ByVal sender As System.Object, _
                                         ByVal e As System.Windows.Controls.SelectionChangedEventArgs)
    Dim customer = CType(cmbCustomers.SelectedValue, Tuple(Of String, String))
    If (customer Is Nothing) Then Return
    lstOrders.ItemsSource =
        From o In ctx.Orders
        Where o.CustomerID = customer.Item1
        Select New With _
        { _
            o.OrderID, _
```

```

        o.OrderDate, _
        o.RequiredDate _
    }
End Sub

```

Sample of C# Code

```

private void cmbCustomers_SelectionChanged(object sender,
    SelectionChangedEventArgs e)
{
    var customer = (Tuple<string,string>)cmbCustomers.SelectedValue;
    if (customer == null) return;
    lstOrders.ItemsSource =
        from o in ctx.Orders
        where o.CustomerID == customer.Item1
        select new
        {
            o.OrderID,
            o.OrderDate,
            o.RequiredDate
        };
}

```

6. In the *Exit* event handler method, add code to end the application. Your code should look like the following:

Sample of Visual Basic Code

```

Private Sub mnuExit_Click(ByVal sender As System.Object, _
    ByVal e As System.Windows.RoutedEventArgs)
    Application.Current.Shutdown()
End Sub

```

Sample of C# Code

```

private void mnuExit_Click(object sender, RoutedEventArgs e)
{
    Application.Current.Shutdown();
}

```

7. Choose Debug | Start Debugging to run the application. If you select a customer from the combo box, the list box will populate with the list of OrderID, OrderDate, and RequiredDate. Click Exit to shut down the application.

Lesson Summary

This lesson provided detailed information about how to run LINQ to SQL queries.

- You can use the table properties on the *DataContext* object to run LINQ queries.
- If you provide a *where* clause, LINQ to SQL will create a SQL query that includes the *where* clause to limit the rows returned to the application.
- If you provide a projection in your *select* clause to limit the properties that are selected, LINQ to SQL will create a SQL query that includes a matching column filter to limit the data returned to the application.

- LINQ to SQL supports both inner and outer joins.
- LINQ to SQL supports grouping and aggregation.
- You can use the *Skip* and *Take* extension methods to implement paging over data.

Lesson Review

You can use the following questions to test your knowledge of the information in Lesson 2, “Executing Queries Using LINQ to SQL.” 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. Which query extension methods can you use to implement paging over a LINQ to SQL query?
 - A. *Contains* and *Intersect*
 - B. *GroupBy* and *Last*
 - C. *Skip* and *Take*
 - D. *First* and *Last*
2. When using a LINQ query to join two tables, you must specify how the two tables are related by using which keyword?
 - A. *let*
 - B. *equals*
 - C. *into*
 - D. *by*

Lesson 3: Submitting Changes to the Database

Until now, the lessons have been focused on data retrieval, so you might be wondering about how to change data. LINQ itself is a query-based tool for data retrieval, but the LINQ to SQL classes do provide means for you perform insert, update, and delete operations on your data. This lesson focuses on that.

After this lesson, you will be able to:

- Understand how LINQ to SQL tracks changes and caches objects.
- Know the life cycle a LINQ to SQL entity.
- Modify LINQ to SQL entities.
- Add new LINQ to SQL entities.
- Delete LINQ to SQL entities.
- Execute stored procedures using LINQ to SQL classes.
- Submit all changes back to the database server.
- Submit changes within a transaction.

Estimated lesson time: 60 minutes

In the .NET Framework, classes are reference types, and an instance of a class is called an *object*. Each object has a unique identity, and many variables can refer to the same object. If two variables are referring to the same object, changes made through one variable are also visible through the other variable.

When working with relational database tables, a row has a primary key that is unique, so no two rows may share the same primary key. This is a constraint in the table. As long as you are working in the table, the primary key certainly feels like the unique identifier that objects have.

You start running into problems when retrieving data from the table for use in your application. When the data is retrieved as rows, there is no expectation that two rows representing the same data actually correspond to the same row objects that were retrieved. For example, if you query for a specific customer twice, you get two rows of data that contain the same information.

Using *DataContext* to Track Changes and Cache Objects

When working with objects, you typically expect that if you ask *DataContext* for the same information again, it will give you the same object instance. Certainly, this is the behavior you want, right? You want to make sure that if you have two variables that reference customer number 123, and you go through one of the variables to change the customer name, you will

see the new name when going through the other variable to view the name. You don't want duplicate objects in memory just because you queried for the same customer twice.

The *DataContext* object manages object identity for you so that rows retrieved from the database table are automatically logged in the *DataContext* object's internal identity table by the row's primary key when the object is created. If the same row is retrieved again by this *DataContext* object, the original instance is handed back to you. Your application sees only the state of the row that was first retrieved. If, for example, you retrieve a row in your application and the row is modified in the database table by a different application, you won't get the updated data if you query for this row again because the row is delivered to you from the *DataContext* object's cache.

It might seem rather weird that querying for the same row returned the original state of the row, even though the row had been changed by another application. In essence, the new data was thrown away. The reason for this behavior is that LINQ to SQL needs this behavior to support optimistic concurrency. *DataContext* contains a method called *SubmitChanges*, which sends all your changes back to the database table. This is explained in more detail later in this lesson.

If the database contains a table without a primary key, LINQ to SQL allows queries to be submitted over the table, but it doesn't allow updates because the framework cannot identify which row to update, given the lack of a unique key.



EXAM TIP

The exam will contain questions related to modifying data using the *DataContext* class, so be sure you understand the role of *DataContext* when changes need to be sent to SQL Server.

If the object requested by the query is identified as one that has already been retrieved, no query is executed at all. The identity table acts as a cache, storing all previously retrieved objects.

The Life Cycle of an Entity

Consider the following scenario in which an object is retrieved by a LINQ to SQL query and modified, maybe several times, until your application is ready to send the changes back to the server. You might repeat this scenario until your application no longer needs the entity object information. If the entity object is no longer referenced by your application, the object will be reclaimed by the garbage collector, just like any other .NET Framework object. As long as you submitted the changes to the *DataContext* object by using its *SubmitChanges* method, the data is sent and remains in the database. Because of the database persistence, you can query the *DataContext* object for the same data, and you will receive an object that appears and behaves like the original object that was garbage-collected. From that perspective, the *DataContext* object provides the illusion that the lifetime of the object is beyond any single run-time instantiation.

This section focuses on a single instantiation of an entity object so that a cycle refers to the time span of that object instance within a particular run-time context. This cycle starts when the *DataContext* object becomes aware of a new instance and ends when the object or *DataContext* object is no longer needed.

When you retrieve objects from SQL Server by using the *DataContext* object, LINQ to SQL automatically stores information about the entity objects. The *DataContext* object provides an object-tracking service that tracks the state of your objects. LINQ to SQL objects always have a state, and Table 4-1 shows the states a LINQ to SQL object can have. In terms of performance and resource usage, object tracking using the *identity tracking service* is not costly, but making changes to the object by using the *change tracking service* can be costly.

TABLE 4-1 The States Available to LINQ to SQL Objects

STATE	DESCRIPTION
<i>Untracked</i>	An object not tracked by LINQ to SQL due to one of the following reasons: You instantiated the object yourself. The object was created through deserialization. The object came from a different <i>DataContext</i> object.
<i>Unchanged</i>	An object retrieved by using the current <i>DataContext</i> object and not known to have been modified since it was created.
<i>PossiblyModified</i>	An object that is attached to a <i>DataContext</i> object will be in this state unless you specify otherwise when you attach.
<i>ToBeInserted</i>	An object not retrieved by using the current <i>DataContext</i> object, which will send an <i>insert</i> statement to the database.
<i>ToBeUpdated</i>	An object known to have been modified since it was retrieved, which sends an <i>update</i> statement to the database.
<i>ToBeDeleted</i>	An object marked for deletion, which will send a <i>delete</i> statement to the database.
<i>Deleted</i>	An object that has been deleted in the database; this state is final and does not allow for additional transitions.

The following code retrieves a customer from the Northwind database. This customer will be used in the examples that follow.

Sample of Visual Basic Code

```
Dim customer As Customer
Dim ctx As New NorthwindDataContext()
Private Sub mnuCreateEntity_Click(ByVal sender As System.Object, _
                                ByVal e As System.Windows.RoutedEventArgs)

    Dim sw = New StringWriter()
    ctx.Log = sw
    customer = (From c In ctx.Customers
```

```

        Where c.CustomerID = "ALFKI"
        Select c).First()
    MessageBox.Show(sw.GetStringBuilder().ToString())
End Sub

```

Sample of C# Code

```

private Customer customer;
private NorthwindDataContext ctx = new NorthwindDataContext();
private void mnuCreateEntity_Click(object sender, RoutedEventArgs e)
{
    var sw = new StringWriter();
    ctx.Log = sw;
    customer = (from c in ctx.Customers
                where c.CustomerID == "ALFKI"
                select c).First();
    MessageBox.Show(sw.GetStringBuilder().ToString());
}

```

SQL Query

```

SELECT TOP (1) [t0].[CustomerID], [t0].[CompanyName],
[t0].[ContactName], [t0].[ContactTitle],
[t0].[Address], [t0].[City], [t0].[Region],
[t0].[PostalCode], [t0].[Country], [t0].[Phone], [t0].[Fax]
FROM [dbo].[Customers] AS [t0]
WHERE [t0].[CustomerID] = @p0
-- @p0: Input NVarChar (Size = 4000; Prec = 0; Scale = 0) [ALFKI]

```

In this example, there is a small difference in behavior between Visual Basic 2010 and C#. In Visual Basic 2010, every time you click the menu option to run this code, the query is sent to SQL Server. In C#, on the first click of the menu option, the SQL query is run, but subsequent clicks simply return the existing object that's in the cache. In either case, you won't see any changes that are made to the row in the database. Weird? Both languages use the same underlying .NET Framework classes, so you would think the behavior should be the same. If you reverse-engineer the compiled method, you'll find that the Visual Basic and C# code produce very different Intermediate Language (IL). Also, if you reverse-engineer the compiled Northwind.Designer.vb and the Northwind.Designer.cs code, you'll see that they produce very different IL code as well. The point is that the underlying framework is the same, but the designer-generated code, and the compiled C# and Visual Basic code, are different.

Modifying Existing Entities

When LINQ to SQL creates a new object, the object is in *Unchanged* state. If you modify the object, the *DataContext* object will change the state of the object to the *ToBeUpdated* state. This is what the *DataContext* object uses to determine which objects must be persisted when you submit your changes.

How does the *DataContext* object know when you change your object? Change notifications are accomplished through the *PropertyChanging* event that's in property setters. When

DataContext receives a change notification, it creates a copy of the object and changes its state to *ToBeUpdated*.

Recall that, as explained in Lesson 1, the entity class implements the *INotifyPropertyChanging* interface. You could create an entity class that does not implement *INotifyPropertyChanging*, and, in that scenario, *DataContext* maintains a copy of the values that objects had when they were first instantiated. When you call *SubmitChanges*, *DataContext* will compare the current and original values, field by field, to decide whether the object has been changed.

The following code sample modifies the customer object that was retrieved in the previous code example and then submits the change back to the database by using the *SubmitChanges* method on the *DataContext* object. Be sure to run the previous example to create the customer object first.

Sample of Visual Basic Code

```
Private Sub mnuModifyEntity_Click(ByVal sender As System.Object, _
                                ByVal e As System.Windows.RoutedEventArgs)
    customer.ContactName = "Marty " + DateTime.Now.ToLongTimeString()
    ctx.SubmitChanges()
End Sub
```

Sample of C# Code

```
private void mnuModifyEntity_Click(object sender, RoutedEventArgs e)
{
    customer.ContactName = "Marty " + DateTime.Now.ToLongTimeString();
    ctx.SubmitChanges();
}
```

SQL Query

```
UPDATE [dbo].[Customers]
SET [ContactName] = @p10
WHERE ([CustomerID] = @p0) AND
      ([CompanyName] = @p1) AND
      ([ContactName] = @p2) AND
      ([ContactTitle] = @p3) AND
      ([Address] = @p4) AND
      ([City] = @p5) AND
      ([Region] IS NULL) AND
      ([PostalCode] = @p6) AND
      ([Country] = @p7) AND
      ([Phone] = @p8) AND
      ([Fax] = @p9)
-- @p0: Input NChar (Size = 5; Prec = 0; Scale = 0) [ALFKI]
-- @p1: Input NVarChar (Size = 4000; Prec = 0; Scale = 0) [Alfreds Futterkiste]
-- @p2: Input NVarChar (Size = 4000; Prec = 0; Scale = 0) [Maria Anders]
-- @p3: Input NVarChar (Size = 4000; Prec = 0; Scale = 0) [Sales Representative]
-- @p4: Input NVarChar (Size = 4000; Prec = 0; Scale = 0) [Obere Str. 57]
-- @p5: Input NVarChar (Size = 4000; Prec = 0; Scale = 0) [Berlin]
-- @p6: Input NVarChar (Size = 4000; Prec = 0; Scale = 0) [12209]
-- @p7: Input NVarChar (Size = 4000; Prec = 0; Scale = 0) [Germany]
-- @p8: Input NVarChar (Size = 4000; Prec = 0; Scale = 0) [030-0074321]
```

```
-- @p9: Input NVarChar (Size = 4000; Prec = 0; Scale = 0) [030-0076545]
-- @p10: Input NVarChar (Size = 4000; Prec = 0; Scale = 0) [Marty 11:44:30 AM]
-- Context: SqlProvider(Sql2008) Model: AttributedMetaModel Build: 4.0.30319.1
```

In this code sample, the *where* clause in the SQL query is not simply looking for a matching primary key; it's comparing all fields to find a match, and it's using the original values to perform the search. If someone modified that row in the database, your application will not find a match. This will generate a concurrency error.

If you make a change to a relationship, the object that has the foreign key, which is typically the child object, will have the authoritative information about the parent. Optionally, you can have a reference from the parent to the child. It's the generic *EntitySet* of *TEntity* and *EntityRef* of *TEntity* that maintain bidirectional references to ensure consistency of one-to-many and one-to-one relationships.

Adding New Entities to *DataContext*

A new object you instantiate yourself is unknown to *DataContext* and is in *Untracked* state because no *DataContext* class is aware of your object. You can use the *InsertOnSubmit* method, or you can assign your new object to an object that is already attached, and *DataContext* will discover the new object so that it can be saved to the database. You can also call the *InsertAllOnSubmit* method when you have many objects to insert.

The following sample code demonstrates the creation of an employee object that is then added to the *DataContext* object and submitted to the database.

Sample of Visual Basic Code

```
Private Sub mnuAddNewEntity_Click(ByVal sender As System.Object, _
                                ByVal e As System.Windows.RoutedEventArgs)
    Dim ctx = New NorthwindDataContext()
    Dim sw = New StringWriter()
    ctx.Log = sw
    Dim employee = New Employee With
    {
        .FirstName = "John",
        .LastName = "Smith"
    }
    ctx.Employees.InsertOnSubmit(employee)
    ctx.SubmitChanges()
    MessageBox.Show(sw.GetStringBuilder().ToString())
End Sub
```

Sample of C# Code

```
private void mnuAddNewEntity_Click(object sender, RoutedEventArgs e)
{
    var ctx = new NorthwindDataContext();
    var sw = new StringWriter();
    ctx.Log = sw;
    var employee = new Employee
    {
        FirstName = "John",
```

```

        LastName = "Smith"
    };
    ctx.Employees.InsertOnSubmit(employee);
    ctx.SubmitChanges();
    MessageBox.Show(sw.GetStringBuilder().ToString());
}

```

SQL Query

```

INSERT INTO [dbo].[Employees]([LastName], [FirstName], [Title],
[TitleOfCourtesy], [BirthDate], [HireDate], [Address], [City], [Region],
[PostalCode], [Country], [HomePhone], [Extension], [Photo],
[Notes], [ReportsTo], [PhotoPath])
VALUES (@p0, @p1, @p2, @p3, @p4, @p5, @p6, @p7, @p8,
    @p9, @p10, @p11, @p12, @p13, @p14, @p15, @p16)

SELECT CONVERT(Int,SCOPE_IDENTITY()) AS [value]
-- @p0: Input NVarChar (Size = 4000; Prec = 0; Scale = 0) [Smith]
-- @p1: Input NVarChar (Size = 4000; Prec = 0; Scale = 0) [John]
-- @p2: Input NVarChar (Size = 4000; Prec = 0; Scale = 0) [Null]
-- @p3: Input NVarChar (Size = 4000; Prec = 0; Scale = 0) [Null]
-- @p4: Input DateTime (Size = -1; Prec = 0; Scale = 0) [Null]
-- @p5: Input DateTime (Size = -1; Prec = 0; Scale = 0) [Null]
-- @p6: Input NVarChar (Size = 4000; Prec = 0; Scale = 0) [Null]
-- @p7: Input NVarChar (Size = 4000; Prec = 0; Scale = 0) [Null]
-- @p8: Input NVarChar (Size = 4000; Prec = 0; Scale = 0) [Null]
-- @p9: Input NVarChar (Size = 4000; Prec = 0; Scale = 0) [Null]
-- @p10: Input NVarChar (Size = 4000; Prec = 0; Scale = 0) [Null]
-- @p11: Input NVarChar (Size = 4000; Prec = 0; Scale = 0) [Null]
-- @p12: Input NVarChar (Size = 4000; Prec = 0; Scale = 0) [Null]
-- @p13: Input Image (Size = 8000; Prec = 0; Scale = 0) [Null]
-- @p14: Input NText (Size = -1; Prec = 0; Scale = 0) [Null]
-- @p15: Input Int (Size = -1; Prec = 0; Scale = 0) [Null]
-- @p16: Input NVarChar (Size = 4000; Prec = 0; Scale = 0) [Null]
-- Context: SqlProvider(Sql2008) Model: AttributedMetaModel Build: 4.0.30319.1

```

In this code sample, an employee object was created and populated with data. The employee object is then added to the Employees table on the *DataContext* object by using the *InsertOnSubmit* method. Finally, the *SubmitChanges* method on the *DataContext* object is called to send the new employee to the Employees database table. The SQL query performs the insert and then executes a *SELECT* statement to retrieve the *SCOPE_IDENTITY*, which is the value of the primary key that was created. The EmployeeID column is an identity column that automatically provides a new sequential number for each row added. After the code runs, you can look at the *EmployeeID* property to see the ID that was generated.

Deleting Entities

When you want to delete rows from a database table, you can call the *DeleteOnSubmit* or *DeleteAllOnSubmit* methods on the appropriate table property of the *DataContext* object. Deleting typically requires you to locate the item or items to be deleted and then pass them to the appropriate aforementioned method. The following is a code sample that shows how to delete an employee whose EmployeeID is 10.

Sample of Visual Basic Code

```
Private Sub mnuDeleteEntity_Click(ByVal sender As System.Object, _
                                ByVal e As System.Windows.RoutedEventArgs)
    Dim ctx = New NorthwindDataContext()
    Dim sw = New StringWriter()
    ctx.Log = sw
    Dim employee = (From emp In ctx.Employees
                    Where emp.EmployeeID = 10
                    Select emp).First()
    ctx.Employees.DeleteOnSubmit(employee)
    ctx.SubmitChanges()
    MessageBox.Show(sw.GetStringBuilder().ToString())
End Sub
```

Sample of C# Code

```
private void mnuDeleteEntity_Click(object sender, RoutedEventArgs e)
{
    var ctx = new NorthwindDataContext();
    var sw = new StringWriter();
    ctx.Log = sw;
    var employee = (from emp in ctx.Employees
                    where emp.EmployeeID == 10
                    select emp).First();
    ctx.Employees.DeleteOnSubmit(employee);
    ctx.SubmitChanges();
    MessageBox.Show(sw.GetStringBuilder().ToString());
}
```

SQL Query

```
SELECT TOP (1) [t0].[EmployeeID], [t0].[LastName], [t0].[FirstName],
[t0].[Title], [t0].[TitleOfCourtesy], [t0].[BirthDate], [t0].[HireDate],
[t0].[Address], [t0].[City], [t0].[Region], [t0].[PostalCode], [t0].[Country],
[t0].[HomePhone], [t0].[Extension], [t0].[Notes],
[t0].[ReportsTo], [t0].[PhotoPath]
FROM [dbo].[Employees] AS [t0]
WHERE [t0].[EmployeeID] = @p0
-- @p0: Input Int (Size = -1; Prec = 0; Scale = 0) [10]
-- Context: SqlProvider(Sql2008) Model: AttributedMetaModel Build: 4.0.30319.1

DELETE FROM [dbo].[Employees] WHERE ([EmployeeID] = @p0) AND
([LastName] = @p1) AND
([FirstName] = @p2) AND ([Title] IS NULL) AND
([TitleOfCourtesy] IS NULL) AND ([BirthDate] IS NULL) AND
([HireDate] IS NULL) AND ([Address] IS NULL) AND
([City] IS NULL) AND ([Region] IS NULL) AND
([PostalCode] IS NULL) AND ([Country] IS NULL) AND
([HomePhone] IS NULL) AND ([Extension] IS NULL) AND
([ReportsTo] IS NULL) AND ([PhotoPath] IS NULL)
-- @p0: Input Int (Size = -1; Prec = 0; Scale = 0) [10]
-- @p1: Input NVarChar (Size = 4000; Prec = 0; Scale = 0) [Smith]
-- @p2: Input NVarChar (Size = 4000; Prec = 0; Scale = 0) [John]
-- Context: SqlProvider(Sql2008) Model: AttributedMetaModel Build: 4.0.30319.1
```

This code sample starts by querying for the employee whose EmployeeID is 10. If you had run the previous code sample that adds an employee named John Smith, you would have an employee with an EmployeeID of 10. Next, this code sample executes the *DeleteOnSubmit* method on the Employees *Table* property that's on the *DataContext* object. Finally, the call is made to the *SubmitChanges* method, which sends the *delete* command to the database.

In the SQL query, the row had to be retrieved to be deleted. If you want to delete an employee's name without retrieving it first, implement a stored procedure that takes the ID of the employee as a parameter.

LINQ to SQL does not support or recognize cascade-delete operations. If you want to delete a row in a table that has constraints against it, you have two options. The first option is to set the ON DELETE CASCADE rule in the foreign-key constraint in the database. This automatically deletes the child rows when a parent row is deleted. The other option is to write your own code to first delete the child objects that would otherwise prevent the parent object from being deleted.

Using Stored Procedures

In Lesson 1, you were introduced to the LINQ to SQL designer, and this lesson mentioned that you can add stored procedures to the LINQ to SQL designer. The stored procedures you add to the designer become methods on your *DataContext* object, which makes calling a stored procedure quite easy. The following sample code demonstrates the call to a stored procedure called *CustOrderHist*, which contains a *SELECT* statement that returns a list of all the products a customer has purchased.

Sample of Visual Basic Code

```
Private Sub mnuStoredProc_Click(ByVal sender As System.Object, _
                                ByVal e As System.Windows.RoutedEventArgs)
    Dim ctx = New NorthwindDataContext()
    Dim sw = New StringWriter()
    ctx.Log = sw
    dg.ItemsSource = ctx.CustOrderHist("ALFKI")
    MessageBox.Show(sw.GetStringBuilder().ToString())
End Sub
```

Sample of C# Code

```
private void mnuStoredProc_Click(object sender, RoutedEventArgs e)
{
    var ctx = new NorthwindDataContext();
    var sw = new StringWriter();
    ctx.Log = sw;
    dg.ItemsSource = ctx.CustOrderHist("ALFKI");
    MessageBox.Show(sw.GetStringBuilder().ToString());
}
```

SQL Query

```
EXEC @RETURN_VALUE = [dbo].[CustOrderHist] @CustomerID = @p0
```

```
-- @p0: Input NChar (Size = 5; Prec = 0; Scale = 0) [ALFKI]
-- @RETURN_VALUE: Output Int (Size = -1; Prec = 0; Scale = 0) [Null]
-- Context: SqlProvider(Sql2008) Model: AttributedMetaModel Build: 4.0.30319.1
```

This code is quite simple. You pass the customer ID of a customer (ALFKI in this case), and you get back a list of objects that can be easily bound to the data grid.

If you call stored procedures that have *OUTPUT* parameters, the method created defines parameters as *ByRef* (C# *ref*), which gives you access to the returned value.

Using *DataContext* to Submit Changes

After you use LINQ to SQL to retrieve data, you might make many changes to the objects, but remember that these changes are made only to your in-memory objects. No changes are sent to the database until you call the *SubmitChanges* method on the *DataContext* object.

When you call the *SubmitChanges* method, the *DataContext* object tries to translate your changes into SQL commands. Although you can use your own custom logic to override these actions, the order of submission is orchestrated by a *change processor*.

The first thing the change processor will do is examine the set of known objects to determine whether new objects have been attached. If so, these new objects are added to the set of tracked objects.

Next, all objects that have pending changes are ordered into a sequence based on the dependencies between the objects. Objects whose changes depend on other objects will be located after their dependencies.

The *DataContext* object then starts a transaction to encapsulate the individual *insert*, *update*, and *delete* commands.

Finally, the changes to the objects are sent to the database server, one by one, as SQL commands. Any errors detected by the database cause the submission process to stop, and an exception is thrown. The transaction rolls back all changes to the database. *DataContext* still has all changes, so you can try to correct the problem and call *SubmitChanges* again if need be.

Following successful execution of *SubmitChanges*, all objects known to the *DataContext* object are in the *Unchanged* state. (The single exception is represented by those that have been successfully deleted from the database, which are in *Deleted* state and unusable in that *DataContext* instance.)

Submitting Changes in a Transaction

LINQ to SQL supports three transaction models when submitting your changes back to the database. This section covers these models in the order of checks performed.

- **Explicit Local Transaction** If the *Transaction* property is set to a (*IdbTransaction*) transaction, the *SubmitChanges* call is executed in the context of the same transaction. In this scenario, you are responsible for committing or rolling back the transaction

after execution of the transaction. Also, the connection that created the transaction must be the same connection used when instantiating *DataContext*. This throws an exception if a different connection is used.

- **Explicit Distributable Transaction** In this scenario, you can call any of the LINQ to SQL methods, including, but not limited to, the *SubmitChanges* method, in the scope of an active transaction. LINQ to SQL will detect that the call is in the scope of a transaction and will not create a new transaction. Also, the connection will not be closed. For example, you can perform queries and *SubmitChanges* executions in the context of this transaction.
- **Implicit Transaction** When you call *SubmitChanges*, LINQ to SQL automatically checks to see whether the call is already in the scope of a transaction or if the *Transaction* property (*IdbTransaction*) is set to a transaction that you started. If no transaction is found, LINQ to SQL starts a local transaction (*IdbTransaction*) and uses it to execute the generated SQL commands. When all SQL commands have been successfully completed, LINQ to SQL commits the local transaction and returns.

Writing LINQ Queries to Display Data

In this practice, you continue the order entry application from Lesson 2, “Executing Queries Using LINQ to SQL,” by adding a graphical user interface (GUI) for order entry and then using LINQ to SQL to add the new order to the database.

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

EXERCISE 1 Add Order Entry to the Application

In this exercise, you modify the WPF application that you created in Lesson 2 by creating the GUI for order entry and then adding code to store the new order into the database by using LINQ to SQL.

1. In Visual Studio .NET 2010, choose File | Open | Project. Open the project from Lesson 2.
2. In Solution Explorer, right-click *OrderEntryProject*, choose Add | Window, and enter **OrderWindow.xaml** as the name of the new window.
3. In the XAML window, enter the following markup to create the window for entering the order information. Your XAML should look like the following:

Sample of Visual Basic XAML

```
<Window x:Class="OrderWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="OrderWindow" SizeToContent="WidthAndHeight">
    <Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="Auto" />
```

```

        <ColumnDefinition Width="Auto" />
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto" />
        <RowDefinition Height="Auto" />
        <RowDefinition Height="Auto" />
    </Grid.RowDefinitions>
    <DatePicker Grid.Column="1" Margin="10" Name="dtOrder" />
    <DatePicker Margin="10" Name="dtRequired" Grid.Column="1" Grid.Row="1" />
    <Button Content="Cancel" Grid.Row="2" Margin="10"
        Name="btnCancel" Click="btnCancel_Click" />
    <Button Content="OK" Margin="10" Name="btnOk"
        Grid.Column="1" Grid.Row="2" Click="btnOk_Click" />
    <TextBlock Margin="10" Text="Order Date:" />
    <TextBlock Margin="10" Text="Required Date:" Grid.Row="1" />
</Grid>
</Window>

```

Sample of C# XAML

```

<Window x:Class="OrderEntryProject.OrderWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="OrderWindow" SizeToContent="WidthAndHeight">
    <Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="Auto" />
            <ColumnDefinition Width="Auto" />
        </Grid.ColumnDefinitions>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
        </Grid.RowDefinitions>
        <DatePicker Grid.Column="1" Margin="10" Name="dtOrder" />
        <DatePicker Margin="10" Name="dtRequired" Grid.Column="1" Grid.Row="1" />
        <Button Content="Cancel" Grid.Row="2" Margin="10"
            Name="btnCancel" Click="btnCancel_Click" />
        <Button Content="OK" Margin="10" Name="btnOk"
            Grid.Column="1" Grid.Row="2" Click="btnOk_Click" />
        <TextBlock Margin="10" Text="Order Date:" />
        <TextBlock Margin="10" Text="Required Date:" Grid.Row="1" />
    </Grid>
</Window>

```

4. Right-click the OrderWindow.xaml file and click View Code to go to the code-behind window. Add a public *string* property called *CustomerID* to the top of the *OrderWindow* class. Your code should look like the following:

Sample of Visual Basic Code

```
Public Property CustomerID As String
```

Sample of C# Code

```
public string CustomerID { get; set; }
```

5. Double-click *btnCancel* to create the *btnCancel_Click* event handler method automatically and add a single line of code to call the *Close* method so this window will be closed.
6. Double-click *btnOk* to create the *btnOk_Click* event handler method automatically and add code to create a new *Order* object with the values from the main window. Use *DataContext* to go to the *Orders* collection and call *InsertOnSubmit* with the new order. Your completed window code should look like the following:

Sample of Visual Basic Code

Public Class OrderWindow

Public Property CustomerID As String

```
Private Sub btnOk_Click(ByVal sender As System.Object, _
                        ByVal e As System.Windows.RoutedEventArgs)
    Dim ctx = CType(Application.Current.Properties("ctx"), NorthwindDataContext)
    Dim Order = New Order With _
    { _
        .CustomerID = CustomerID, _
        .OrderDate = dtOrder.SelectedDate, _
        .RequiredDate = dtRequired.SelectedDate _
    }
    ctx.Orders.InsertOnSubmit(Order)
    Close()
End Sub
```

```
Private Sub btnCancel_Click(ByVal sender As System.Object, _
                            ByVal e As System.Windows.RoutedEventArgs)
    Close()
End Sub
End Class
```

Sample of C# Code

using System.Windows;

```
namespace OrderEntryProject
{
    /// <summary>
    /// Interaction logic for OrderWindow.xaml
    /// </summary>
    public partial class OrderWindow : Window
    {
        public string CustomerID { get; set; }

        public OrderWindow()
        {
            InitializeComponent();
        }

        private void btnCancel_Click(object sender, RoutedEventArgs e)
        {
            Close();
        }
    }
}
```

```

    }

    private void btnOk_Click(object sender, RoutedEventArgs e)
    {
        var ctx = (NorthwindDataContext) App.Current.Properties["ctx"];
        Order order = new Order
        {
            CustomerID = this.CustomerID,
            OrderDate = dtOrder.SelectedDate,
            RequiredDate = dtRequired.SelectedDate
        };
        ctx.Orders.InsertOnSubmit(order);
        this.Close();
    }
}

```

7. Open the code-behind screen for *MainWindow* and add code to the *Save* event handler method to submit the *DataContext* changes to the database and display a message stating "Saved".
8. Call the customer combo box's *SelectionChanged* event handler method to force the *ListBox* to be updated. Your code should look like the following:

Sample of Visual Basic Code

```

Private Sub mnuSave_Click(ByVal sender As System.Object, _
                        ByVal e As System.Windows.RoutedEventArgs)
    ctx.SubmitChanges()
    MessageBox.Show("Saved")
    cmbCustomers_SelectionChanged(Nothing, Nothing)
End Sub

Private Sub mnuOrder_Click(ByVal sender As System.Object, _
                        ByVal e As System.Windows.RoutedEventArgs)
    Dim customer = CType(cmbCustomers.SelectedValue, Tuple(Of String, String))
    Dim window = New OrderWindow With {.CustomerID = customer.Item1}
    window.ShowDialog()
End Sub

```

Sample of C# Code

```

private void mnuSave_Click(object sender, RoutedEventArgs e)
{
    ctx.SubmitChanges();
    MessageBox.Show("Saved");
    cmbCustomers_SelectionChanged(null, null);
}

private void mnuOrder_Click(object sender, RoutedEventArgs e)
{
    var customer = (Tuple<string,string>)cmbCustomers.SelectedValue;
    OrderWindow window = new OrderWindow {CustomerID = customer.Item1};
    window.ShowDialog();
}

```

9. Choose Debug | Start Debugging to run your application.
10. Select a customer from the combo box. You should see a list of orders for that customer.
11. With a customer selected, click New Order on the menu. Enter an order date and required date and click OK.
12. Click Save. After you click OK to close the Saved window, you should see the updated list of orders, which includes your new order.

Lesson Summary

This lesson provided detailed information on how to send changes back to the database using LINQ to SQL.

- *DataContext* contains the identity tracking service that tracks objects created by *DataContext*.
- *DataContext* contains the change tracking service that keeps track of the state of your object to facilitate sending changes back to the database.
- The change tracking service will keep the original values of your objects, which enables *DataContext* to identify concurrency errors, such as when someone else modifies the same data with which you are working.
- You call the *SubmitChanges* method on the *DataContext* object to send all changes back to the database.
- To make *DataContext* aware of your object, call the *InsertOnSubmit* method on the appropriate table property of *DataContext* with your newly constructed object.
- To delete data from the database, call the *DeleteOnSubmit* method on the appropriate table property of *DataContext* with the entity object you want to delete.
- Changes that are sent back to the database are in an implicit transaction, but you can also create an explicit transaction for when you need other actions to be part of the same transaction.

Lesson Review

You can use the following questions to test your knowledge of the information in Lesson 2, "Executing Queries Using LINQ to SQL." 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. You retrieved a row of data into an entity object by using a LINQ to SQL *DataContext* object. You haven't made any changes to the object, but you know that someone else has modified the data row in the database table, so you rerun your query, using the same LINQ to SQL *DataContext* object, to retrieve the updated data. What can be said about the result of the second query?
 - A. It returns the updated data and you can use it immediately.
 - B. The changes are thrown out and you use the cached data, so you don't see the changes.
 - C. An exception is thrown due to the difference in data.
 - D. An exception is thrown because you already have the object, so you can't re-query unless you create a new *DataContext* object.
2. You ran a LINQ to SQL query to retrieve the products that you are going to mark as discontinued. After running the query, you looped through the returned products and set their *Discontinued* property to *true*. What must you do to ensure that the changes go back to the database?
 - A. Call the *Update* method on the *DataContext* object.
 - B. Nothing; the changes are sent when you modify the object.
 - C. Call the *Dispose* method on the *DataContext* object.
 - D. Call the *SubmitChanges* method on the *DataContext* object.

Case Scenario

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

Case Scenario: Object-Oriented Data Access

Your boss has given you the task of designing a new software application. He is not a developer, but he read an article that described the benefits of object-oriented applications, so the only constraint that was conveyed to you was that the application must be an object-oriented application. The application will require you to store and modify data that will reside in a SQL Server database that you will also design. Answer the following questions regarding the implementation of the data access layer you will be creating.

1. If you chose to use traditional ADO.NET classes, such as *DataSet* and *SqlDataAdapter*, will that satisfy the requirement to be an object-oriented application? Does the use of LINQ to SQL satisfy the object-oriented requirement?
2. Describe some differences between these two technologies.

Suggested Practices

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

Create an Application That Uses LINQ to SQL Queries

You should create at least one application that uses the LINQ to SQL queries. This can be accomplished by completing the practices at the end of Lessons 1 and 2 or by completing the following Practice 1.

- **Practice 1** Create an application that requires you to query a database for data from at least two tables that are related. This could be movies that have actors, artists who record music, or people who have vehicles. Add LINQ to SQL queries to provide searching and filtering of the data.
- **Practice 2** Complete Practice 1 and then add LINQ to SQL queries that join the tables. Be sure to provide both inner and outer joins. Also, add queries that perform grouping and aggregates.

Create an Application That Modifies Data by Using LINQ to SQL

You should create at least one application that uses LINQ to SQL to modify data in the database. This can be accomplished by performing the practices at the end of Lessons 1, 2, and 3 or by completing the following Practice 1.

- **Practice 1** Create an application that requires you to collect data into at least two database tables that are related. This could be movies that have actors, artists who record music, or people who have vehicles. Use LINQ to SQL to add, delete, and modify the data.

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.

CHAPTER 5

LINQ to XML

XML has been a rapidly growing technology because it provides a verbose means for transferring data that can be understood easily by computers as well as by people. You will often need to query the XML data.

Another common requirement is to transform XML into a different format. In some scenarios, you simply want to convert XML to a different form of XML. In other scenarios, you might want to convert XML into HTML. You might even want to convert XML into text.

This chapter's first lesson shows how you can use *XmlDocument* and *XmlReader* classes to query XML data. Lesson 2, "Querying with LINQ to XML," shows how you can use LINQ to XML to retrieve data from XML. Lesson 3, "Transforming XML Using LINQ to XML," uses LINQ to XML to transform XML data.

Exam objectives in this chapter:

- Query XML.

Lessons in this chapter:

- Lesson 1: Working with the *XmlDocument* and *XmlReader* Classes **297**
- Lesson 2: Querying with LINQ to XML **320**
- Lesson 3 Transforming XML Using LINQ to XML **344**

Before You Begin

You must have some understanding of Microsoft C# or Visual Basic 2010, and you should be familiar with XPath query language, also known as XML Path Language. XPath is an XML technology that was created to provide a common syntax and semantics to address parts of an XML document. XPath has been a W3C (World Wide Web Consortium, <http://www.w3c.org>) recommendation since November 1999.

XPath uses a path notation for navigating through the hierarchical structure of an XML document that is similar to that used for navigating the folder hierarchy on your disk drive when you locate a file. Just as you can locate a file by specifying a relative or explicit path, you can locate parts of an XML document by supplying a relative or explicit path. Even the asterisk (*) is useful as the “all” wildcard when locating parts of an XML document. This chapter exposes you to simple XPath queries, but XPath is not the chapter’s focus.

This chapter requires only the hardware and software listed at the beginning of this book.



REAL WORLD

Glenn Johnson

I have worked on many projects that required the use of XML to transfer data to and from remote systems and web services. In the past, when an XML response was received, I would parse (shred) the XML using various XPath queries to retrieve the necessary data. With LINQ to XML, I can shred the XML by using LINQ queries to retrieve the necessary data.

Lesson 1: Working with the *XmlDocument* and *XmlReader* Classes

The *XmlDocument* and *XmlReader* classes have existed since Microsoft .NET Framework 1.0. This lesson explores each of these classes, showing benefits and drawbacks of using each in your code.

After this lesson, you will be able to:

- Create an *XmlDocument* object.
- Iterate the nodes in an *XmlDocument* object.
- Search for nodes in an *XmlDocument* object.
- Implement the *XmlReader* class.

Estimated lesson time: 45 minutes

The *XmlDocument* Class

The W3C has provided standards that define the structure and a standard programming interface called the Document Object Model (DOM) that can be used in a wide variety of environments and applications for XML documents. Classes that support the DOM typically are capable of random access navigation and modification of the XML document.

The XML classes are accessible by setting a reference to the System.Xml.dll file and adding the *Imports System.Xml* (C# using *System.Xml*;) directive to the code.

The *XmlDocument* class is an in-memory representation of XML using the DOM Level 1 and Level 2. This class can be used to navigate and edit the XML nodes.

There is another class, *XmlDataDocument*, which inherits from the *XmlDocument* class and represents relational data. The *XmlDataDocument* class, in the System.Data.dll assembly, can expose its data as a data set to provide relational and nonrelational views of the data. This lesson focuses on the *XmlDocument* class.

These classes provide many methods to implement the Level 2 specification and contain methods to facilitate common operations. The methods are summarized in Table 5-1. The *XmlDocument* class, which inherits from *XmlNode*, contains all the methods for creating XML elements and XML attributes.

TABLE 5-1 Summary of the *XmlDocument* Methods

METHOD	DESCRIPTION
<i>CreateNode</i>	Creates an XML node in the document. There are also specialized <i>Create</i> methods for each node type such as <i>CreateElement</i> or <i>CreateAttribute</i> .
<i>CloneNode</i>	Creates a duplicate of an XML node. This method takes a Boolean argument called <i>deep</i> . If <i>deep</i> is <i>false</i> , only the node is copied; if <i>deep</i> is <i>true</i> , all child nodes are recursively copied as well.
<i>GetElementById</i>	Locates and returns a single node based on its <i>ID</i> attribute. This requires a document type definition (DTD) that identifies an attribute as being an <i>ID</i> type. An attribute with the name <i>ID</i> is not an <i>ID</i> type by default.
<i>GetElementsByTagName</i>	Locates and returns an <i>XmlNode</i> list containing all the descendant elements based on the element name.
<i>ImportNode</i>	Imports a node from a different <i>XmlDocument</i> class into the current document. The source node remains unmodified in the original <i>XmlDocument</i> class. This method takes a Boolean argument called <i>deep</i> . If <i>deep</i> is <i>false</i> , only the node is copied; if <i>deep</i> is <i>true</i> , all child nodes are recursively copied as well.
<i>InsertBefore</i>	Inserts an <i>XmlNode</i> list immediately before the referenced node. If the referenced node is <i>Nothing</i> (or <i>null</i> in C#), the new node is inserted at the end of the child list. If the node already exists in the tree, the original node is removed when the new node is inserted.
<i>InsertAfter</i>	Inserts an <i>XmlNode</i> list immediately after the referenced node. If the referenced node is <i>Nothing</i> (or <i>null</i> in C#), the new node is inserted at the beginning of the child list. If the node already exists in the tree, the original node is removed when the new node is inserted.
<i>Load</i>	Loads an XML document from a disk file, Uniform Resource Locator (URL), or stream.
<i>LoadXml</i>	Loads an XML document from a string.
<i>Normalize</i>	Ensures that there are no adjacent text nodes in the document. This is like saving the document and reloading it. This method can be desirable when text nodes are being programmatically added to an <i>XmlDocument</i> class, and the text nodes could be side by side. Normalizing combines the adjacent text nodes to produce a single text node.

<i>PrependChild</i>	Inserts a node at the beginning of the child node list. If the new node is already in the tree, it is removed before it is inserted. If the node is an <i>XmlDocument</i> fragment, the complete fragment is added.
<i>ReadNode</i>	Loads a node from an XML document by using an <i>XmlReader</i> object. The reader must be on a valid node before executing this method. The reader reads the opening tag, all child nodes, and the closing tag of the current element. This repositions the reader to the next node.
<i>RemoveAll</i>	Removes all children and attributes from the current node.
<i>RemoveChild</i>	Removes the referenced child.
<i>ReplaceChild</i>	Replaces the referenced child with a new node. If the new node is already in the tree, it is removed before it is inserted.
<i>Save</i>	Saves the XML document to a disk file, URL, or stream.
<i>SelectNodes</i>	Selects a list of nodes that match the XPath expression.
<i>SelectSingleNode</i>	Selects the first node that matches the XPath expression.
<i>WriteTo</i>	Writes a node to another XML document using an <i>XmlTextWriter</i> class.
<i>WriteContentsTo</i>	Writes a node and all its descendants to another XML document using an <i>XmlTextWriter</i> class.

Creating the *XmlDocument* Object

To create an *XmlDocument* object, start by instantiating an *XmlDocument* class. The *XmlDocument* object contains *CreateElement* and *CreateAttribute* methods that add nodes to the *XmlDocument* object. The *XmlElement* contains the *Attributes* property, which is an *XmlAttributeCollection* type. The *XmlAttributeCollection* type inherits from the *XmlNamedNodeMap* class, which is a collection of names with corresponding values.

The following code shows how an *XmlDocument* class can be created from the beginning and saved to a file. Note that *import System.Xml* (C# *using System.Xml;*) and *Import System.IO* (C# *using System.IO;*) was added to the top of the code file.

Sample of Visual Basic Code

```
Private Sub CreateAndSaveXmlDocumentToolStripMenuItem_Click( _
    ByVal sender As System.Object, ByVal e As System.EventArgs) _
    Handles CreateAndSaveXmlDocumentToolStripMenuItem.Click
    'Declare and create new XmlDocument
    Dim xmlDoc As New XmlDocument()

    Dim el As XmlElement
    Dim childCounter As Integer
    Dim grandChildCounter As Integer
```

```

'Create the xml declaration first
xmlDoc.AppendChild( _
    xmlDoc.CreateXmlDeclaration("1.0", "utf-8", Nothing))

'Create the root node and append into doc
e1 = xmlDoc.CreateElement("MyRoot")
xmlDoc.AppendChild(e1)

'Child Loop
For childCounter = 1 To 4
    Dim childElmt As XmlElement
    Dim childAttr As XmlAttribute

    'Create child with ID attribute
    childElmt = xmlDoc.CreateElement("MyChild")
    childAttr = xmlDoc.CreateAttribute("ID")
    childAttr.Value = childCounter.ToString()
    childElmt.Attributes.Append(childAttr)

    'Append element into the root element
    e1.AppendChild(childElmt)
    For grandChildCounter = 1 To 3
        'Create grandchildren
        childElmt.AppendChild(xmlDoc.CreateElement("MyGrandChild"))
    Next
Next

'Save to file
xmlDoc.Save(GetFilePath("XmlDocumentTest.xml"))
txtLog.AppendText("XmlDocumentTest.xml Created" + vbCrLf)

End Sub

Private Function getFilePath(ByVal fileName As String) As String
    Return Path.Combine(Environment.GetFolderPath( _
        Environment.SpecialFolder.Desktop), fileName)
End Function

```

Sample of C# Code

```

private void createAndSaveXmlDocumentToolStripMenuItem_Click(
    object sender, EventArgs e)
{
    //Declare and create new XmlDocument
    var xmlDoc = new XmlDocument();

    XmlElement e1;
    int childCounter;
    int grandChildCounter;

    //Create the xml declaration first
    xmlDoc.AppendChild(
        xmlDoc.CreateXmlDeclaration("1.0", "utf-8", null));

    //Create the root node and append into doc
    e1 = xmlDoc.CreateElement("MyRoot");
}

```

```

xmlDoc.AppendChild(e1);

//Child Loop
for (childCounter = 1; childCounter <= 4; childCounter++)
{
    XmlElement childElmt;
    XmlAttribute childAttr;

    //Create child with ID attribute
    childElmt = xmlDoc.CreateElement("MyChild");
    childAttr = xmlDoc.CreateAttribute("ID");
    childAttr.Value = childCounter.ToString();
    childElmt.Attributes.Append(childAttr);

    //Append element into the root element
    e1.AppendChild(childElmt);
    for (grandChildCounter = 1; grandChildCounter <= 3;
        grandChildCounter++)
    {
        //Create grandchildren
        childElmt.AppendChild(xmlDoc.CreateElement("MyGrandChild"));
    }
}

//Save to file
xmlDoc.Save(getFilePath("XmlDocumentTest.xml"));
txtLog.AppendText("XmlDocumentTest.xml Created\r\n");
}

private string getFilePath(string fileName)
{
    return Path.Combine(Environment.GetFolderPath(
        Environment.SpecialFolder.Desktop), fileName);
}

```

This code started by creating an instance of *XmlDocument*. Next, the XML declaration is created and placed inside the child collection. An exception is thrown if this is not the first child of *XmlDocument*. After that, the root element is created and the child nodes with corresponding attributes are created. Finally, a call is made to the *getFilePath* helper method to assemble a file path to save the file to your desktop. This helper method will be used in subsequent code samples. The following is the XML file that was produced by running the code sample:

XML File

```

<?xml version="1.0" encoding="utf-8"?>
<MyRoot>
  <MyChild ID="1">
    <MyGrandChild />
    <MyGrandChild />
    <MyGrandChild />
  </MyChild>
  <MyChild ID="2">

```

```

        <MyGrandChild />
        <MyGrandChild />
        <MyGrandChild />
    </MyChild>
    <MyChild ID="3">
        <MyGrandChild />
        <MyGrandChild />
        <MyGrandChild />
    </MyChild>
    <MyChild ID="4">
        <MyGrandChild />
        <MyGrandChild />
        <MyGrandChild />
    </MyChild>
</MyRoot>

```

Parsing an *XmlDocument* Object by Using the DOM

An *XmlDocument* object can be parsed by using a recursive routine to loop through all elements. The following code has an example of parsing *XmlDocument*. Note that *imports System.Text* (C# *using System.Text*;) was added.

Sample of Visual Basic Code

```

Private Sub ParsingAnXmlDocumentToolStripMenuItem_Click( _
    ByVal sender As System.Object, ByVal e As System.EventArgs) _
    Handles ParsingAnXmlDocumentToolStripMenuItem.Click
    Dim xmlDoc As New XmlDocument()
    xmlDoc.Load(getFilePath("XmlDocumentTest.xml"))
    RecurseNodes(xmlDoc.DocumentElement)
End Sub

Public Sub RecurseNodes(ByVal node As XmlNode)
    Dim sb As New StringBuilder()
    'start recursive loop with level 0
    RecurseNodes(node, 0, sb)
    txtLog.Text = sb.ToString()
End Sub

Public Sub RecurseNodes( _
    ByVal node As XmlNode, ByVal level As Integer, _
    ByVal sb As StringBuilder)
    sb.AppendFormat("{0,2} Type:{1,-9} Name:{2,-13} Attr:", _
        level, node.NodeType, node.Name)

    For Each attr As XmlAttribute In node.Attributes
        sb.AppendFormat("{0}={1} ", attr.Name, attr.Value)
    Next
    sb.AppendLine()

    For Each n As XmlNode In node.ChildNodes
        RecurseNodes(n, level + 1, sb)
    Next
End Sub

```


Sample of C# Code

```
private void parsingAndXmlDocumentToolStripMenuItem_Click(object sender, EventArgs e)
{
    XmlDocument xmlDoc = new XmlDocument();
    xmlDoc.Load(getFilePath("XmlDocumentTest.xml"));
    RecurseNodes(xmlDoc.DocumentElement);
}

public void RecurseNodes(XmlNode node)
{
    var sb = new StringBuilder();
    //start recursive loop with level 0
    RecurseNodes(node, 0, sb);
    txtLog.Text = sb.ToString();
}

public void RecurseNodes(XmlNode node, int level, StringBuilder sb)
{
    sb.AppendFormat("{0,2} Type:{1,-9} Name:{2,-13} Attr:",
        level, node.NodeType, node.Name);

    foreach (XmlAttribute attr in node.Attributes)
    {
        sb.AppendFormat("{0}={1} ", attr.Name, attr.Value);
    }
    sb.AppendLine();

    foreach (XmlNode n in node.ChildNodes)
    {
        RecurseNodes(n, level + 1, sb);
    }
}
```

This code starts by loading an XML file and then calling the *RecurseNodes* method, which is overloaded. The first call simply passes the *xmlDoc* root node. The recursive call passes the recursion level and a string builder object. Each time the *RecurseNodes* method executes, the node information is printed, and a recursive call is made for each child the node has. The following is the result.

Parsing Result

0	Type:Element	Name:MyRoot	Attr:
1	Type:Element	Name:MyChild	Attr:ID=1
2	Type:Element	Name:MyGrandChild	Attr:
2	Type:Element	Name:MyGrandChild	Attr:
2	Type:Element	Name:MyGrandChild	Attr:
1	Type:Element	Name:MyChild	Attr:ID=2
2	Type:Element	Name:MyGrandChild	Attr:
2	Type:Element	Name:MyGrandChild	Attr:
2	Type:Element	Name:MyGrandChild	Attr:
1	Type:Element	Name:MyChild	Attr:ID=3
2	Type:Element	Name:MyGrandChild	Attr:
2	Type:Element	Name:MyGrandChild	Attr:
2	Type:Element	Name:MyGrandChild	Attr:
1	Type:Element	Name:MyChild	Attr:ID=4

```

2 Type:Element      Name:MyGrandChild  Attr:
2 Type:Element      Name:MyGrandChild  Attr:
2 Type:Element      Name:MyGrandChild  Attr:

```

Searching the *XmlDocument* Object

The *SelectSingleNode* method can locate an element; it requires an XPath query to be passed into the method. The following code sample calls the *SelectSingleNode* method to locate the *MyChild* element, the ID of which is 3, by using an XPath query. The sample code is as follows:

Sample of Visual Basic Code

```

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

    Dim xmlDoc As New XmlDocument()
    xmlDoc.Load(getFilePath("XmlDocumentTest.xml"))

    Dim node = xmlDoc.SelectSingleNode("//MyChild[@ID='3']")
    RecurseNodes(node)
End Sub

```

Sample of C# Code

```

private void searchingAnXmlDocumentToolStripMenuItem_Click(
    object sender, EventArgs e)
{
    var xmlDoc = new XmlDocument();
    xmlDoc.Load(getFilePath("XmlDocumentTest.xml"));

    var node = xmlDoc.SelectSingleNode("//MyChild[@ID='3']");
    RecurseNodes(node);
}

```

The *SelectSingleNode* method can perform an XPath lookup on any element or attribute. The following is a display of the result.

Search Result

```

0 Type:Element      Name:MyChild      Attr:ID=3
1 Type:Element      Name:MyGrandChild  Attr:
1 Type:Element      Name:MyGrandChild  Attr:
1 Type:Element      Name:MyGrandChild  Attr:

```

The *GetElementsByTagName* method returns an *XmlNode* list containing all matched elements. The following code returns a list of nodes with the tag name *MyGrandChild*.

Sample of Visual Basic Code

```

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

    Dim xmlDoc As New XmlDocument()
    xmlDoc.Load(getFilePath("XmlDocumentTest.xml"))

```

```

Dim elmts = xmlDoc.GetElementsByTagName("MyGrandChild")

Dim sb As New StringBuilder()
For Each node As XmlNode In elmts
    RecurseNodes(node, 0, sb)
Next
txtLog.Text = sb.ToString()
End Sub

```

Sample of C# Code

```

private void getElementsByTagNameToolStripMenuItem_Click(
    object sender, EventArgs e)
{
    var xmlDoc = new XmlDocument();
    xmlDoc.Load(getFilePath("XmlDocumentTest.xml"));

    var elmts = xmlDoc.GetElementsByTagName("MyGrandChild");

    var sb = new StringBuilder();
    foreach (XmlNode node in elmts)
    {
        RecurseNodes(node, 0, sb);
    }
    txtLog.Text = sb.ToString();
}

```

This method works well, even for a single node lookup, when searching by tag name. The following is the execution result.

Search Result

```

0 Type:Element   Name:MyGrandChild Attr:
0 Type:Element   Name:MyGrandChild Attr:
0 Type:Element   Name:MyGrandChild Attr:
0 Type:Element   Name:MyGrandChild Attr:
0 Type:Element   Name:MyGrandChild Attr:
0 Type:Element   Name:MyGrandChild Attr:
0 Type:Element   Name:MyGrandChild Attr:
0 Type:Element   Name:MyGrandChild Attr:
0 Type:Element   Name:MyGrandChild Attr:
0 Type:Element   Name:MyGrandChild Attr:
0 Type:Element   Name:MyGrandChild Attr:
0 Type:Element   Name:MyGrandChild Attr:

```

The *SelectNodes* method, which requires an XPath query to be passed into the method, can also retrieve an *XmlNode* list. The previous code sample has been modified to call the *SelectNodes* method to achieve the same result, as shown in the following code:

Sample of Visual Basic Code

```

Private Sub SelectNodesToolStripMenuItem_Click( _
    ByVal sender As System.Object, ByVal e As System.EventArgs) _
    Handles SelectNodesToolStripMenuItem.Click
    Dim xmlDoc As New XmlDocument()
    xmlDoc.Load(getFilePath("XmlDocumentTest.xml"))

```

```

Dim elmts = xmlDoc.SelectNodes("//MyGrandChild")

Dim sb As New StringBuilder()
For Each node As XmlNode In elmts
    RecurseNodes(node, 0, sb)
Next
txtLog.Text = sb.ToString()
End Sub

```

Sample of C# Code

```

private void selectNodesToolStripMenuItem_Click(
    object sender, EventArgs e)
{
    var xmlDoc = new XmlDocument();
    xmlDoc.Load(getFilePath("XmlDocumentTest.xml"));

    var elmts = xmlDoc.SelectNodes("//MyGrandChild");

    var sb = new StringBuilder();
    foreach (XmlNode node in elmts)
    {
        RecurseNodes(node, 0, sb);
    }
    txtLog.Text = sb.ToString();
}

```

This method can perform an XPath lookup on any XML node, including elements, attributes, and text nodes. This provides much more querying flexibility, because the *SelectElementsByTagName* node is limited to a tag name.

The *XmlReader* Class

The *XmlReader* class is an abstract base class that provides methods to read and parse XML. One of the more common child classes of the *XmlReader* is *XmlTextReader*, which reads an XML file node by node.

The *XmlReader* class provides the fastest and least memory-consuming means to read and parse XML data by providing forward-only, noncaching access to an XML data stream. This class is ideal when it's possible that the desired information is near the top of the XML file and the file is large. If random access is required when accessing XML, use the *XmlDocument* class. The following code reads the XML file that was created in the previous example and displays information about each node:

Sample of Visual Basic Code

```

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

    Dim sb As New StringBuilder()
    Dim xmlReader As New _
        XmlTextReader(getFilePath("XmlDocumentTest.xml"))

```

```

Do While xmlReader.Read()
    Select Case xmlReader.NodeType
        Case XmlNodeType.XmlDeclaration, _
            XmlNodeType.Element, _
            XmlNodeType.Comment
            Dim s As String
            sb.AppendFormat("{0}: {1} = {2}", _
                xmlReader.NodeType, _
                xmlReader.Name, _
                xmlReader.Value)
            sb.AppendLine()
        Case XmlNodeType.Text
            Dim s As String
            sb.AppendFormat(" - Value: {0}", _
                xmlReader.Value)
            sb.AppendLine()
    End Select

    If xmlReader.HasAttributes Then
        Do While xmlReader.MoveToNextAttribute()
            sb.AppendFormat(" - Attribute: {0} = {1}", _
                xmlReader.Name, xmlReader.Value)
            sb.AppendLine()
        Loop
    End If
Loop
xmlReader.Close()
txtLog.Text = sb.ToString()
End Sub

```

Sample of C# Code

```

private void parsingWithXmlReaderToolStripMenuItem_Click(object sender, EventArgs e)
{
    var sb = new StringBuilder();
    var xmlReader = new XmlTextReader(getFilePath("XmlDocumentTest.xml"));

    while (xmlReader.Read())
    {
        switch (xmlReader.NodeType)
        {
            case XmlNodeType.XmlDeclaration:
            case XmlNodeType.Element:
            case XmlNodeType.Comment:
                sb.AppendFormat("{0}: {1} = {2}",
                    xmlReader.NodeType,
                    xmlReader.Name,
                    xmlReader.Value);

                sb.AppendLine();
                break;
            case XmlNodeType.Text:
                sb.AppendFormat(" - Value: {0}", xmlReader.Value);
                sb.AppendLine();
                break;
        }
    }
}

```

```

        if (xmlReader.HasAttributes)
        {
            while (xmlReader.MoveToNextAttribute())
            {
                sb.AppendFormat(" - Attribute: {0} = {1}",
                                xmlReader.Name,
                                xmlReader.Value);
                sb.AppendLine();
            }
        }
    }
    xmlReader.Close();
    txtLog.Text = sb.ToString();
}

```

This code opens the file and then performs a simple loop, reading one element at a time until finished. For each node read, a check is made on *NodeType*, and the node information is printed. When a node is read, its corresponding attributes are read as well. A check is made to see whether the node has attributes, and they are displayed. The following is the result of the sample code execution.

Parse Result

```

XmlDeclaration: xml = version="1.0" encoding="utf-8"
  - Attribute: version = 1.0
  - Attribute: encoding = utf-8
Element: MyRoot =
Element: MyChild =
  - Attribute: ID = 1
Element: MyGrandChild =
Element: MyGrandChild =
Element: MyGrandChild =
Element: MyChild =
  - Attribute: ID = 2
Element: MyGrandChild =
Element: MyGrandChild =
Element: MyGrandChild =
Element: MyChild =
  - Attribute: ID = 3
Element: MyGrandChild =
Element: MyGrandChild =
Element: MyGrandChild =
Element: MyChild =
  - Attribute: ID = 4
Element: MyGrandChild =
Element: MyGrandChild =
Element: MyGrandChild =

```

When viewing the results, notice that many lines end with an equals sign because none of the nodes contained text. The *MyChild* elements have attributes that are displayed.



EXAM TIP

For the exam, understand that the *XmlReader* provides the fastest means to access XML data and is read-only, forward-only. *XmlDocument* provides the simplest means to access XML data and provides random access to any XML node.

Work with the *XmlDocument* and *XmlReader* Classes

In this practice, you analyze an XML file, called *Orders.xml*, which contains order information. Your first objective is to write a program that can provide the total price of all orders. You also need to provide the total and the average freight cost, per order. Here is an example of what the file looks like.

Orders.xml File

```
<Orders>
  <Order OrderNumber="S043659">
    <LineItem Line="1" PID="349" Qty="1" Price="2024.9940" Freight="50.6249" />
    <LineItem Line="2" PID="350" Qty="3" Price="2024.9940" Freight="151.8746" />
    <LineItem Line="3" PID="351" Qty="1" Price="2024.9940" Freight="50.6249" />
    <LineItem Line="4" PID="344" Qty="1" Price="2039.9940" Freight="50.9999" />
    <LineItem Line="5" PID="345" Qty="1" Price="2039.9940" Freight="50.9999" />
    <LineItem Line="6" PID="346" Qty="2" Price="2039.9940" Freight="101.9997" />
    <LineItem Line="7" PID="347" Qty="1" Price="2039.9940" Freight="50.9999" />
    <LineItem Line="8" PID="229" Qty="3" Price="28.8404" Freight="2.1630" />
    <LineItem Line="9" PID="235" Qty="1" Price="28.8404" Freight="0.7210" />
    <LineItem Line="10" PID="218" Qty="6" Price="5.7000" Freight="0.8550" />
    <LineItem Line="11" PID="223" Qty="2" Price="5.1865" Freight="0.2593" />
    <LineItem Line="12" PID="220" Qty="4" Price="20.1865" Freight="2.0187" />
  </Order>
  <Order OrderNumber="S043660">
    <LineItem Line="1" PID="326" Qty="1" Price="419.4589" Freight="10.4865" />
    <LineItem Line="2" PID="319" Qty="1" Price="874.7940" Freight="21.8699" />
  </Order>
<!--Many more orders here -->
</Orders>
```

Your second objective is to determine whether it's faster to use *XmlDocument* or *XmlReader* to retrieve this data, because you need to process many of these files every day, and performance is critical.

This practice is intended to focus on the features that have been defined in this lesson, so a Console Application project will be implemented. The first exercise implements the solution based on *XmlDocument*, whereas the second exercise implements the solution based on *XmlReader*.

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

EXERCISE 1 Creating the Project and Implementing the *XmlDocument* Solution

In this exercise, you create a Console Application project and add code to retrieve the necessary data by using the *XmlDocument* class.

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

After Visual Studio .NET finishes creating the project, Module1.vb (C# Program.cs) will be displayed.

3. In *Main*, declare a *string* variable for your file name and assign "Orders.xml" to it. Add the *parseWithXmlDocument* method and pass the file name as a parameter. Add this method to your code. Finally, add code to prompt the user to press Enter to end the application. Your code should look like the following:

Sample of Visual Basic Code

```
Module Module1
    Sub Main()
        Dim fileName = "Orders.xml"
        parseWithXmlDocument(fileName)

        Console.WriteLine("Press <Enter> to end")
        Console.ReadLine()
    End Sub

    Private Sub parseWithXmlDocument(ByVal fileName As String)

    End Sub
End Module
```

Sample of C# Code

```
namespace OrderProcessor
{
    class Program
    {
        static void Main(string[] args)
        {
            string fileName = "Orders.xml";
            parseWithXmlDocument(fileName);

            Console.WriteLine("Press <Enter> to end");
            Console.ReadLine();
        }

        private static void parseWithXmlDocument(string fileName)
        {
```



```

    }
  }
}

```

4. Add the Orders.xml file to your project. Right-click the *OrderProcessor* node in Solution Explorer and choose Add | Existing Item. In the bottom right corner of the Add Existing Item dialog box, click the drop-down list and select All Files (*.*). Navigate to the Begin folder for this exercise and select Orders.xml. If you don't see the Orders.xml file, check whether All Files (*.*) has been selected.
5. In Solution Explorer, click the Orders.xml file you just added. In the Properties window, set the *Copy to Output Directory* property to *Copy If Newer*.
Because this file will reside in the same folder as your application, you will be able to use the file name without specifying a path.
6. In the *parseWithXmlDocument* method, instantiate a *Stopwatch* and assign the object to a variable. Start the stopwatch. Declare variables for the total order price, total freight cost, average freight cost, and order count. In C#, add *using System.Diagnostics;* to the top of the file. Your code should look like the following:

Sample of Visual Basic Code

```

Private Sub parseWithXmlDocument(ByVal fileName As String)
    Dim sw = New Stopwatch()
    sw.Start()
    Dim totalOrderPrice As Decimal = 0
    Dim totalFreightCost As Decimal = 0
    Dim orderQty As Integer = 0

End Sub

```

Sample of C# Code

```

private static void parseWithXmlDocument(string fileName)
{
    var sw = new Stopwatch();
    sw.Start();
    decimal totalOrderPrice = 0;
    decimal totalFreightCost = 0;
    decimal orderQty = 0;

}

```

7. Add code to load the Orders.xml file into an *XmlDocument* object. You must also add *imports System.Xml.Linq* (C# using *System.Xml.Linq*). Add code to get the order count by implementing an XPath query to get the *Order* elements and the count of elements returned. Your code should look like the following:

Sample of Visual Basic Code

```

Dim doc = New XmlDocument()
doc.Load(fileName)
orderQty = doc.SelectNodes("//Order").Count

```

Sample of C# Code

```
var doc = new XmlDocument();
doc.Load(fileName);
orderQty = doc.SelectNodes("//Order").Count;
```

8. Add code to retrieve a node list containing all the line items by implementing an XPath query. Loop over all the line items and retrieve the freight and line price (quantity x price). Add the line price and the freight to the total order price. Add the freight to the total freight price. Your code should look like the following:

Sample of Visual Basic Code

```
For Each node As XmlNode In doc.SelectNodes("//LineItem")
    Dim freight = CDec(node.Attributes("Freight").Value)
    Dim linePrice = CDec(node.Attributes("Price").Value) _
        * CDec(node.Attributes("Qty").Value)
    totalOrderPrice += linePrice + freight
    totalFreightCost += freight
Next
```

Sample of C# Code

```
foreach (XmlNode node in doc.SelectNodes("//LineItem"))
{
    var freight = decimal.Parse(node.Attributes["Freight"].Value);
    var linePrice = decimal.Parse(node.Attributes["Price"].Value)
        * decimal.Parse(node.Attributes["Qty"].Value);
    totalOrderPrice += linePrice + freight;
    totalFreightCost += freight;
}
```

9. Add code to display the total order price, the total freight cost, and the average freight cost per order. Stop the stopwatch and display the elapsed time. Your completed method should look like the following:

Sample of Visual Basic Code

```
Private Sub parseWithXmlDocument(ByVal fileName As String)
    Dim sw = New Stopwatch()
    sw.Start()
    Dim totalOrderPrice As Decimal = 0
    Dim totalFreightCost As Decimal = 0
    Dim averageFreightCost As Decimal = 0
    Dim orderQty As Integer = 0

    Dim doc = New XmlDocument()
    doc.Load(fileName)
    orderQty = doc.SelectNodes("//Order").Count

    For Each node As XmlNode In doc.SelectNodes("//LineItem")
        Dim freight = CDec(node.Attributes("Freight").Value)
        Dim linePrice = CDec(node.Attributes("Price").Value) _
            * CDec(node.Attributes("Qty").Value)
        totalOrderPrice += linePrice + freight
        totalFreightCost += freight
    Next
```

```

Next

Console.WriteLine("Total Order Price: {0:C}", totalOrderPrice)
Console.WriteLine("Total Freight Cost: {0:C}", totalFreightCost)
Console.WriteLine("Average Freight Cost per Order: {0:C}", _
    totalFreightCost / orderQty)

sw.Stop()
Console.WriteLine("Time to Parse XmlDocument: {0}", sw.Elapsed)
End Sub

```

Sample of C# Code

```

private static void parseWithXmlDocument(string fileName)
{
    var sw = new Stopwatch();
    sw.Start();
    decimal totalOrderPrice = 0;
    decimal totalFreightCost = 0;
    decimal averageFreightCost = 0;
    decimal orderQty = 0;

    var doc = new XmlDocument();
    doc.Load(fileName);
    orderQty = doc.SelectNodes("//Order").Count;

    foreach (XmlNode node in doc.SelectNodes("//LineItem"))
    {
        var freight = decimal.Parse(node.Attributes["Freight"].Value);
        var linePrice = decimal.Parse(node.Attributes["Price"].Value)
            * decimal.Parse(node.Attributes["Qty"].Value);
        totalOrderPrice += linePrice + freight;
        totalFreightCost += freight;
    }

    Console.WriteLine("Total Order Price: {0:C}", totalOrderPrice);
    Console.WriteLine("Total Freight Cost: {0:C}", totalFreightCost);
    Console.WriteLine("Average Freight Cost per Order: {0:C}",
        totalFreightCost/orderQty);

    sw.Stop();
    Console.WriteLine("Time to Parse XmlDocument: {0}", sw.Elapsed);
}

```

10. Run the application. Your total time will vary based on your machine configuration, but your output should look like the following:

Result

```

Total Order Price: $82,989,370.79
Total Freight Cost: $2,011,265.92
Average Freight Cost per Order: $529.84
Time to Parse XmlDocument: 00:00:01.3775482
Press <Enter> to end

```

EXERCISE 2 Implementing the *XmlReader* Solution

In this exercise, you extend the Console Application project from Exercise 1 by adding code to retrieve the necessary data using the *XmlReader* class.

1. In Visual Studio .NET 2010, choose File | Open | Project.
2. Select the project you created in Exercise 1.
3. In *Main*, after the call to *parseWithXmlDocument*, add the *parseWithXmlReader* method and pass the file name as a parameter. Add this method to your code. Your code should look like the following:

Sample of Visual Basic Code

```
Sub Main()  
    Dim fileName = "Orders.xml"  
    parseWithXmlDocument(fileName)  
    parseWithXmlReader(fileName)  
    Console.WriteLine("Press <Enter> to end")  
    Console.ReadLine()  
End Sub  
  
Private Sub parseWithXmlReader(ByVal fileName As String)  
  
End Sub
```

Sample of C# Code

```
static void Main(string[] args)  
{  
    string fileName = "Orders.xml";  
    parseWithXmlDocument(fileName);  
    parseWithXmlReader(fileName);  
    Console.WriteLine("Press <Enter> to end");  
    Console.ReadLine();  
}  
  
private static void parseWithXmlReader(string fileName)  
{  
  
}
```

4. In the *parseWithXmlReader* method, instantiate *Stopwatch* and assign the object to a variable. Start the stopwatch. Declare variables for the total order price, total freight cost, and order count. Your code should look like the following:

Sample of Visual Basic Code

```
Private Sub parseWithXmlReader(ByVal fileName As String)  
    Dim sw = New Stopwatch()  
    sw.Start()  
    Dim totalOrderPrice As Decimal = 0  
    Dim totalFreightCost As Decimal = 0  
    Dim orderQty As Integer = 0  
  
End Sub
```

Sample of C# Code

```
private static void parseWithXmlReader(string fileName)
{
    var sw = new Stopwatch();
    sw.Start();
    decimal totalOrderPrice = 0;
    decimal totalFreightCost = 0;
    decimal orderQty = 0;

}
```

5. Add a *using* statement to instantiate an *XmlTextReader* object and assign the object to a variable named *xmlReader*. In the *using* statement, add a *while* loop to iterate over all nodes. In the loop, add code to check the node type to see whether it is an element. Your code should look like the following:

Sample of Visual Basic Code

```
Using xmlReader As New XmlTextReader(fileName)
    Do While xmlReader.Read()
        If xmlReader.NodeType = XmlNodeType.Element Then

            End If
        Loop
    End Using
```

Sample of C# Code

```
using (var xmlReader = new XmlTextReader(fileName))
{
    while (xmlReader.Read())
    {
        if(xmlReader.NodeType==XmlNodeType.Element)
        {

        }
    }
}
```

6. Inside the *if* statement, add a *select* (C# *switch*) statement that increments the order quantity variable if the element's node name is *Order*. If the node name is *LineItem*, add code to retrieve the quantity, price, and freight. Add the freight to the total freight and add the total cost of the line to the total order cost variable. Your code should look like the following:

Sample of Visual Basic Code

```
Select Case xmlReader.Name
    Case "Order"
        orderQty += 1
    Case "LineItem"
        Dim qty = CDec(xmlReader.GetAttribute("Qty"))
        Dim price = CDec(xmlReader.GetAttribute("Price"))
        Dim freight = CDec(xmlReader.GetAttribute("Freight"))
        totalFreightCost += freight
```

```

        totalOrderPrice += (qty * price) + freight
    End Select

```

Sample of C# Code

```

switch(xmlReader.Name)
{
    case "Order":
        ++orderQty;
        break;
    case "LineItem":
        var qty = decimal.Parse(xmlReader.GetAttribute("Qty"));
        var price = decimal.Parse(xmlReader.GetAttribute("Price"));
        var freight = decimal.Parse(xmlReader.GetAttribute("Freight"));
        totalFreightCost += freight;
        totalOrderPrice += (qty * price) + freight;
        break;
}

```

7. Add code to display the total order price, the total freight cost, and the average freight cost per order. Stop the stopwatch and display the elapsed time. Your completed method should look like the following:

Sample of Visual Basic Code

```

Private Sub parseWithXmlReader(ByVal fileName As String)
    Dim sw = New Stopwatch()
    sw.Start()
    Dim totalOrderPrice As Decimal = 0
    Dim totalFreightCost As Decimal = 0
    Dim averageFreightCost As Decimal = 0
    Dim orderQty As Integer = 0

    Using xmlReader As New XmlTextReader(fileName)
        Do While xmlReader.Read()
            If xmlReader.NodeType = XmlNodeType.Element Then
                Select Case xmlReader.Name
                    Case "Order"
                        orderQty += 1
                    Case "LineItem"
                        Dim qty = CDec(xmlReader.GetAttribute("Qty"))
                        Dim price = CDec(xmlReader.GetAttribute("Price"))
                        Dim freight = CDec(xmlReader.GetAttribute("Freight"))
                        totalFreightCost += freight
                        totalOrderPrice += (qty * price) + freight
                End Select
            End If
        Loop
    End Using

    Console.WriteLine("Total Order Price: {0:C}", totalOrderPrice)
    Console.WriteLine("Total Freight Cost: {0:C}", totalFreightCost)
    Console.WriteLine("Average Freight Cost per Order: {0:C}", _
        totalFreightCost / orderQty)

    sw.Stop()

```

```

        Console.WriteLine("Time to Parse XmlReader: {0}", sw.Elapsed)
    End Sub

```

Sample of C# Code

```

private static void parseWithXmlReader(string fileName)
{
    var sw = new Stopwatch();
    sw.Start();
    decimal totalOrderPrice = 0;
    decimal totalFreightCost = 0;
    decimal averageFreightCost = 0;
    decimal orderQty = 0;

    using (var xmlReader = new XmlTextReader(fileName))
    {
        while (xmlReader.Read())
        {
            if (xmlReader.NodeType == XmlNodeType.Element)
            {
                switch (xmlReader.Name)
                {
                    case "Order":
                        ++orderQty;
                        break;
                    case "LineItem":
                        var qty = decimal.Parse(xmlReader.GetAttribute("Qty"));
                        var price = decimal.Parse(xmlReader.GetAttribute("Price"));
                        var freight = decimal.Parse(
                            xmlReader.GetAttribute("Freight"));
                        totalFreightCost += freight;
                        totalOrderPrice += (qty * price) + freight;
                        break;
                }
            }
        }
    }
    Console.WriteLine("Total Order Price: {0:C}", totalOrderPrice);
    Console.WriteLine("Total Freight Cost: {0:C}", totalFreightCost);
    Console.WriteLine("Average Freight Cost per Order: {0:C}",
        totalFreightCost / orderQty);

    sw.Stop();
    Console.WriteLine("Time to Parse XmlReader: {0}", sw.Elapsed);
}

```

8. Run the application. Your total time will vary based on your machine configuration, but you should find that *XmlReader* is substantially faster. Your output should look like the following, which includes the result from Exercise 1.

Result

```

Total Order Price: $82,989,370.79
Total Freight Cost: $2,011,265.92
Average Freight Cost per Order: $529.84
Time to Parse XmlDocument: 00:00:01.2218770

```

Total Order Price: \$82,989,370.79
Total Freight Cost: \$2,011,265.92
Average Freight Cost per Order: \$529.84
Time to Parse XmlReader: 00:00:00.5919724
Press <Enter> to end

Lesson Summary

This lesson provided detailed information about the *XmlDocument* and the *XmlReader* classes.

- The *XmlDocument* class provides in-memory, random, read-write access to XML nodes.
- The *XmlReader* class provides fast-streaming, forward-only, read-only access to XML nodes.
- The *XmlDocument* class is easier to use, whereas the *XmlReader* class is faster.
- The *XmlDocument* class enables you to retrieve XML nodes by using the element name.
- The *XmlDocument* class enables you to retrieve XML nodes by using an XPath query.

Lesson Review

You can use the following questions to test your knowledge of the information in Lesson 1, "Working with the *XmlDocument* and *XmlReader* Classes." 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 an XML file, you want to run several queries for data in the file based on filter criteria the user will be entering for a particular purpose. Which class would be more appropriate for these queries on the file?
A. *XmlDocument*
B. *XmlReader*
2. Every day, you receive hundreds of XML files. You are responsible for reading the file to retrieve sales data from the file and store it into a SQL database. Which class would be more appropriate for these queries on the file?
A. *XmlDocument*
B. *XmlReader*

3. You have a service that receives a very large XML-based history file once a month. These files can be up to 20GB in size, and you need to retrieve the header information that contains the history date range and the customer information on which this file is based. Which class would be more appropriate to retrieve the data in these files?
- A. *XmlDocument*
 - B. *XmlReader*

Lesson 2: Querying with LINQ to XML

The use of LINQ to XML provides a powerful means to query XML data, using a language-based syntax that supports IntelliSense. LINQ to XML also enables you to transform XML in a very simple manner. This lesson covers the various aspects of querying XML by using LINQ to XML, and the next lesson explores transformations.

After this lesson, you will be able to:

- Understand the *XObject* class hierarchy.
- Create *XDocument* objects.
- Implement LINQ to XML queries.
- Perform LINQ to XML queries with aggregates.
- Execute LINQ to XML joins.
- Use LINQ to XML with namespaces.

Estimated lesson time: 45 minutes

Introducing the *XDocument* Family

In the quest to design a way to use LINQ over XML data, Microsoft needed to create a new set of classes that would simplify the task, so it created the *XDocument* classes: *XElement*, *XAttribute*, *XNamespace*, *XDirective*, and more. To use these classes, you must add a reference to the *System.Xml.Linq.dll* assembly, and then you can add an *imports* (C# *using*) *System.Xml.Linq* statement to your code. Figure 5-1 shows many of the classes that comprise the *XDocument* family.

As Figure 5-1 shows, many, but not all, classes inherit from the *XNode* class. In the class hierarchy, the *XAttribute* class is not an *XNode*. This lesson focuses on the *XDocument*, *XElement*, *XAttribute*, and *XNamespace* classes and their parent classes.

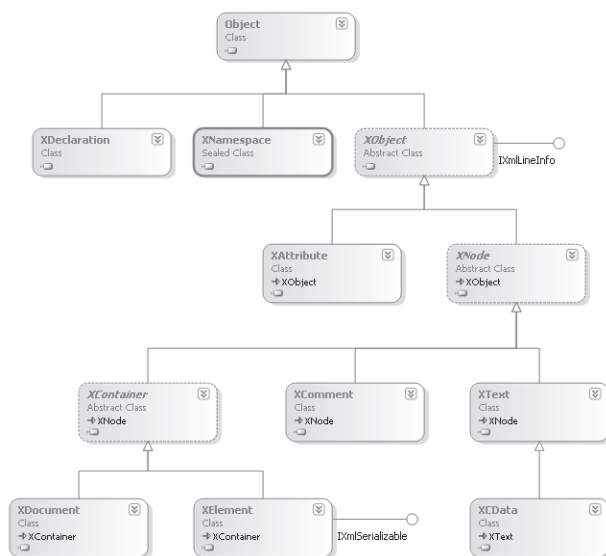


FIGURE 5-1 The *XDocument* class family simplifies LINQ access to XML data.

The *XObject* Class

Most of the *XDocument* classes derive from the *XObject* class. This is a *MustInherit* (C# *abstract*) class that contains the following members that will be inherited.

- **BaseUri** A read-only property containing the base URI string for the current object.
- **Document** A read-only property that references the *XDocument* object to which the current object belongs.
- **NodeType** An abstract, read-only property that returns *XmlNodeType* of the current node. Child classes override this to return the proper node type.
- **Parent** A read-only property that returns the *XElement* parent of the current node.
- **AddAnnotation** A method for adding an annotation object to the object's annotations list. Think of an annotation as something like a notation—or, better yet, as an attachment. The annotation is not limited to being a string; it can be any object you want to attach to the current *XObject* instance. Annotations are not persisted and do not show when using the *ToString* method to retrieve the XML representation as a string.
- **Annotation** Generic and non-generic methods that return the first annotation of the type passed to this method as a generic parameter or regular method parameter.
- **Annotations** Generic and non-generic methods that return a collection of the type passed to this method as a generic parameter or regular method parameter.

- **RemoveAnnotations** Generic and non-generic methods that remove the annotations of the type passed to this method as a generic parameter or regular method parameter.
- **Changed** An event that occurs if the current *XObject* class or any of its descendents have changed.
- **Changing** An event that occurs before the current *XObject* class or any of its descendents change.

In addition to these members, the *XObject* class explicitly implements the *IXmlLineInfo* interface, which contains *LineNumber* and *LinePosition* properties and the *HasLineInfo* method.

The *XAttribute* Class

The *XAttribute* object represents an XML attribute. The *XAttribute* class is derived from the *XObject* class, so it inherits all the members of the *XObject* class. In addition, the *XAttribute* class contains the following members.

- **EmptySequence** A shared (C# *static*) read-only property that returns an empty generic *IEnumerable* of *XAttribute*.
- **IsNameSpaceDeclaration** A read-only property that returns *true* if the current *XAttribute* object represents a namespace identifier.
- **Name** A read-only property that returns the name of the XML attribute as a string.
- **NextAttribute** A read-only property that returns a reference to the next XML attribute. Returns *null* if the current *XAttribute* object has no parent or if the current *XAttribute* object is the last XML attribute of the parent.
- **NodeType** Overrides the inherited *NodeType* property from *XObject*. Returns *XmlNodeType.XAttribute*.
- **PreviousAttribute** A read-only property that returns a reference to the previous XML attribute. Returns *null* if the current *XAttribute* object has no parent or if there is no previous XML attribute.
- **Value** A string property that gets or sets the value of the XML attribute. When you attempt to set this property, the property is validated before the value is set. If you attempt to assign *Nothing* (C# *null*) to this property, an *ArgumentNullException* is thrown. Also, if this attribute is a namespace declaration, the namespace is validated. If validation succeeds, the *Changing* event inherited from *XObject* is raised, the value is set, and the *Changed* event is raised.
- **Operator** A group of 25 explicit conversion operators that enable you to convert the *XAttribute* object to many types by using *CType* (C# *explicit cast*) based on the XML attribute's value. Internally, this uses the *XmlConvert* class to convert the value of the XML attribute to the desired type.

- **Remove** A method that removes the current *XAttribute* object from the parent element. It throws an *InvalidOperationException* if the current *XAttribute* object has no parent.
- **SetValue** A method that attempts to make the object parameter a string that is passed in and assigns the string to the current *XAttribute* object's value.
- **ToString** A method that overrides the *ToString* method inherited from the *Object* class. This override emits the name and value of the XML attribute as you would expect to see it in an XML document.

The *XNode* Class

The *XNode* class is derived from the *XObject* class, so it contains all the *XObject* members. The *XNode* class contains the following members.

- **DocumentOrderComparer** A shared (C# *static*), read-only property that returns a comparer of the type *XNodeDocumentOrderComparer*. This comparer can compare the relative positions of two nodes.
- **EqualityComparer** A shared (C# *static*), read-only property that returns a comparer of the type *XNodeEqualityComparer*. This comparer can compare two nodes for equality.
- **NextNode** A read-only property that returns a reference to the next node or *Nothing* (C# *null*) if there is no next node.
- **PreviousNode** A read-only property that returns a reference to the previous node or *Nothing* (C# *null*) if there is no previous node.
- **AddAfterSelf** A method that enables you to pass in content that should be added after the current node object.
- **AddBeforeSelf** A method that enables you to pass in content that should be added before the current node object.
- **Ancestors** A method that accepts a name parameter and returns a collection of the ancestor elements of the current node that have the specified name.
- **CompareDocumentOrder** A shared (C# *static*) method that compares two nodes to determine their relative XML document order. This method accepts two *XNode* objects and returns an *Integer* (C# *int*) value by which *0* indicates that the two nodes are equal, *-1* indicates that the first node is before the second node, and *+1* indicates that the first node is after the second node.
- **CreateReader** A method that returns an *XmlReader* that can read the contents of the current node and its descendents.
- **DeepEquals** A shared (C# *static*) method that accepts two *XNode* parameters. This method compares the values of the two elements and the values of the descendents and returns *true* if the nodes are equal.

- **ElementsAfterSelf** A method that returns a collection of the sibling element nodes after this node, in document order. There is also an overload that accepts an *XName* object consisting of the name of the elements you want to return.
- **ElementsBeforeSelf** A method that returns a collection of the sibling element nodes before this node, in document order. There is also an overload that accepts an *XName* object consisting of the name of the elements you want to return.
- **IsAfter** A method that accepts an *XNode* parameter and returns *true* if the current node is after the *XNode* passed into this method.
- **IsBefore** A method that accepts an *XNode* parameter and returns *true* if the current node is before the *XNode* passed into this method.
- **NodeAfterSelf** A method that returns a reference to the node after the current node. Returns *Nothing* (C# *null*) if there is no node after the current node.
- **NodeBeforeSelf** A method that returns a reference to the node before the current node. Returns *Nothing* (C# *null*) if there is no node before the current node.
- **ReadFrom** A *shared* (C# *static*) method that accepts an *XmlReader* parameter and returns an *XNode* object that represents the first node encountered by *XmlReader*.
- **Remove** A method that removes the current node from its parent. This throws an *InvalidOperationException* if there is no parent.
- **ReplaceWith** A method that replaces the node with the specified content passed in as a parameter. The content can be simple content, a collection of content objects, a parameter list of content objects, or *null*.
- **ToString** A method that overrides the *ToString* method in *Object*. This returns the XML of the current node and its contents as a string.
- **WriteTo** A *MustImplement* (C# *abstract*) method that takes an *XmlWriter* parameter.

The XContainer Class

The *XContainer* class represents a node that can contain other nodes and is a *MustInherit* (C# *abstract*) class that derives from the *XNode* class. It inherits the members of *XNode* and *XObject*. The *XDocument* and *XElement* classes derive from this class. The following is a list of its members.

- **FirstNode** A read-only property that returns a reference to the first node in this container. If there are no nodes in the container, this property returns *Nothing* (C# *null*).
- **LastNode** A read-only property that returns a reference to the last node in this container. If there are no nodes in the container, this property returns *Nothing* (C# *null*).
- **Add** A method that accepts one or several content objects and adds them and their children to this container.
- **AddFirst** A method that accepts one or several content objects and adds them and their children to the current *XContainer* object as the first node.

- **CreateWriter** A method that returns an *XmlWriter* object that can be used to add elements or attributes to the current *XContainer* object.
- **DescendantNodes** A method that returns descendant elements plus leaf nodes contained in the current *XContainer* object.
- **Descendants** A method that returns references to the descendant nodes of the current *XContainer* object as a generic *IEnumerable* of *XElement*. There is also an overload that accepts a name as an *XName* parameter and returns references only to the descendant nodes that match the name parameter. This method does not return a reference to the current *XContainer* object.
- **Element** A method that accepts a name as an *XName* parameter and returns a reference to the first child element with a name that matches the name parameter or returns *null* if no match is found.
- **Elements** A method that returns references to all child elements of the current *XContainer* object as a generic *IEnumerable* of *XElement*. There is also an overload that accepts a name as an *XName* parameter and returns references only to the child nodes that match the name parameter.
- **Nodes** A method that returns a reference to all nodes in the current *XContainer* object as a generic *IEnumerable* of the *XNode* object.
- **RemoveNodes** A method that removes all nodes from the current *XContainer* object. Note that this method does not remove any attributes the *XContainer* object might have.
- **ReplaceNodes** A method that accepts one or several content objects as a parameter and replaces the existing content with the content parameter.

The XElement Class

The *XElement* class represents an XML element with a *Name* property of the type *XName*. The *XName* class is composed of a local name and a namespace. Optionally, *XElement* can contain XML attributes of type *XAttribute*, derives from *XContainer*, and inherits the members of *XContainer*, *XNode*, and *XObject*.

The *XElement* class explicitly implements the *IXmlSerializable* interface, which contains the *GetSchema*, *ReadXml*, and *WriteXml* methods so this object can be serialized. The following are members of the *XElement* class.

- **EmptySequence** A *shared* (C# *static*), read-only property that returns an empty generic *IEnumerable* of the *XElement* object.
- **FirstAttribute** A read-only property that returns a reference to the first XML attribute on this element or *null* if there are no attributes.
- **HasAttributes** A read-only property that returns *true* if the current element has at least one attribute.

- **HasElements** A read-only property that returns *true* if the current element has at least one child element.
- **IsEmpty** A read-only property that returns *true* if the current element has no child elements.
- **LastAttribute** A read-only property that returns a reference to the last XML attribute on this element or *null* if there are no attributes.
- **Name** A read-write property that contains the name of the current element as an *XName* object. The *XName* object contains a local name and a namespace.
- **NodeType** A read-only property that returns *XmlNodeType.Element*.
- **Value** A read-write string property that contains the text of the current element as an *XName* object. If the current element contains a mixture of text and child elements, the text of all children is concatenated and returned. If you pass an *XName* into the property setter, all existing content will be removed and replaced with the *XName* object.
- **AncestorsAndSelf** A method that returns a generic *IEnumerable* of an *XElement* sequence containing references to the current element and all its ancestor elements. There is also an overload that accepts an *XName* parameter for the name of the elements you want to retrieve.
- **Attribute** A method that accepts a name parameter as an *XName* and returns a reference to the attribute with a name that matches on the current element. This method returns *Nothing* (C# *null*) if no attribute is found.
- **Attributes** A method that returns a generic *IEnumerable* of an *XAttribute* sequence containing references to all attributes of the current element. There is also an overload that accepts a name parameter as an *XName* and returns a reference to the attribute with a name that matches on the current element. This method returns an empty sequence if no attribute is found.
- **DescendantNodesAndSelf** A method that returns a generic *IEnumerable* of an *XNode* sequence containing references to the current node and all its descendant nodes.
- **DescendantsAndSelf** A method that returns a generic *IEnumerable* of an *XElement* sequence containing references to the current element and all its descendant elements. There is also an overload that accepts a string parameter for the name of the elements you want to retrieve.
- **GetDefaultNamespace** A method that returns the default namespace of the current element as an *XNamespace* object. The *XNamespace* type has a *NamespaceName* property that contains the URI of the namespace. If there is no default namespace, this method returns the *XNamespace.None* value.
- **GetNamespaceOfPrefix** A method that accepts a prefix parameter as a string and returns a reference to the associated namespace or *null* if the namespace is not found.

- **GetPrefixOfNamespace** A method that returns the prefix of the current element as a string.
- **Load** A *shared* (C# *static*) method that creates a new *XElement* object and initializes it with the contents passed into the method. There are eight overloads for this method that enable you to pass various objects as parameters, such as *Stream*, *TextReader*, *XmlReader*, or a *string* URL.
- **Operator** There are 25 explicit conversion operators, which enable you to convert the *XElement* object to many types by using *CType* (C# *explicit cast*) based on the XML element's value. Internally, this uses the *XmlConvert* class to convert the value of the XML element to the desired type.
- **Parse** A *shared* (C# *static*) method that creates a new *XElement* object and initializes it with the contents parameter as *string* passed into the method. There is also an overload for this method that enables you to pass load options to preserve white space.
- **RemoveAll** A method that removes all attributes and all elements from the current element.
- **RemoveAttributes** A method that removes all attributes from the current element.
- **ReplaceAll** A method that replaces the child nodes and attributes of the current element with the content parameter passed into this method. The content can be simple content, a collection of content objects, a parameter list of content objects, or *null*.
- **ReplaceAttributes** A method that replaces the attributes of the current element with the content parameter. The content can be simple content, a collection of content objects, a parameter list of content objects, or *null*.
- **Save** A method that outputs the current element's XML tree to a file, *Stream*, *TextWriter*, or *XmlWriter* object.
- **SetElementValue** A method that sets the value of a child element. The value is assigned to the first child element that matches the name parameter. If no child element with a matching name exists, a new child element is added. If the value is *null*, the first child element with the given name, if any, is deleted.
- **SetValue** A method that assigns the *Object* parameter to the *Value* property of the current element by converting the given value to *string*.
- **WriteTo** A method that writes the current element to the *XmlWriter* parameter.

The XDocument Class

The *XDocument* class represents an XML document that can contain a DTD, one root XML element, zero-to-many XML comments, and zero-to-many XML processing instructions. The *XDocument* class derives from *XContainer* and therefore inherits the members of *XContainer*, *XNode*, and *XObject*. The following are members of the *XDocument* class.

- **Declaration** A read/write property of type *XDeclaration*. The *XDeclaration* class contains properties for *Encoding*, *Standalone*, and *Version*.

- **DocumentType** A read-only property of *XDocumentType*, which is the Document Type Definition (DTD).
- **NodeType** A read-only property that returns *XmlNodeType.Document*.
- **Root** A read-only property that returns a reference to the first XML element in the XML document. The property returns *Nothing* (C# *null*) if there is no root element.
- **Load** A *shared* (C# *static*) method that creates a new *XDocument* object and initializes it with the contents passed into the method. There are eight overloads for this method that enable you to load content from a *Stream*, *TextReader*, *XmlReader*, or *string* URI.
- **Parse** A *shared* (C# *static*) method that creates a new *XDocument* object and initializes it with the XML passed as a *string* parameter. There is also an overload for this method that enables you to pass load options to preserve white space.
- **Save** A method that outputs the current document's XML tree to a file or *XmlWriter*.
- **WriteTo** A method that writes the current document to a *Stream*, *XmlWriter*, *TextWriter*, or file.

Using the *XDocument* Classes

The previous introduction to the *XDocument* classes described the members of the more common classes, whereas this section will implement the *XDocument* family of classes.

The *XDocument* class can be populated very easily by using the constructor, the *Load* method, or the *Parse* method. The following code sample uses the *Parse* method to load an XML string into an *XDocument* object. After that, the *Save* method saves the XML document to the *XDocumentTest.xml* file.

Sample of Visual Basic Code

```
Private Sub ParseXDocumentToolStripMenuItem_Click( _
    ByVal sender As System.Object, ByVal e As System.EventArgs) _
    Handles ParseXDocumentToolStripMenuItem.Click
    Dim xml = _
        "<CustomersOrders>" & _
        "  <Customer CustomerID='ALFKI' CompanyName='Alfreds Futterkiste'>" & _
        "    <Order OrderID='10643' Freight='29.4600' />" & _
        "    <Order OrderID='10692' Freight='61.0200' />" & _
        "    <Order OrderID='10702' Freight='23.9400' />" & _
        "    <Order OrderID='10835' Freight='69.5300' />" & _
        "    <Order OrderID='10952' Freight='40.4200' />" & _
        "    <Order OrderID='11011' Freight='1.2100' />" & _
        "  </Customer>" & _
        "  <Customer CustomerID='ANATR' CompanyName='Ana Trujillo'>" & _
        "    <Order OrderID='10308' Freight='1.6100' />" & _
        "    <Order OrderID='10625' Freight='43.9000' />" & _
        "    <Order OrderID='10759' Freight='11.9900' />" & _
        "    <Order OrderID='10926' Freight='39.9200' />" & _
        "  </Customer>" & _
        "  <Customer CustomerID='ANTON' CompanyName='Antonio Moreno'>" & _
        "    <Order OrderID='10365' Freight='22.0000' />" & _
        "    <Order OrderID='10507' Freight='47.4500' />" & _
```

```

"      <Order OrderID='10535' Freight='15.6400' />" & _
"      <Order OrderID='10573' Freight='84.8400' />" & _
"      <Order OrderID='10677' Freight='4.0300' />" & _
"      <Order OrderID='10682' Freight='36.1300' />" & _
"      <Order OrderID='10856' Freight='58.4300' />" & _
"    </Customer>" & _
"    <Customer CustomerID='AROUT' CompanyName='Around the Horn'>" & _
"      <Order OrderID='10355' Freight='41.9500' />" & _
"      <Order OrderID='10383' Freight='34.2400' />" & _
"      <Order OrderID='10453' Freight='25.3600' />" & _
"      <Order OrderID='10558' Freight='72.9700' />" & _
"      <Order OrderID='10707' Freight='21.7400' />" & _
"      <Order OrderID='10741' Freight='10.9600' />" & _
"      <Order OrderID='10743' Freight='23.7200' />" & _
"      <Order OrderID='10768' Freight='146.3200' />" & _
"      <Order OrderID='10793' Freight='4.5200' />" & _
"      <Order OrderID='10864' Freight='3.0400' />" & _
"      <Order OrderID='10920' Freight='29.6100' />" & _
"      <Order OrderID='10953' Freight='23.7200' />" & _
"      <Order OrderID='11016' Freight='33.8000' />" & _
"    </Customer>" & _
"</CustomersOrders>"
Dim doc = XDocument.Parse(xml)
doc.Save(getFilePath("XDocumentTest.xml"))
MessageBox.Show("XDocument Saved")
End Sub

```

Sample of C# Code

```

private void parseXDocumentToolStripMenuItem_Click(
    object sender, EventArgs e)
{
    string xml = @"
    <CustomersOrders>
      <Customer CustomerID='ALFKI' CompanyName='Alfreds Futterkiste'>
        <Order OrderID='10643' Freight='29.4600' />
        <Order OrderID='10692' Freight='61.0200' />
        <Order OrderID='10702' Freight='23.9400' />
        <Order OrderID='10835' Freight='69.5300' />
        <Order OrderID='10952' Freight='40.4200' />
        <Order OrderID='11011' Freight='1.2100' />
      </Customer>
      <Customer CustomerID='ANATR' CompanyName='Ana Trujillo'>
        <Order OrderID='10308' Freight='1.6100' />
        <Order OrderID='10625' Freight='43.9000' />
        <Order OrderID='10759' Freight='11.9900' />
        <Order OrderID='10926' Freight='39.9200' />
      </Customer>
      <Customer CustomerID='ANTON' CompanyName='Antonio Moreno'>
        <Order OrderID='10365' Freight='22.0000' />
        <Order OrderID='10507' Freight='47.4500' />
        <Order OrderID='10535' Freight='15.6400' />
        <Order OrderID='10573' Freight='84.8400' />
        <Order OrderID='10677' Freight='4.0300' />
        <Order OrderID='10682' Freight='36.1300' />
        <Order OrderID='10856' Freight='58.4300' />
      </Customer>
    </CustomersOrders>
  "
}

```

```

        </Customer>
        <Customer CustomerID='AROUT' CompanyName='Around the Horn'>
            <Order OrderID='10355' Freight='41.9500' />
            <Order OrderID='10383' Freight='34.2400' />
            <Order OrderID='10453' Freight='25.3600' />
            <Order OrderID='10558' Freight='72.9700' />
            <Order OrderID='10707' Freight='21.7400' />
            <Order OrderID='10741' Freight='10.9600' />
            <Order OrderID='10743' Freight='23.7200' />
            <Order OrderID='10768' Freight='146.3200' />
            <Order OrderID='10793' Freight='4.5200' />
            <Order OrderID='10864' Freight='3.0400' />
            <Order OrderID='10920' Freight='29.6100' />
            <Order OrderID='10953' Freight='23.7200' />
            <Order OrderID='11016' Freight='33.8000' />
        </Customer>
    </CustomersOrders>
";
var doc = XDocument.Parse(xml);
doc.Save(getFilePath("XDocumentTest.xml"));
MessageBox.Show("XDocument Saved");
}

```

Looking at the sample code, you might like the C# code better than the Visual Basic code because C# enables you to create a string that spans many rows. Visual Basic has an alternate way to load XML into an *XDocument*, as shown in the following code sample, which implements the *XDocument* constructor.

Sample of Visual Basic Code

```

Private Sub XDocumentConstructorToolStripMenuItem_Click( _
    ByVal sender As System.Object, ByVal e As System.EventArgs) _
    Handles XDocumentConstructorToolStripMenuItem.Click
    Dim doc As XDocument = _
        <?xml version="1.0" encoding="utf-8" standalone="yes"?>
        <CustomersOrders>
            <Customer CustomerID='ALFKI' CompanyName='Alfreds Futterkiste'>
                <Order OrderID='10643' Freight='29.4600' />
                <Order OrderID='10692' Freight='61.0200' />
                <Order OrderID='10702' Freight='23.9400' />
                <Order OrderID='10835' Freight='69.5300' />
                <Order OrderID='10952' Freight='40.4200' />
                <Order OrderID='11011' Freight='1.2100' />
            </Customer>
            <Customer CustomerID='ANATR' CompanyName='Ana Trujillo'>
                <Order OrderID='10308' Freight='1.6100' />
                <Order OrderID='10625' Freight='43.9000' />
                <Order OrderID='10759' Freight='11.9900' />
                <Order OrderID='10926' Freight='39.9200' />
            </Customer>
            <Customer CustomerID='ANTON' CompanyName='Antonio Moreno'>
                <Order OrderID='10365' Freight='22.0000' />
                <Order OrderID='10507' Freight='47.4500' />
                <Order OrderID='10535' Freight='15.6400' />
                <Order OrderID='10573' Freight='84.8400' />
            </Customer>
        </CustomersOrders>
    </XDocument>

```

```

        <Order OrderID='10677' Freight='4.0300' />
        <Order OrderID='10682' Freight='36.1300' />
        <Order OrderID='10856' Freight='58.4300' />
    </Customer>
    <Customer CustomerID='AROUT' CompanyName='Around the Horn'>
        <Order OrderID='10355' Freight='41.9500' />
        <Order OrderID='10383' Freight='34.2400' />
        <Order OrderID='10453' Freight='25.3600' />
        <Order OrderID='10558' Freight='72.9700' />
        <Order OrderID='10707' Freight='21.7400' />
        <Order OrderID='10741' Freight='10.9600' />
        <Order OrderID='10743' Freight='23.7200' />
        <Order OrderID='10768' Freight='146.3200' />
        <Order OrderID='10793' Freight='4.5200' />
        <Order OrderID='10864' Freight='3.0400' />
        <Order OrderID='10920' Freight='29.6100' />
        <Order OrderID='10953' Freight='23.7200' />
        <Order OrderID='11016' Freight='33.8000' />
    </Customer>
</CustomersOrders>
doc.Save(getFilePath("XDocumentTest.xml"))
MessageBox.Show("XDocument Saved")
End Sub

```

Sample of C# Code

```

private void xDocumentConstructorToolStripMenuItem_Click(
    object sender, EventArgs e)
{
    var doc = new XDocument(
        new XElement("CustomersOrders",
            new XElement("Customer",
                new XAttribute("CustomerID", "ALFKI"),
                new XAttribute("CompanyName", "Alfreds Futterkiste"),
                new XElement("Order",
                    new XAttribute("OrderID", "10643"),
                    new XAttribute("Freight", "29.4600")),
                new XElement("Order",
                    new XAttribute("OrderID", "10692"),
                    new XAttribute("Freight", "61.0200")),
                new XElement("Order",
                    new XAttribute("OrderID", "10702"),
                    new XAttribute("Freight", "23.9400")),
                new XElement("Order",
                    new XAttribute("OrderID", "10835"),
                    new XAttribute("Freight", "69.5300")),
                new XElement("Order",
                    new XAttribute("OrderID", "10952"),
                    new XAttribute("Freight", "40.4200")),
                new XElement("Order",
                    new XAttribute("OrderID", "11011"),
                    new XAttribute("Freight", "1.2100"))),
            new XElement("Customer",
                new XAttribute("CustomerID", "ANATR"),
                new XAttribute("CompanyName", "Ana Trujillo"),
                new XElement("Order",

```

```

        new XAttribute("OrderID", "10308"),
        new XAttribute("Freight", "1.6100")),
new XElement("Order",
    new XAttribute("OrderID", "10625"),
    new XAttribute("Freight", "43.9000")),
new XElement("Order",
    new XAttribute("OrderID", "10759"),
    new XAttribute("Freight", "11.9900")),
new XElement("Order",
    new XAttribute("OrderID", "10926"),
    new XAttribute("Freight", "39.9200"))),
new XElement("Customer",
    new XAttribute("CustomerID", "ANTON"),
    new XAttribute("CompanyName", "Antonio Moreno"),
    new XElement("Order",
        new XAttribute("OrderID", "10365"),
        new XAttribute("Freight", "22.0000")),
    new XElement("Order",
        new XAttribute("OrderID", "10507"),
        new XAttribute("Freight", "47.4500")),
    new XElement("Order",
        new XAttribute("OrderID", "10535"),
        new XAttribute("Freight", "15.6400")),
    new XElement("Order",
        new XAttribute("OrderID", "10573"),
        new XAttribute("Freight", "84.8400")),
    new XElement("Order",
        new XAttribute("OrderID", "10677"),
        new XAttribute("Freight", "4.0300")),
    new XElement("Order",
        new XAttribute("OrderID", "10682"),
        new XAttribute("Freight", "36.1300")),
    new XElement("Order",
        new XAttribute("OrderID", "10856"),
        new XAttribute("Freight", "58.5300"))),
new XElement("Customer",
    new XAttribute("CustomerID", "AROUT"),
    new XAttribute("CompanyName", "Around the Horn"),
    new XElement("Order",
        new XAttribute("OrderID", "10355"),
        new XAttribute("Freight", "41.9500")),
    new XElement("Order",
        new XAttribute("OrderID", "10383"),
        new XAttribute("Freight", "34.2400")),
    new XElement("Order",
        new XAttribute("OrderID", "10453"),
        new XAttribute("Freight", "25.3600")),
    new XElement("Order",
        new XAttribute("OrderID", "10558"),
        new XAttribute("Freight", "72.9700")),
    new XElement("Order",
        new XAttribute("OrderID", "10707"),
        new XAttribute("Freight", "21.7400")),
    new XElement("Order",
        new XAttribute("OrderID", "10741"),

```

```

        new XAttribute("Freight", "10.9600")),
    new XElement("Order",
        new XAttribute("OrderID", "10743"),
        new XAttribute("Freight", "23.7200")),
    new XElement("Order",
        new XAttribute("OrderID", "10768"),
        new XAttribute("Freight", "146.3200")),
    new XElement("Order",
        new XAttribute("OrderID", "10793"),
        new XAttribute("Freight", "4.5200")),
    new XElement("Order",
        new XAttribute("OrderID", "10864"),
        new XAttribute("Freight", "3.0400")),
    new XElement("Order",
        new XAttribute("OrderID", "10920"),
        new XAttribute("Freight", "29.6100")),
    new XElement("Order",
        new XAttribute("OrderID", "10953"),
        new XAttribute("Freight", "23.7200")),
    new XElement("Order",
        new XAttribute("OrderID", "11016"),
        new XAttribute("Freight", "33.8000"))));
doc.Save(getFilePath("XDocumentTest.xml"));
MessageBox.Show("XDocument Saved");
}

```

In this code sample, the Visual Basic code is much simpler than the C# code. The Visual Basic compiler parses the XML literals and creates the same code that the C# compiler creates. This is very different from the previous example, in which the *Parse* method parsed an XML string. In this example, the code runs faster than the *Parse* example because there is no need to parse at run time.

Implementing LINQ to XML Queries

You can use many of the techniques covered in Chapter 3, “Introducing LINQ,” to query an *XDocument* or *XElement* object. By using the XML from the previous examples, you might want to return the freight and customer ID for order ID 10677, which can be accomplished as shown in the following code sample:

Sample of Visual Basic Code

```

Private Sub LINQQueryToolStripMenuItem_Click( _
    ByVal sender As System.Object, ByVal e As System.EventArgs) _
    Handles LINQQueryToolStripMenuItem.Click
    Dim doc = XDocument.Load(getFilePath("XDocumentTest.xml"))
    Dim result = (From order In doc.Descendants("Order")
        Where order.Attribute("OrderID").Value = "10677"
        Select New With
        {
            .OrderID = CType(order.Attribute("OrderID"), Integer),
            .CustomerID = CType(order.Parent.Attribute("CustomerID"), String),
            .Freight = CType(order.Attribute("Freight"), Decimal)
        }).FirstOrDefault()
    txtLog.Text = String.Format("OrderID:{0} CustomerID:{1} Freight:{2:C}", _

```

```

        result.OrderID, result.CustomerID, result.Freight)
End Sub

```

Sample of C# Code

```

private void LINQQueryToolStripMenuItem_Click(
    object sender, EventArgs e)
{
    var doc = XDocument.Load(getFilePath("XDocumentTest.xml"));
    var result = (from order in doc.Descendants("Order")
        where order.Attribute("OrderID").Value == "10677"
        select new
        {
            OrderID=(int)order.Attribute("OrderID"),
            CustomerID = (string)order.Parent.Attribute("CustomerID"),
            Freight = (decimal)order.Attribute("Freight")
        }).FirstOrDefault();
    txtLog.Text = string.Format("OrderID:{0} CustomerID:{1} Freight:{2:C}",
        result.OrderID, result.CustomerID, result.Freight);
}

```

In this code sample, the *Load* method of the *XDocument* class is executed to retrieve the XML document from the previous examples, and then a LINQ query is applied to the *XDocument* object. The *select* clause creates an anonymous type containing *OrderID*, *CustomerID*, and *Freight*. The *FirstOrDefault* query extension method is implemented to return a single object of the anonymous type. The result is then displayed in the *txtLog* text box.

In the code sample, *CType* (C# *explicit cast*) converts the attribute to the appropriate type. The *XAttribute* class has 25 explicit conversion operators to convert the *XAttribute* object to a different .NET Framework type. This feature simplifies the syntax of the LINQ query.

Using LINQ to XML with Aggregates

If you want to query for the sum of the freight for each customer, you can use LINQ to XML as shown in the following code sample:

Sample of Visual Basic Code

```

Private Sub LINQQuerySumToolStripMenuItem_Click( _
    ByVal sender As System.Object, ByVal e As System.EventArgs) _
    Handles LINQQuerySumToolStripMenuItem.Click
    Dim doc = XDocument.Load(getFilePath("XDocumentTest.xml"))
    Dim result = From customer In doc.Descendants("Customer")
        Select New With
        {
            .CustomerID = CType(customer.Attribute("CustomerID"), String),
            .TotalFreight = customer.Descendants("Order") _
                .Sum(Function(o) CType(o.Attribute("Freight"), Decimal))
        }
    txtLog.Clear()
    For Each customer In result
        txtLog.AppendText( _
            String.Format("CustomerID:{0} TotalFreight:{1,8:C}" + vbCrLf, _
                customer.CustomerID, customer.TotalFreight))
    Next

```



```

    Next
End Sub

```

Sample of C# Code

```

private void LINQQuerySumToolStripMenuItem_Click(
    object sender, EventArgs e)
{
    var doc = XDocument.Load(getFilePath("XDocumentTest.xml"));
    var result = from customer in doc.Descendants("Customer")
        select new
        {
            CustomerID = (string)customer.Attribute("CustomerID"),
            TotalFreight = customer.Descendants("Order")
                .Sum(o=>(decimal)o.Attribute("Freight"))
        };
    txtLog.Clear();
    foreach (var customer in result)
    {
        txtLog.AppendText( string.Format("CustomerID:{0} TotalFreight:{1,8:C}\r\n",
            customer.CustomerID, customer.TotalFreight));
    }
}

```

The sample code starts by loading the XML file into an *XDocument* object. Next, a LINQ to XML query is created that retrieves *Customer* elements, but the *select* clause creates an anonymous type containing *CustomerID* and *TotalFreight*. *TotalFreight* is calculated by retrieving the *Order* elements of each customer and then executes the *Sum* query extension method by which the *Freight* attribute is converted to a decimal type to perform the aggregation.

Using LINQ to XML Joins

Joins can be accomplished between LINQ to XML and other LINQ providers such as LINQ to Objects. The following code sample starts with an array of orders to be retrieved. This array could have come from a multiple selection list box or some other customer input screen. The order array is then joined to the XML file that was used in the previous examples to retrieve the desired order information.

Sample of Visual Basic Code

```

Private Sub LINQQuerySumToolStripMenuItem_Click( _
    ByVal sender As System.Object, ByVal e As System.EventArgs) _
    Handles LINQQuerySumToolStripMenuItem.Click
    Dim orders() = {"10707", "10835", "10953"}
    Dim doc = XDocument.Load(getFilePath("XDocumentTest.xml"))
    Dim result = From order In doc.Descendants("Order")
        Join selected In orders On
            CType(order.Attribute("OrderID"), String) Equals selected
    Select New With
    {
        .OrderID = CType(order.Attribute("OrderID"), Integer),
        .CustomerID = CType(order.Parent.Attribute("CustomerID"), String),
        .Freight = CType(order.Attribute("Freight"), Decimal)
    }

```

```

    }
    txtLog.Clear()
    For Each order In result
        txtLog.AppendText( _
            String.Format("OrderID:{0} CustomerID:{1} TotalFreight:{2:C}" + vbCrLf, _
                order.OrderID, order.CustomerID, order.Freight))
    Next
End Sub

```

Sample of C# Code

```

private void LINQQueryJoinToolStripMenuItem_Click(
    object sender, EventArgs e)
{
    string[] orders = {"10707", "10835", "10953"};

    var doc = XDocument.Load(getFilePath("XDocumentTest.xml"));
    var result = from order in doc.Descendants("Order")
        join selected in orders
            on (string) order.Attribute("OrderID") equals selected
        select new
        {
            OrderID = (int) order.Attribute("OrderID"),
            CustomerID = (string) order.Parent.Attribute("CustomerID"),
            Freight = (decimal) order.Attribute("Freight")
        };
    txtLog.Clear();
    foreach (var order in result)
    {
        txtLog.AppendText(
            string.Format("OrderID:{0} CustomerID:{1} Freight:{2:C}\r\n",
                order.OrderID, order.CustomerID, order.Freight));
    }
}

```

Using LINQ to XML with Namespaces

LINQ to XML supports the use of namespaces, also known as the namespace URI, in addition to the local name of an XML node. Like .NET Framework namespaces, XML namespaces avoid naming collisions, especially when combining multiple XML documents that might have nodes that have the same name but different meanings. For example, a *Title* element would have a different meaning if it refers to a book and is compared to the *Title* element that refers to a person.

When working with XML namespaces, you can assign a prefix to the namespace. The prefixes can be the source of many problems because prefixes are scoped to their context, so a prefix of *abc* can be associated with namespace *x* in one part of the XML document and can be associated with namespace *y* in a different part of the document.

The following code sample uses an XML document that contains namespace definitions. Three queries are run against it.

Sample of Visual Basic Code

```
Private Sub LINQQueryNamespaceToolStripMenuItem_Click( _
    ByVal sender As System.Object, ByVal e As System.EventArgs) _
    Handles LINQQueryNamespaceToolStripMenuItem.Click
    Dim xml = "<Root xmlns:aw='http://www.adventure-works.com' " & _
        "      xmlns='http://www.xyz.com'> " & _
        "    <Child>1</Child> " & _
        "    <aw:Child>2</aw:Child> " & _
        "    <Child>3</Child> " & _
        "    <aw:Child>4</aw:Child> " & _
        "    <Child>5</Child> " & _
        "    <aw:Child>6</aw:Child> " & _
        "</Root>"
    Dim doc = XDocument.Parse(xml)
    txtLog.Clear()

    Dim result1 = From c In doc.Descendants("Child")
        Select c
    txtLog.AppendText("Query for Child\r\n")
    For Each xElement In result1
        txtLog.AppendText(CType(xElement, String) + vbCrLf)
    Next

    Dim aw = XNamespace.Get("http://www.adventure-works.com")
    Dim result2 = From c In doc.Descendants(aw + "Child")
        Select c
    txtLog.AppendText("Query for aw+Child" + vbCrLf)
    For Each xElement In result2
        txtLog.AppendText(CType(xElement, String) + vbCrLf)
    Next

    Dim defaultns = XNamespace.Get("http://www.xyz.com")
    Dim result3 = From c In doc.Descendants(defaultns + "Child")
        Select c
    txtLog.AppendText("Query for defaultns+Child\r\n")
    For Each xElement In result3

        txtLog.AppendText(CType(xElement, String) + vbCrLf)
    Next

    txtLog.AppendText("Done" + vbCrLf)
End Sub
```

Sample of C# Code

```
private void LINQQueryNamespaceToolStripMenuItem_Click(
    object sender, EventArgs e)
{
    var xml =
        @"<Root xmlns:aw='http://www.adventure-works.com'
          xmlns='http://www.xyz.com'>
          <Child>1</Child>
          <aw:Child>2</aw:Child>
          <Child>3</Child>
          <aw:Child>4</aw:Child>
```

```

        <Child>5</Child>
        <aw:Child>6</aw:Child>
    </Root>";

var doc = XDocument.Parse(xml);
txtLog.Clear();

var result1 = from c in doc.Descendants("Child")
               select c;
txtLog.AppendText("Query for Child\r\n");
foreach (var xElement in result1)
{
    txtLog.AppendText((string)xElement + "\r\n");
}

var aw = XNamespace.Get("http://www.adventure-works.com");
var result2 = from c in doc.Descendants(aw + "Child")
               select c;
txtLog.AppendText("Query for aw+Child\r\n");
foreach (var xElement in result2)
{
    txtLog.AppendText((string)xElement + "\r\n");
}

var defaultns = XNamespace.Get("http://www.xyz.com");
var result3 = from c in doc.Descendants(defaultns + "Child")
               select c;
txtLog.AppendText("Query for defaultns+Child\r\n");
foreach (var xElement in result3)
{
    txtLog.AppendText((string)xElement + "\r\n");
}

txtLog.AppendText("Done\r\n");
}

```

In this code sample, the *XDocument* class is being populated by passing the XML string parameter to the *Parse* method, and the *txtLog* text box is cleared.

Next, the first query that produced *result1* is displayed. This query does not produce any elements because the query is looking for Child elements that are not in any namespace. In this example, all the Child elements are in a namespace; even the Child nodes that have no prefix are in the default namespace (*http://www.xyz.com*).

The second query is for the Child elements in the namespace defined by the *aw* prefix. For this example, an *aw* variable is created and assigned to the *http://www.adventure-works.com* names 2, 4, and 6.

The third query is for the Child elements in the default namespace. For this example, a *defaultns* variable is created and assigned to the *http://www.xyz.com* namespace. You can use the *defaultns* variable by adding the *defaultns* prefix to the local name (Child) by using the plus sign before the local name. This query produces the *result3* variable that enumerates as 1, 3, and 5.

Work with the *XDocument* Class

This practice is a continuation of the practice exercises in Lesson 1. In this practice, you analyze an XML file, called *Orders.xml*, which contains order information. Your first objective is to write a program that can provide the total price of all orders. You also need to provide the total and the average freight cost per order. In this exercise, you use the *XDocument* class and measure its performance by using the *Stopwatch* class.

EXERCISE Implementing the *XDocument* Solution

In this exercise, you extend the Console Application project from Lesson 1 by adding code to retrieve the necessary data by using the *XmlReader* class.

1. In Visual Studio .NET 2010, choose File | Open | Project.
2. Select the project you created in Lesson 1 or open the project in the Begin folder for Lesson 2.
3. In *Main*, after the call to *parseWithXmlReader*, add a *parseWithXDocument* method and pass the file name as a parameter. Add this method to your code. Your code should look like the following:

Sample of Visual Basic Code

```
Sub Main()  
    Dim fileName = "Orders.xml"  
    parseWithXmlDocument(fileName)  
    parseWithXmlReader(fileName)  
    parseWithXDocument(fileName)  
    Console.WriteLine("Press <Enter> to end")  
    Console.ReadLine()  
End Sub  
  
Private Sub parseWithXDocument (ByVal fileName As String)  
  
End Sub
```

Sample of C# Code

```
static void Main(string[] args)  
{  
    string fileName = "Orders.xml";  
    parseWithXmlDocument(fileName);  
    parseWithXmlReader(fileName);  
    parseWithXDocument(fileName);  
    Console.WriteLine("Press <Enter> to end");  
    Console.ReadLine();  
}  
  
private static void parseWithXDocument(string fileName)  
{  
  
}
```

4. In the *parseWithXDocument* method, instantiate a *Stopwatch*, assign the object to a variable, and start it. Declare variables for the total order price, the total freight cost, the average freight cost, and the order count. Add a line of code to load *XDocument* into memory. Add *imports System.Xml.Linq* (C# using *System.Xml.Linq*;) to the top of your file. Your code should look like the following:

Sample of Visual Basic Code

```
Private Sub parseWithXDocument(ByVal fileName As String)
    Dim sw = New Stopwatch()
    sw.Start()
    Dim totalOrderPrice As Decimal = 0
    Dim totalFreightCost As Decimal = 0
    Dim orderQty As Integer = 0
    Dim doc = XDocument.Load(fileName)

End Sub
```

Sample of C# Code

```
private static void parseWithXDocument(string fileName)
{
    var sw = new Stopwatch();
    sw.Start();
    decimal totalOrderPrice = 0;
    decimal totalFreightCost = 0;
    decimal orderQty = 0;
    var doc = XDocument.Load(fileName);
}
```

5. Add a *for each* (C# *foreach*) loop to iterate over all *Order* elements. In the loop, increment the order quantity variable and add a nested *for each* (C# *foreach*) loop to iterate over all the *LineItem* elements of the current *Order*. In the nested loop, add code to retrieve the quantity, price, and freight from the line item. Add the freight to the total freight cost and add the line item cost to the total order price. Your code should look like the following:

Sample of Visual Basic Code

```
For Each order In doc.Descendants("Order")
    orderQty += 1
    For Each lineItem In order.Descendants("LineItem")
        Dim qty = CType(lineItem.Attribute("Qty"), Decimal)
        Dim price = CType(lineItem.Attribute("Price"), Decimal)
        Dim freight = CType(lineItem.Attribute("Freight"), Decimal)
        totalFreightCost += freight
        totalOrderPrice += (qty * price) + freight
    Next
Next
```

Sample of C# Code

```
foreach (var order in doc.Descendants("Order"))
{
    ++orderQty;
```

```

foreach (var lineItem in order.Descendants("LineItem"))
{
    var qty = (decimal)lineItem.Attribute("Qty");
    var price = (decimal)lineItem.Attribute("Price");
    var freight = (decimal)lineItem.Attribute("Freight");
    totalFreightCost += freight;
    totalOrderPrice += (qty * price) + freight;
}
}

```

6. Add code to display the total order price, the total freight cost, and the average freight cost per order. Stop the stopwatch and display the elapsed time. Your completed method should look like the following:

Sample of Visual Basic Code

```

Private Sub parseWithXDocument(ByVal fileName As String)
    Dim sw = New Stopwatch()
    sw.Start()
    Dim totalOrderPrice As Decimal = 0
    Dim totalFreightCost As Decimal = 0
    Dim orderQty As Integer = 0
    Dim doc = XDocument.Load(fileName)

    For Each order In doc.Descendants("Order")
        orderQty += 1
        For Each lineItem In order.Descendants("LineItem")
            Dim qty = CType(lineItem.Attribute("Qty"), Decimal)
            Dim price = CType(lineItem.Attribute("Price"), Decimal)
            Dim freight = CType(lineItem.Attribute("Freight"), Decimal)
            totalFreightCost += freight
            totalOrderPrice += (qty * price) + freight
        Next
    Next

    Console.WriteLine("Total Order Price: {0:C}", totalOrderPrice)
    Console.WriteLine("Total Freight Cost: {0:C}", totalFreightCost)
    Console.WriteLine("Average Freight Cost per Order: {0:C}", _
        totalFreightCost / orderQty)

    sw.Stop()
    Console.WriteLine("Time to Parse XDocument: {0}", sw.Elapsed)
End Sub

```

Sample of C# Code

```

private static void parseWithXDocument(string fileName)
{
    var sw = new Stopwatch();
    sw.Start();
    decimal totalOrderPrice = 0;
    decimal totalFreightCost = 0;
    decimal orderQty = 0;
    var doc = XDocument.Load(fileName);

    foreach (var order in doc.Descendants("Order"))
    {

```

```

        ++orderQty;
        foreach (var lineItem in order.Descendants("LineItem"))
        {
            var qty = (decimal)lineItem.Attribute("Qty");
            var price = (decimal)lineItem.Attribute("Price");
            var freight = (decimal)lineItem.Attribute("Freight");
            totalFreightCost += freight;
            totalOrderPrice += (qty * price) + freight;
        }
    }
    Console.WriteLine("Total Order Price: {0:C}", totalOrderPrice);
    Console.WriteLine("Total Freight Cost: {0:C}", totalFreightCost);
    Console.WriteLine("Average Freight Cost per Order: {0:C}",
        totalFreightCost / orderQty);

    sw.Stop();
    Console.WriteLine("Time to Parse XDocument: {0}", sw.Elapsed);
}

```

7. Run the application. Your total time will vary based on your machine configuration, but your output should look like the following:

```

Result
Total Order Price: $82,989,370.79
Total Freight Cost: $2,011,265.92
Average Freight Cost per Order: $529.84
Time to Parse XmlDocument: 00:00:00.7383706
Total Order Price: $82,989,370.79
Total Freight Cost: $2,011,265.92
Average Freight Cost per Order: $529.84
Time to Parse XmlReader: 00:00:00.3213059
Total Order Price: $82,989,370.79
Total Freight Cost: $2,011,265.92
Average Freight Cost per Order: $529.84
Time to Parse XDocument: 00:00:00.4891875
Press <Enter> to end

```

After coding and running this application, you see the results. The fastest test was *XmlReader*, after which are *XDocument* and *XmlDocument*. Because *XDocument* provides more capabilities using LINQ to XML, you probably want to use *XDocument* in most scenarios except when performance is most important.

Lesson Summary

This lesson provided detailed information about the *XDocument* class family.

- The *XDocument* class provides in-memory, random, read-write access to an XML document.
- The *XDocument* class provides access to the node by using LINQ to XML classes.
- When working with *XAttribute* objects, you can retrieve a typed value by using the *CType* (C# *explicit cast*) statement to convert the attribute value to the desired type.

- When working with *XElement* objects, you can retrieve a typed value by using the *CType* (C# *explicit cast*) statement to convert the element value to the desired type.
- The *XDocument* and *XElement* classes provide *Load* methods for loading from an *XmlReader* file.
- The *XDocument* and *XElement* classes provide a constructor that enables you to pass in *XNode* classes representing the content. The Visual Basic compiler enables you to specify the XML as a string, and it parses the string and generates statements to create the appropriate *XElement* and *XAttribute* objects.
- The *XDocument* and *XElement* classes provide a *Parse* method that enables you to pass in an XML string that will be parsed into the appropriate XML content.

Lesson Review

You can use the following questions to test your knowledge of the information in Lesson 2, “Querying with LINQ to XML.” 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 an XML file, you want to run several queries for data using LINQ to XML. Which class would be most appropriate for these queries on the file?
 - A. *XmlDocument*
 - B. *XmlReader*
 - C. *XDocument*
2. In your code, you have a string variable that contains XML. Which method can you use to convert this into an *XDocument* class so you can run LINQ to XML queries?
 - A. *Load*
 - B. *Constructor*
 - C. *WriteTo*
 - D. *Parse*

Lesson 3: Transforming XML Using LINQ to XML

In addition to performing LINQ queries, another benefit of LINQ to XML is the ability to perform transformations. Prior to LINQ to XML, the best way to transform XML was to use XSLT, but you will certainly find that LINQ to XML is much easier to use than XSLT.

What can you transform XML to? You can transform it to a different form of XML, or to text, or to HTML, or to objects. Best of all is that you have complete control using Visual Basic or C#.

After this lesson, you will be able to:

- Transform XML to objects.
- Transform XML to text.
- Transform XML to XML.

Estimated lesson time: 45 minutes

Transforming XML to Objects

To transform XML to objects, you simply use all the techniques described in this chapter to load and query the XML and the techniques described in Chapter 3 to convert to objects.

The code sample that follows uses a *Customer* class and an *Order* class. These classes are defined as follows:

Sample of Visual Basic Code

```
Public Class Customer
    Public Property Id() As String
    Public Property Name() As String
    Public Property Orders() As List(Of Order)
End Class

Public Class Order
    Public Property Id() As Integer
    Public Property Freight() As Decimal
End Class
```

Sample of C# Code

```
public class Customer
{
    public string Id { get; set; }
    public string Name { get; set; }
    public List<Order> Orders { get; set; }
}

public class Order
{
    public int Id { get; set; }
```

```

    public decimal Freight { get; set; }
}

```

In the following code sample, an XML document is loaded into an *XDocument* object from the *XDocumentTest.xml* file that was saved in the previous lesson examples. This XML document contains a list of customers with their orders, and then a LINQ query is provided to create a generic *IEnumerable* of *Customer*, but, within the *select* statement, a nested LINQ query is populating the *Orders* property of each customer. Finally, the results are displayed.

Sample of Visual Basic Code

```

Private Sub TransformToObjectsToolStripMenuItem_Click( _
    ByVal sender As System.Object, ByVal e As System.EventArgs) _
    Handles TransformToObjectsToolStripMenuItem.Click
    Dim doc = XDocument.Load(getFilePath("XDocumentTest.xml"))

    Dim CustomersOrders = _
        From c In doc.Descendants("Customer")
        Select New Customer With
        {
            .Id = CType(c.Attribute("CustomerID"), String),
            .Name = CType(c.Attribute("CompanyName"), String),
            .Orders = (From o In c.Elements("Order")
                Select New Order With
                {
                    .Id = CType(o.Attribute("OrderID"), Integer),
                    .Freight = CType(o.Attribute("Freight"), Decimal)
                }).ToList()
        }

    txtLog.Clear()
    For Each c In CustomersOrders
        txtLog.AppendText( _
            String.Format("ID:{0} Name:{1}", c.Id, c.Name) + vbCrLf)
        For Each o In c.Orders
            txtLog.AppendText( _
                String.Format("    OrderID:{0} Freight:{1,7:C}", _
                    o.Id, o.Freight) + vbCrLf)
        Next
    Next
End Sub

```

Sample of C# Code

```

private void transformToObjectsToolStripMenuItem_Click(
    object sender, EventArgs e)
{
    var doc = XDocument.Load(getFilePath("XDocumentTest.xml"));

    var CustomersOrders =
        from c in doc.Descendants("Customer")
        select new Customer
        {
            Id = (string)c.Attribute("CustomerID"),
            Name = (string)c.Attribute("CompanyName"),

```

```

        Orders = (from o in c.Elements("Order")
                    select new Order
                    {
                        Id = (int)o.Attribute("OrderID"),
                        Freight = (decimal)o.Attribute("Freight")
                    }).ToList()
    };

    txtLog.Clear();
    foreach (var c in CustomersOrders)
    {
        txtLog.AppendText(
            String.Format("ID:{0} Name:{1}\r\n", c.Id, c.Name));
        foreach (var o in c.Orders)
        {
            txtLog.AppendText(
                String.Format("    OrderID:{0} Freight:{1,7:C}\r\n",
                    o.Id, o.Freight));
        }
    }
}

```

Result

```

ID:ALFKI Name:Alfreds Futterkiste
    OrderID:10643 Freight: $29.46
    OrderID:10692 Freight: $61.02
    OrderID:10702 Freight: $23.94
    OrderID:10835 Freight: $69.53
    OrderID:10952 Freight: $40.42
    OrderID:11011 Freight: $1.21
ID:ANATR Name:Ana Trujillo
    OrderID:10308 Freight: $1.61
    OrderID:10625 Freight: $43.90
    OrderID:10759 Freight: $11.99
    OrderID:10926 Freight: $39.92
ID:ANTON Name:Antonio Moreno
    OrderID:10365 Freight: $22.00
    OrderID:10507 Freight: $47.45
    OrderID:10535 Freight: $15.64
    OrderID:10573 Freight: $84.84
    OrderID:10677 Freight: $4.03
    OrderID:10682 Freight: $36.13
    OrderID:10856 Freight: $58.53
ID:AROUT Name:Around the Horn
    OrderID:10355 Freight: $41.95
    OrderID:10383 Freight: $34.24
    OrderID:10453 Freight: $25.36
    OrderID:10558 Freight: $72.97
    OrderID:10707 Freight: $21.74
    OrderID:10741 Freight: $10.96
    OrderID:10743 Freight: $23.72
    OrderID:10768 Freight: $146.32
    OrderID:10793 Freight: $4.52
    OrderID:10864 Freight: $3.04
    OrderID:10920 Freight: $29.61

```

```
OrderID:10953 Freight: $23.72
OrderID:11016 Freight: $33.80
```

The result shows that the transformation took place. In a real application, you would probably use the *Customer* and *Order* objects for a business purpose rather than simply to display them on the screen.

Transforming XML to Text

In the preceding code sample, if your only goal was to display the results, you could have simply converted the XML to a generic *IEnumerable* of *String* and then displayed the results. The following code sample demonstrates the use of LINQ to XML to convert XML to text.

Sample of Visual Basic Code

```
Private Sub TransformToTextToolStripMenuItem_Click( _
    ByVal sender As System.Object, ByVal e As System.EventArgs) _
    Handles TransformToTextToolStripMenuItem.Click
    Dim doc = XDocument.Load(getFilePath("XDocumentTest.xml"))

    Dim CustomersOrders = _
        From c In doc.Descendants("Customer")
        Select New With
        {
            .CustomerInfo = _
                String.Format("ID:{0} Name:{1}" + vbCrLf, _
                    CType(c.Attribute("CustomerID"), String),
                    CType(c.Attribute("CompanyName"), String)
                ),
            .OrderInfo = From o In c.Elements("Order")
                Select String.Format(
                    "    OrderID:{0} Freight:{1,7:C}" + vbCrLf, _
                    CType(o.Attribute("OrderID"), Integer),
                    CType(o.Attribute("Freight"), Decimal))
        }

    txtLog.Clear()
    For Each c In CustomersOrders
        txtLog.AppendText(c.CustomerInfo)
        For Each o In c.OrderInfo
            txtLog.AppendText(o)
        Next
    Next
End Sub
```

Sample of C# Code

```
private void transformToTextToolStripMenuItem_Click(
    object sender, EventArgs e)
{
    var doc = XDocument.Load(getFilePath("XDocumentTest.xml"));
    var customersOrders =
        from c in doc.Descendants("Customer")
        select new
        {

```

```

        CustomerInfo = string.Format(
            "ID:{0} Name:{1}\r\n",
            c.Attribute("CustomerID"),
            c.Attribute("Name")),
        OrderInfo = from o in c.Elements("Order")
                    select string.Format(
                        "    OrderID:{0} Freight:{1,7:C}\r\n",
                        (int)o.Attribute("OrderID"),
                        (decimal)o.Attribute("Freight"))
    };
    foreach (var c in customersOrders)
    {
        txtLog.AppendText(c.CustomerInfo);
        foreach (var o in c.OrderInfo)
        {
            txtLog.AppendText(o);
        }
    }
}

```

This code sample produces the same results as the preceding one. The difference is that the string formatting is in the LINQ queries, and instead of creating a *Customer* and an *Order* class, an anonymous type holds the result as strings, so the nested loop displaying the result is much simpler.

Transforming XML to XML

You can use LINQ to XML to transform an XML document in one format into an XML document in a different format. Visual Basic really shines with its use of XML literals, to which you were introduced in Lesson 1. Visual Basic implementation of XML literals can simplify XML to XML transformations. C# users who want to use XML literals must learn Visual Basic, but don't throw away your C# book. Instead, you can do all your XML literals in a separate Visual Basic DLL project and then set a reference to the DLL in your C# code.

In the following sample, the XML document that contains customers and their orders is transformed to an XML document that has the same information but is formatted differently.

Sample of Visual Basic Code

```

Private Sub TransformXMLToXMLToolStripMenuItem_Click( _
    ByVal sender As System.Object, ByVal e As System.EventArgs) _
    Handles TransformXMLToXMLToolStripMenuItem.Click
    Dim doc = XDocument.Load(getFilePath("XDocumentTest.xml"))
    Dim newXml = _
        <root>
            <%= From o In doc...<Order> _
                Select <Order CustID=<%= o.Parent.@CustomerID %>
                    CustName=<%= o.Parent.@CompanyName %>
                    OrdID=<%= o.@OrderID %>
                    OrdFreight=<%= o.@Freight %>/>

            %>
        </root>

```

```

        txtLog.Text = newXml.ToString()
    End Sub

```

Sample of C# Code

```

private void transformXMLToXMLToolStripMenuItem_Click(
    object sender, EventArgs e)
{
    var doc = XDocument.Load(getFilePath("XDocumentTest.xml"));
    var newXml = new XDocument(
        new XElement("root",
            from o in doc.Descendants("Order")
            select new XElement("Order",
                new XAttribute("CustID", o.Parent.Attribute("CustomerID").Value),
                new XAttribute("CustName", o.Parent.Attribute("CompanyName").Value),
                new XAttribute("OrdID", o.Attribute("OrderID").Value),
                new XAttribute("OrdFreight", o.Attribute("Freight").Value)
            )
        )
    );
    txtLog.Text = newXml.ToString();
}

```

Result

```

<root>
  <Order CustID="ALFKI" CustName="Alfreds Futterkiste" OrdID="10643"
OrdFreight="29.4600" />
  <Order CustID="ALFKI" CustName="Alfreds Futterkiste" OrdID="10692"
OrdFreight="61.0200" />
  <Order CustID="ALFKI" CustName="Alfreds Futterkiste" OrdID="10702"
OrdFreight="23.9400" />
  <Order CustID="ALFKI" CustName="Alfreds Futterkiste" OrdID="10835"
OrdFreight="69.5300" />
  <Order CustID="ALFKI" CustName="Alfreds Futterkiste" OrdID="10952"
OrdFreight="40.4200" />
  <Order CustID="ALFKI" CustName="Alfreds Futterkiste" OrdID="11011" OrdFreight="1.2100"
/>
  <Order CustID="ANATR" CustName="Ana Trujillo" OrdID="10308" OrdFreight="1.6100" />
  <Order CustID="ANATR" CustName="Ana Trujillo" OrdID="10625" OrdFreight="43.9000" />
  <Order CustID="ANATR" CustName="Ana Trujillo" OrdID="10759" OrdFreight="11.9900" />
  <Order CustID="ANATR" CustName="Ana Trujillo" OrdID="10926" OrdFreight="39.9200" />
  <Order CustID="ANTON" CustName="Antonio Moreno" OrdID="10365" OrdFreight="22.0000" />
  <Order CustID="ANTON" CustName="Antonio Moreno" OrdID="10507" OrdFreight="47.4500" />
  <Order CustID="ANTON" CustName="Antonio Moreno" OrdID="10535" OrdFreight="15.6400" />
  <Order CustID="ANTON" CustName="Antonio Moreno" OrdID="10573" OrdFreight="84.8400" />
  <Order CustID="ANTON" CustName="Antonio Moreno" OrdID="10677" OrdFreight="4.0300" />
  <Order CustID="ANTON" CustName="Antonio Moreno" OrdID="10682" OrdFreight="36.1300" />
  <Order CustID="ANTON" CustName="Antonio Moreno" OrdID="10856" OrdFreight="58.5300" />
  <Order CustID="AROUT" CustName="Around the Horn" OrdID="10355" OrdFreight="41.9500" />
  <Order CustID="AROUT" CustName="Around the Horn" OrdID="10383" OrdFreight="34.2400" />
  <Order CustID="AROUT" CustName="Around the Horn" OrdID="10453" OrdFreight="25.3600" />
  <Order CustID="AROUT" CustName="Around the Horn" OrdID="10558" OrdFreight="72.9700" />
  <Order CustID="AROUT" CustName="Around the Horn" OrdID="10707" OrdFreight="21.7400" />
  <Order CustID="AROUT" CustName="Around the Horn" OrdID="10741" OrdFreight="10.9600" />

```

```

    <Order CustID="AROUT" CustName="Around the Horn" OrdID="10743" OrdFreight="23.7200" />
    <Order CustID="AROUT" CustName="Around the Horn" OrdID="10768" OrdFreight="146.3200"
  />
  <Order CustID="AROUT" CustName="Around the Horn" OrdID="10793" OrdFreight="4.5200" />
  <Order CustID="AROUT" CustName="Around the Horn" OrdID="10864" OrdFreight="3.0400" />
  <Order CustID="AROUT" CustName="Around the Horn" OrdID="10920" OrdFreight="29.6100" />
  <Order CustID="AROUT" CustName="Around the Horn" OrdID="10953" OrdFreight="23.7200" />
  <Order CustID="AROUT" CustName="Around the Horn" OrdID="11016" OrdFreight="33.8000" />
</root>

```

Both of these language samples produce the same result, but the C# version looks more obfuscated than the Visual Basic example. XML literals enable the Visual Basic programmer to simply embed the elements as literals in the code and use `<%= expression block %>` syntax in the code. This is very similar to expression blocks in ASP or ASP.NET applications. When referencing XML axes for navigation, Visual Basic enables you to use three dots instead of typing *Descendants*. You can also use dot notation to access a child element. This example code uses the dot notation to access an attribute by prefixing the attribute name with the `@` symbol.

Use LINQ to XML to Transform Data

In this practice, you transform an `Orders.xml` XML file, which contains order information, into a different format that management can use. Here is an example of what the file looks like:

Orders.xml File

```

<Orders>
  <Order OrderNumber="S043659">
    <LineItem Line="1" PID="349" Qty="1" Price="2024.9940" Freight="50.6249" />
    <LineItem Line="2" PID="350" Qty="3" Price="2024.9940" Freight="151.8746" />
    <LineItem Line="3" PID="351" Qty="1" Price="2024.9940" Freight="50.6249" />
    <LineItem Line="4" PID="344" Qty="1" Price="2039.9940" Freight="50.9999" />
    <LineItem Line="5" PID="345" Qty="1" Price="2039.9940" Freight="50.9999" />
    <LineItem Line="6" PID="346" Qty="2" Price="2039.9940" Freight="101.9997" />
    <LineItem Line="7" PID="347" Qty="1" Price="2039.9940" Freight="50.9999" />
    <LineItem Line="8" PID="229" Qty="3" Price="28.8404" Freight="2.1630" />
    <LineItem Line="9" PID="235" Qty="1" Price="28.8404" Freight="0.7210" />
    <LineItem Line="10" PID="218" Qty="6" Price="5.7000" Freight="0.8550" />
    <LineItem Line="11" PID="223" Qty="2" Price="5.1865" Freight="0.2593" />
    <LineItem Line="12" PID="220" Qty="4" Price="20.1865" Freight="2.0187" />
  </Order>
  <Order OrderNumber="S043660">
    <LineItem Line="1" PID="326" Qty="1" Price="419.4589" Freight="10.4865" />
    <LineItem Line="2" PID="319" Qty="1" Price="874.7940" Freight="21.8699" />
  </Order>
<!--Many more orders here -->
</Orders>

```

The resulting XML document is saved to a `Results.xml` file, and each *Order* element contains a *TotalFreight* attribute. The *LineItem* element is now called *Item*, the *Line* attribute is now called *Number*, the *PID* attribute is now called *ID*, and a *LineTotal* attribute has been added to each line item. The following is a sample of the `Results.xml` file.

Results.xml File

```
<ModifiedOrders>
  <Order OrderID="S043659" TotalFreight="514.1408">
    <Item Number="1" ID="349" Price="2024.9940" Qty="1" LineTotal="2024.994" />
    <Item Number="2" ID="350" Price="2024.9940" Qty="3" LineTotal="6074.982" />
    <Item Number="3" ID="351" Price="2024.9940" Qty="1" LineTotal="2024.994" />
    <Item Number="4" ID="344" Price="2039.9940" Qty="1" LineTotal="2039.994" />
    <Item Number="5" ID="345" Price="2039.9940" Qty="1" LineTotal="2039.994" />
    <Item Number="6" ID="346" Price="2039.9940" Qty="2" LineTotal="4079.988" />
    <Item Number="7" ID="347" Price="2039.9940" Qty="1" LineTotal="2039.994" />
    <Item Number="8" ID="229" Price="28.8404" Qty="3" LineTotal="86.5212" />
    <Item Number="9" ID="235" Price="28.8404" Qty="1" LineTotal="28.8404" />
    <Item Number="10" ID="218" Price="5.7000" Qty="6" LineTotal="34.2" />
    <Item Number="11" ID="223" Price="5.1865" Qty="2" LineTotal="10.373" />
    <Item Number="12" ID="220" Price="20.1865" Qty="4" LineTotal="80.746" />
  </Order>
  <Order OrderID="S043660" TotalFreight="32.3564">
    <Item Number="1" ID="326" Price="419.4589" Qty="1" LineTotal="419.4589" />
    <Item Number="2" ID="319" Price="874.7940" Qty="1" LineTotal="874.794" />
  </Order>
<!--Many more orders here -->
</ ModifiedOrders >
```

This project will be implemented as a simple Console application.

EXERCISE Creating the Project and Implementing the Transformation

In this exercise, you create a Console Application project and then add code to the transformed *XDocument* class and save to the Result.xml file.

1. In Visual Studio .NET 2010, choose File | New | Project.
2. Select your desired programming language and then select the Console Application template. For the project name, enter **OrderTransformer**. Be sure to select a desired location for this project. For the solution name, enter **OrderTransformerSolution**. Be sure that Create Directory For Solution is selected and then click OK. After Visual Studio .NET finishes creating the project, Module1.vb (C# Program.cs) is displayed.
3. In *Main*, declare a *string* for your file name and assign Orders.xml to it. Add code to the bottom of *Main* to prompt the user to press Enter to end the application. Your code should look like the following:

Sample of Visual Basic Code

```
Module Module1
  Sub Main()
    Dim fileName = "Orders.xml"

    Console.WriteLine("Press <Enter> to end")
    Console.ReadLine()
  End Sub
End Module
```

Sample of C# Code

```
namespace OrderTransformer
```

```

{
    class Program
    {
        static void Main(string[] args)
        {
            string fileName = "Orders.xml";

            Console.WriteLine("Press <Enter> to end");
            Console.ReadLine();
        }
    }
}

```

4. Add the Orders.xml file to your project. Right-click the *OrderTransformer* node in Solution Explorer and choose Add | Existing Item. In the bottom right corner of the Add Existing Item dialog box, click the drop-down list and select All Files (*.*). Navigate to the Begin folder for this exercise and select Orders.xml. If you don't see the Orders.xml file, check whether All Files (*.*) has been selected.
5. In Solution Explorer, click the Orders.xml file you just added. In the Properties window, set the *Copy to Output Directory* property to *Copy If Newer*.
Because this file will reside in the same folder as your application, you will be able to use the file name without specifying a path.
6. In the *Main* method, add code to load the Orders.xml file into an *XDocument* object and assign this object to a *doc* variable. For C#, add *using System.Xml.Linq;* to the top of your file. Your code should look like the following:

Sample of Visual Basic Code

```

Module Module1
    Sub Main()
        Dim fileName = "Orders.xml"
        Dim doc = XDocument.Load(fileName)

        Console.WriteLine("Press <Enter> to end")
        Console.ReadLine()
    End Sub
End Module

```

Sample of C# Code

```

namespace OrderTransformer
{
    class Program
    {
        static void Main(string[] args)
        {
            string fileName = "Orders.xml";
            var doc = XDocument.Load(fileName);

            Console.WriteLine("Press <Enter> to end");
            Console.ReadLine();
        }
    }
}

```

```

    }
}

```

7. Declare a *result* variable and assign your LINQ to XML transformation to it. You need to iterate the *Order* elements to complete the transformation. If you're using Visual Basic, be sure to take advantage of XML literals. Your code should look like the following:

Sample of Visual Basic Code

```

Dim result = _
    <ModifiedOrders>
        <%= From o In doc...<Order> _
            Select <Order OrderID=<%= o.@OrderNumber %>
                TotalFreight=<%= o.<LineItem>.Sum(
                    Function(li) CType(li.@Freight, Decimal)) %>>
            <%= From li In o.<LineItem>
                Select <Item Number=<%= li.@Line %>
                    ID=<%= li.@PID %>
                    Price=<%= li.@Price %>
                    Qty=<%= li.@Qty %>
                    LineTotal=<%= li.@Price * li.@Qty %>/>
                %>
            </Order>
        %>
    </ModifiedOrders>

```

Sample of C# Code

```

var result = new XElement("ModifiedOrders",
    from o in doc.Descendants("Order")
    select new XElement("Order",
        new XAttribute("OrderID", (string)o.Attribute("OrderNumber")),
        new XAttribute("TotalFreight",
            o.Elements("LineItem").Sum(li=>(decimal) li.Attribute("Freight"))),
        from li in o.Elements("LineItem")
        select new XElement("Item",
            new XAttribute("Number", (int)li.Attribute("Line")),
            new XAttribute("ID", (int)li.Attribute("PID")),
            new XAttribute("Price", (decimal)li.Attribute("Price")),
            new XAttribute("Qty", (int)li.Attribute("Qty")),
            new XAttribute("LineTotal",
                (decimal)li.Attribute("Price") *
                (int)li.Attribute("Qty"))
            )
        )
    );

```

8. Add code to save the result to a Results.xml file on your desktop or to a location of your choice. Add *imports System.IO* (C# using *System.IO*;) to the top of your file. The completed *Main* method should look like the following.

Sample of Visual Basic Code

```

Sub Main()
    Dim fileName = "Orders.xml"
    Dim doc = XDocument.Load(fileName)

```

```

Dim result = _
    <ModifiedOrders>
        <%= From o In doc...<Order> _
            Select <Order OrderID=<%= o.@OrderNumber %>
                TotalFreight=<%= o.<LineItem>.Sum(
                    Function(li) CType(li.@Freight, Decimal)) %>>
            <%= From li In o.<LineItem>
                Select <Item Number=<%= li.@Line %>
                    ID=<%= li.@PID %>
                    Price=<%= li.@Price %>
                    Qty=<%= li.@Qty %>
                    LineTotal=<%= li.@Price * li.@Qty %>/>
                %>
            </Order>
        %>
    </ModifiedOrders>
result.Save(Path.Combine(Environment.GetFolderPath( _
    Environment.SpecialFolder.Desktop), "Results.xml"))
Console.Write("Press <Enter> to end")
Console.ReadLine()
End Sub

```

Sample of C# Code

```

static void Main(string[] args)
{
    string fileName = "Orders.xml";
    var doc = XDocument.Load(fileName);
    var result = new XElement("ModifiedOrders",
        from o in doc.Descendants("Order")
        select new XElement("Order",
            new XAttribute("OrderID", (string)o.Attribute("OrderNumber")),
            new XAttribute("TotalFreight",
                o.Elements("LineItem").Sum(li=>(decimal) li.Attribute("Freight"))),
            from li in o.Elements("LineItem")
            select new XElement("Item",
                new XAttribute("Number", (int)li.Attribute("Line")),
                new XAttribute("ID", (int)li.Attribute("PID")),
                new XAttribute("Price", (decimal)li.Attribute("Price")),
                new XAttribute("Qty", (int)li.Attribute("Qty")),
                new XAttribute("LineTotal",
                    (decimal)li.Attribute("Price") *
                    (int)li.Attribute("Qty"))
            )
        )
    );
    result.Save(Path.Combine(Environment.GetFolderPath(
        Environment.SpecialFolder.Desktop), "Results.xml"));

    Console.Write("Press <Enter> to end");
    Console.ReadLine();
}

```

9. Run the application. You should see a message stating, "Press <Enter> to end." Press Enter to end the application. After running the application, locate the Results.xml file and open it. Your file should look like the following:

Results.xml

```
<?xml version="1.0" encoding="utf-8"?>
<ModifiedOrders>
  <Order OrderID="S043659" TotalFreight="514.1408">
    <Item Number="1" ID="349" Price="2024.9940" Qty="1" LineTotal="2024.9940" />
    <Item Number="2" ID="350" Price="2024.9940" Qty="3" LineTotal="6074.9820" />
    <Item Number="3" ID="351" Price="2024.9940" Qty="1" LineTotal="2024.9940" />
    <Item Number="4" ID="344" Price="2039.9940" Qty="1" LineTotal="2039.9940" />
    <Item Number="5" ID="345" Price="2039.9940" Qty="1" LineTotal="2039.9940" />
    <Item Number="6" ID="346" Price="2039.9940" Qty="2" LineTotal="4079.9880" />
    <Item Number="7" ID="347" Price="2039.9940" Qty="1" LineTotal="2039.9940" />
    <Item Number="8" ID="229" Price="28.8404" Qty="3" LineTotal="86.5212" />
    <Item Number="9" ID="235" Price="28.8404" Qty="1" LineTotal="28.8404" />
    <Item Number="10" ID="218" Price="5.7000" Qty="6" LineTotal="34.2000" />
    <Item Number="11" ID="223" Price="5.1865" Qty="2" LineTotal="10.3730" />
    <Item Number="12" ID="220" Price="20.1865" Qty="4" LineTotal="80.7460" />
  </Order>
  <Order OrderID="S043660" TotalFreight="32.3564">
    <Item Number="1" ID="326" Price="419.4589" Qty="1" LineTotal="419.4589" />
    <Item Number="2" ID="319" Price="874.7940" Qty="1" LineTotal="874.7940" />
  </Order>
<!-- more orders here-->
</ModifiedOrders>
```

Lesson Summary

This lesson provided detailed information about transforming data by using LINQ to SQL.

- You can take advantage of LINQ projections to convert XML to objects.
- LINQ projections can also help convert XML to a generic *IEnumerable* of *string*.
- LINQ projections can also be used with the *XDocument* classes to convert XML to a different form of XML.
- If you use Visual Basic, you can take advantage of XML literals to simplify transformations.

Lesson Review

You can use the following questions to test your knowledge of the information in Lesson 3, "Transforming XML Using LINQ to XML." 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. You have an XML file you want to transform to a different form of XML, and you want to use XML literals to simplify the transformation. Which language will you use?
 - A. C#
 - B. Visual Basic
 - C. Either

Case Scenario

In the following case scenario, you will apply what you've learned in this chapter about querying data by using LINQ to XML. You can find answers to these questions in the "Answers" section at the end of this book.

Case Scenario: XML Web Service

You have been given the job of making a call to an XML web service and parsing the response. Your application is very object oriented, so you'll need to convert objects to an XML request, and, when you receive the XML response, you will need to parse the response back to business objects that you can use in the application. Answer the following questions regarding the implementation of the XML web service.

1. What approach will you take to create the XML web service request?
2. What approach will you take to parse the XML web service response?

Suggested Practices

To help you successfully master the exam objectives presented in this chapter, complete the following tasks. You should create at least one application that uses LINQ to XML for querying data. You can do this by completing the practices at the ends of Lessons 1 and 2 or by performing the following Practice 1.

- **Practice 1** Create an application that queries XML data. This could be an application that reads RSS feeds or XML from a web service.
- **Practice 2** Complete Practice 1 and then add code to produce a transformation of the XML to a different type.

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.



CHAPTER 6

ADO.NET Entity Framework

In the previous chapters, you saw various aspects of Language Integrated Query (LINQ) for querying and transforming data. This chapter covers the most sophisticated and flexible implementation of LINQ, LINQ to Entities.

Exam objectives in this chapter:

- Map entities and relationships by using the Entity Data Model.
- Create and customize entity objects.
- Connect a POCO model to the Entity Framework.
- Create the database from the Entity Framework model.
- Create model-defined functions.
- Manage the *DataContext* and *ObjectContext*.
- Implement eager loading.
- Create, update, or delete data by using *ObjectContext*.
- Create an Entity SQL (ESQL) query.
- Manage transactions.

Lessons in this chapter:

- Lesson 1: What Is the ADO.NET Entity Framework? **361**
- Lesson 2: Querying and Updating with the Entity Framework **421**

Before You Begin

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



REAL WORLD

Glenn Johnson

When building object-oriented applications, the creation of business objects seems to go smoothly until it's time to persist the data to a database. I've been involved in applications that use data sets to persist the data. I've also used LINQ to SQL and various third-party object-relational mapping (ORM) products. There are always problems, but LINQ to Entities minimizes the ORM pain.

Lesson 1: What Is the ADO.NET Entity Framework?

When writing object-oriented applications, you want to think of the problem domain and write objects and code that are domain-centric. Writing data access code and creating data access objects that can talk to the database feel like distractions; they represent noise in your application. However, you need some means of persisting your domain objects to the database, which usually means creating a data model that represents your relational database. The Entity Framework can provide this.

After this lesson, you will be able to:

- Understand and explain the ADO.NET Entity Framework.
- Explain the differences between LINQ to SQL and the ADO.NET Entity Framework.
- Model data using the ADO.NET Entity Framework.
- Implement the code first model.
- Implement the database first model.
- Retrieve data using the *ObjectContext* class.
- Understand how the *ObjectContext* object connects to the database.
- Comprehend the life cycle of an entity object.
- Explain the difference between lazy loading, explicit loading, and eager loading.
- Work with complex types.
- Map stored procedures.
- Implement Inheritance in the ADO.NET Entity Framework.
- Use plain old CLR objects with the Entity Framework.

Estimated lesson time: 60 minutes

Entity Framework Architecture Overview

The Entity Framework enables applications to access and change data that is represented as entities and relationships in the conceptual model. The Entity Framework translates object queries against entity types described in the conceptual model into database-specific queries by using information in the model and mapping files. When you execute a query, the results are materialized into objects managed by the Entity Framework. The diagram in Figure 6-1 shows the Entity Framework.

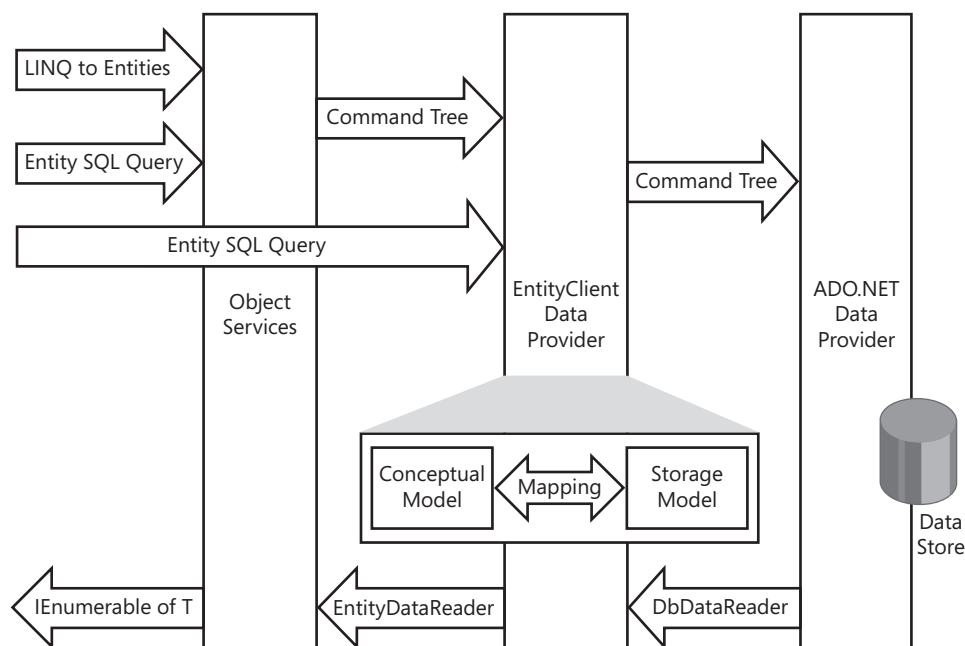


FIGURE 6-1 The Entity Framework diagram translates object queries into relational queries

The diagram shows that you can query a conceptual model and return objects by using LINQ to Entities or Entity SQL (ESQL).

LINQ to Entities provides LINQ support for querying entity types that are defined in a conceptual model.

ESQL is a storage-independent dialect of SQL that works directly with entities in the conceptual model. It can be used with object queries and queries that are executed by using the EntityClient provider.

The Object Services layer is a component of the Entity Framework that enables you to query, insert, update, and delete data, using common language runtime (CLR) objects that are instances of entity types. Object Services supports LINQ and ESQL queries against types that are defined in the conceptual model, is responsible for instantiating the returned data as objects, and propagates object changes back to the data source. Object Services provides change tracking and concurrency handling. The Object Services layer is implemented by classes in the *System.Data.Objects* and *System.Data.Objects.DataClasses* namespaces.

The Entity Framework includes the EntityClient data provider, which manages connections, translates entity queries into data source-specific queries, and returns a data reader the Entity Framework uses to materialize entity data into objects.

Traditional ADO.NET is still used to communicate to the underlying database. This is why it's still helpful to know this technology and the classes that were covered in Chapter 1, "ADO.NET Disconnected Classes," and Chapter 2, "ADO.NET Connected Classes."

Entity Framework vs. LINQ to SQL

In Chapter 4, “LINQ to SQL,” you learned that LINQ to SQL provides a data model that simplifies persistence of your domain model to the relational database. So what’s the difference between the Entity Framework and LINQ to SQL? This is a common question, and Table 6-1 provides a quick reference to the primary differences between the two technologies.

TABLE 6-1 A Comparison of LINQ to SQL and Entity Framework

CATEGORY	LINQ TO SQL	ENTITY FRAMEWORK
Complexity	Less complex	More complex
Model	Domain model	Conceptual data model
DB Server	SQL Server	Variety of database products
Development Time	Rapid application development	More time required but has more features
Mapping Type	Class to single table	Class to multiple tables
Inheritance	Difficult to apply	Simple to apply
File Types	DBML files	EDMX, CDSL, MSL, SSDL files
Complex Type Support	No	Yes
Query Capability	LINQ to SQL through <i>DataContext</i>	LINQ to Entities, ESQL, Object Services, Entity Client
Performance	Slow for first query	Slow for first query but overall better than LINQ to SQL
Future Enhancements	No	Yes
Generate DB from Model	No	Yes

Following is a more detailed look at each of the categories:

- **Complexity** With more features comes more complexity, which is why LINQ to SQL, which has fewer features, is considered easier to use than the Entity Framework, which has more features.
- **Model** LINQ to SQL provides a one-to-one mapping of tables to classes. If you have *Customers*, *Orders*, and *LineItems* tables, you will have a *Customer*, *Order*, and *LineItem* class to match up with rows of each table. The Entity Framework enables you to have a *Customer* class whose data maps to several tables. This means the company name can be in one table, but the address can be in a different table, and the phone number can be in another table, and so on.
- **DB Server** LINQ to SQL supports only Microsoft SQL Server 2000 and later, but even SQL Server 2000 support has some limitations. The Entity Framework has support for IBM DB2, Sybase SqlAnywhere, Oracle, SQL Azure, and many others.

- **Development Time** LINQ to SQL is simple to learn and implement for rapid application development, but LINQ to SQL has limitations that can cause problems in complex applications. The Entity Framework has more capabilities, which can take longer to learn and implement, but these features will minimize problems when creating complex applications.
- **Mapping Type** With LINQ to SQL, each table maps to a single class. If your database has a join table, you must represent this as a class. Also, complex types cannot be easily represented without creating a separate table. For example, if a customer has an address and the Customers table has the address components in the table, you can't represent the address as a separate type. To represent the address as a separate type, you must extract the address to its own table. With the Entity Framework, a class can map to multiple tables. Regarding the address scenario, the address can be a type even if it's contained within the Customers table.
- **Inheritance** LINQ to SQL supports Table per Class Hierarchy (TPH), whereas the Entity Framework supports TPH and Table per Type (TPT). The Entity Framework also provides limited support for Table per Concrete Class (TPC).
- **File Type** LINQ to SQL uses a Database Markup Language (DBML extension) file that contains XML mappings of entities to tables. The Entity Framework uses four files. The first file is the Entity Data Model (EDMX extension), which the Entity Data Model designer uses. At compile time, the other three files are created from the EDMX file. The first of the three files is a Conceptual Schema Definition Language (CSDL extension) file that contains XML definition of the conceptual model. The second file is the Store Schema Definition Language (SSDL) file that contains XML definition of the storage model. The third file is the Mapping Specification Language (MSL extension) file that contains the mappings between the conceptual and storage models.
- **Complex Type Support** A complex type is a nonscalar property of an entity type that does not have a key property. For example, a customer has a phone number, but you want the phone number to be defined as having a country code, an area code, a city code, a number, and an extension. LINQ to SQL doesn't support the creation of complex types, but the Entity Framework does.
- **Query Capability** You can query the database by using LINQ to SQL through the *DataContext* object. With the Entity Framework, you can query the database by using LINQ to Entities through the *ObjectContext* object. The Entity Framework also provides ESQL, which is a SQL-like query language that is good for defining a query as part of the model definition. The Entity Framework also contains the *ObjectQuery* class, used with Object Services for dynamically constructing queries at run time. Last, the Entity Framework contains the *EntityClient* provider, which runs queries against the conceptual model.
- **Performance** Both LINQ to SQL and the Entity Framework are slow when running the first query, but, after that, you should find that both provide acceptable performance, although the Entity Framework provides slightly better performance.

- **Future Enhancements** Microsoft intended to obsolete LINQ to SQL after the Entity Framework release, but LINQ to SQL was popular due to its simplicity, so Microsoft responded to user feedback. You can expect that LINQ to SQL will not receive future enhancements, however.
- **Generate Database from Model** LINQ to SQL has no capability of generating the database from the model. The Entity Framework supports two types of development, Database First and Code First. With Database First development, the database already exists, so there is no need to generate the database from the model. With Code First development, you create your model, and, from the model, you can generate your database.

Modeling Data

To use the Entity Framework, you must create an entity data model that defines your model classes and their mapping into the database schema. After the model is created, you can perform CRUD operations (create, retrieve, update, and delete) using LINQ to Entities and Object Services.

Mapping Scenarios

To create an entity data model, you must have a basic understanding of the mapping scenarios supported by the Entity Framework. The following is a description of each scenario supported by the Entity Framework:

- **Simple mapping** Each entity in the conceptual model is mapped to a single table in the storage model. This is the Entity Data Model default mapping.
- **Entity splitting** Properties from a single entity in the conceptual model are mapped to columns in two or more underlying tables that share a common primary key.
- **Horizontal partitioning in the conceptual model** Multiple entity types in the conceptual model that have the same properties are mapped to a single table. A condition clause specifies which data in the table belongs to which entity type. This mapping is similar to TPH inheritance mapping.
- **TPH inheritance** All types in an inheritance hierarchy are mapped to a single table. A condition clause defines the entity types.
- **TPT inheritance** All types are mapped to individual tables. Properties that belong solely to a base type or a derived type are stored in a table that maps to that type.
- **TPC inheritance** Nonabstract types are each mapped to an individual table. Each of these tables must have columns that map to all the properties of the derived type, including the properties inherited from the base type.
- **Multiple entity sets per type** A single entity type is expressed in two or more entity sets in the conceptual model. Each entity set is mapped to a separate table in the storage model.

- **Complex types** A complex type is a nonscalar property of an entity type that does not have a key property. A complex type can contain other nested complex types. Complex types are mapped to tables in the storage model.
- **Function import mapping** A stored procedure in the storage model is mapped to a *FunctionImport* element in the conceptual model. This function is executed to return entity data, using the mapped stored procedure.
- **Modification function mapping** Stored procedures are defined in the storage model to insert, update, and delete data. These functions are defined for an entity type to provide the update functionality for a specific entity type.
- **Defining query mapping** A query is defined in the storage model that represents a table in the data source. The query is expressed in the native query language of the data source, such as Transact-SQL when mapping to a SQL Server database. This *DefiningQuery* element is mapped to an entity type in the conceptual model. The query is defined in the store-specific query language.
- **Query view mapping** A read-only mapping is defined between entity types in the conceptual model and relational tables in the storage model. This mapping is defined based on an ESQL query against the storage model that returns entities in the conceptual model. When you use a query view, updates cannot be persisted to the data source by using the standard update process. Updates can be made by defining modification function mappings.
- **AssociationSet mapping** Associations define relationships between entities. In a simple mapping with a one-to-one or one-to-many association, associations that define relationships in the conceptual model are mapped to associations in the storage model.
- **Many-to-many associations** This is a special AssociationSet mapping by which both ends of the association are mapped to a link table in the storage model.
- **Self association** A special AssociationSet mapping that supports an association between two entities of the same type, such as an *Employee* entity with an association to another *Employee* entity.

Code First Model vs. Database First Model

What is the Code First model? This is when you create your conceptual model before you create the database. Using the Code First model, you can generate the database from the conceptual model, but first you must create your conceptual model manually. The Database First model enables you to generate the conceptual model from the database schema, but first you must create the database schema manually.

Which model should you use? If the database or the conceptual model already exists, you will surely use the associated model. If nothing exists, simply take the path of least resistance and start working on the end with which you are most comfortable. It's that simple.

Implementing the Code First Model

To use the Code First model, add an ADO.NET Entity Data Model.edmx file to your project: Right-click the project node in Solution Explorer and choose Add | New Item. Select ADO.NET Entity Data Model. This template produces a file with an EDMX extension. Provide a name for your file. This example demonstrates the creation of a conceptual model for maintaining a list of vehicles with their associated repairs. After naming the file, click Add to start the Entity Data Model Wizard.

Figure 6-2 shows the first step in the Entity Data Model Wizard, which is to select whether you want to generate the model from the database (Database First) or start with an empty model (Code First). Select Empty Model and click Finish. You then see the empty Entity Data Model designer screen.

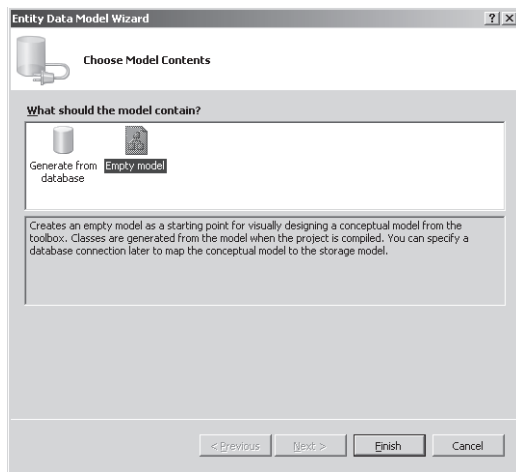


FIGURE 6-2 The first wizard step enables you to choose Database First or Code First.

Use the toolbox to drag an entity out and drop it on the Entity Data Model designer, as shown in Figure 6-3. When you drop the entity on the designer surface, the entity name will be *Entity1*, but you can click the name and type a new name. In Figure 6-3, the entity was renamed to *Vehicle*. The entity automatically gets an Id column.

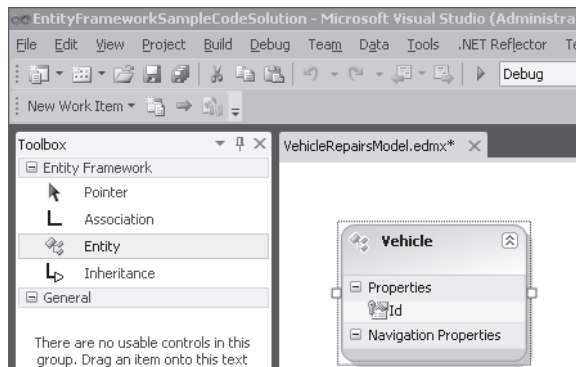


FIGURE 6-3 Dragging an entity from the toolbox to the designer surface.

Clicking the entity causes its properties to be displayed in the Properties window. The following is a list of entity properties.

- **Abstract** Set to *true* if this entity will be a *MustInherit* (C# *abstract*) class. You set this to *true* when the entity will have derived entities, and you never want this entity to be instantiated directly. For example, you might have a *Pet* entity and *Cat* and *Dog* entities that inherit from *Pet*. You never want someone to instantiate *Pet* directly because you won't know what kind of pet this object represents, so you set *Abstract* to *true*, and no one will be able to instantiate *Pet*.
- **Access** Sets the access modifier of generated class to either *Public* or *Internal*.
- **Base Type** This property enables you to set the entity from which this entity will be derived.
- **Documentation** You can enter XML comments for this property. The *summary* and *LongDescription* elements are supported.
- **Entity Set Name** This property contains the name of the collection created on the custom *ObjectContext* class that is created. If you generate the database from this conceptual model, this is also the name of the table that will be generated. The name that is contained in this property is automatically pluralized, but be careful to verify that the pluralized name is valid. For example, entering *Quiz* for the name of the entity will cause Entity Set Name to be set to *Quizzes*, but it should be *Quizzes*. You can change this property as needed.
- **Name** This is the name of the current entity.

After you adjust the entity settings, you will want to add and adjust the user-defined properties. If you select the *Id* property and look at the Properties window, you will see the following property settings:

- **Concurrency Mode** This setting defaults to *None*, which means that this column is not involved in concurrency checks. You can set the property to *Fixed*, which means that the original value of this property is sent as part of the WHERE clause in all update statements or delete statements to ensure that this property has not been modified since it was retrieved.

- **Default Value** This property setting enables you to enter a default initial value for the property.
- **Documentation** You can enter XML comments for this property. The *summary* and *LongDescription* elements are supported.
- **Entity Key** Set this to *true* if the current property is the primary key.
- **Getter** This sets the access modifier for the getter of the current property. Values are *Internal*, *Private*, *Protected*, and *Public*.
- **Name** This setting enables you to set the name of the current property.
- **Nullable** Setting this to *true* enables the current property to contain null values.
- **Setter** This sets the access modifier for the setter of the current property. Values are *Internal*, *Private*, *Protected*, and *Public*.
- **StoreGeneratedPattern** This setting enables you to indicate whether the property value will be auto-generated. *None* indicates that this property value will not be auto-generated. *Identity* indicates that this property value will be an auto-number column and tells the Entity Framework that it needs to get the generated value and feed it back into the entity after an insert. *Computed* indicates that this property value will be calculated after an insert or update. A column whose type is *timestamp* is an example of a column that should have its *StoreGeneratedPattern* property set to *Computed*, because *timestamp* columns get a new value after inserting or updating.
- **Type** Use this setting to set the data type of this property.

To add more properties, right-click the entity and choose Add | Scalar Property. Table 6-2 shows the entities and properties that have been added to the designer surface.

TABLE 6-2 Entities and Properties that Have Been Added to the Designer Surface

ENTITY NAME	PROPERTY NAME	CONCURRENCY MODE	TYPE	MAX LENGTH
<i>Vehicle</i>	<i>Id</i>	Fixed	<i>Int32</i>	
<i>Vehicle</i>	<i>VIN</i>	Fixed	<i>String</i>	17
<i>Vehicle</i>	<i>Make</i>	Fixed	<i>String</i>	20
<i>Vehicle</i>	<i>Model</i>	Fixed	<i>String</i>	20
<i>Vehicle</i>	<i>Year</i>	Fixed	<i>Int32</i>	
<i>Repair</i>	<i>Id</i>	Fixed	<i>Int32</i>	
<i>Repair</i>	<i>VehicleId</i>	Fixed	<i>Int32</i>	
<i>Repair</i>	<i>Description</i>	Fixed	<i>String</i>	100
<i>Repair</i>	<i>Cost</i>	Fixed	<i>Decimal</i>	

To show that two tables are related, add an association from the toolbox by choosing Association, the parent entity (*Vehicle*), and then the child entity (*Repair*). Although not mandatory, you'll find that this operation is best completed by pinning the toolbox open first because this will keep your tables from being covered by the floating toolbox. Pinning the toolbox is accomplished by clicking the thumbtack on the upper-right corner of the toolbox to make the thumbtack point downward (pinned).

After you add the association, check its settings. Simply double-click the association to display its *Referential Constraint* settings. Set the *Principal* property to *Vehicle*, and the *Dependent* property is automatically set to the *Repair* entity. You can also set the *Principal Key* and *Dependent Key* properties. Select *VehicleId* for the *Dependent Key* and click OK. Figure 6-4 shows the completed *Vehicle* (parent) and *Repair* (child) entities with the one-to-many association. Notice that each of the entities now contains *Navigation Properties*. *Navigation Properties* will exist on your classes, and they enable you to navigate easily from the entity with which you're working to either its parent or child entities.

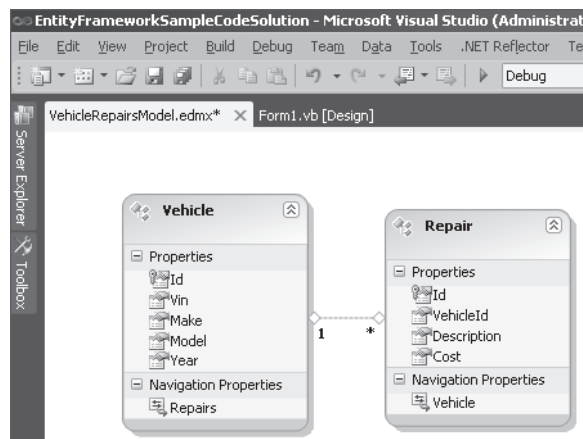


FIGURE 6-4 The completed *Vehicle* and *Repair* entities with the one-to-many association.

Now that this simple conceptual model is complete, create the database. The good news is that you can generate the database from the conceptual model: Right-click the Entity Framework designer surface and then choose *Generate Database From Model*. You are prompted for a database connection. Because this is a new database, you must click *New Connection*. Configure your connection to a valid database server and enter the name of the new database. You are prompted to create a new database. This database is named *RepairDB*. After you create the new database, you see the DDL (Data Definition Language) script and the file name. Click *Finish* to save the script to a file in your project.



EXAM TIP

For the exam, remember that you can create the database from your model by right-clicking the designer surface and choosing *Generate Database From Model*.

The script has been created and saved to a file, but it has not been executed. To execute the script, open the SQL script file, right-click in the opened script file window, and click Execute SQL. You are prompted for a database connection, and then the script executes.

You can view the new database from Server Explorer as shown in Figure 6-5, which shows the Vehicles and Repairs tables that were created and their corresponding columns. The table names came from the pluralized *Entity Set Name* property on each entity. Not shown is the creation of a relationship called FK_VehicleRepair with its foreign key constraint.

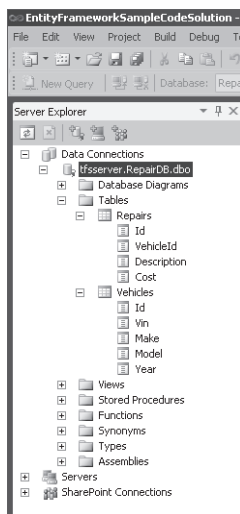


FIGURE 6-5 The created RepairDB contains the Vehicles and Repairs tables.

The database objects that were created are in the dbo schema, which might not be desirable. To change the database schema in which the database objects are created, click the Entity Framework designer surface, and you'll be able to set the general properties of the conceptual model. The properties are as follows:

- **Code Generation Strategy** This turns the default object-layer code generation on or off for the conceptual model. A value of *None* turns off the default code generation; a value of *Default* turns it on. Set the value to *None* when you add a custom text template to generate object-layer code. Custom text templates are covered later in this lesson, in the sections titled "The EntityObject Generator" and "The Self-Tracking Entity Generator."
- **Connection String** Displays the connection string information. To change the connection string, you must modify the .config file.
- **Database Generation Workflow** This is the path for the Window Workflow Foundation file the Generate Database uses from the Model wizard. You can copy and modify the existing file to change the behavior of the database generation.
- **Database Schema Name** The name of the schema to which all generated database objects are added. The default is dbo. If you change the schema name to a schema

that does not exist, the schema is automatically created when you generate the database.

- **DDL Generation Template** This is the name of the T4 text template that transforms the SSDL to DDL when Generate Database from the Model wizard is run.
- **Entity Container Access** This is the access modifier of the entity container. Values can be *Public* or *Internal*. The entity container contains all instances of entities for the entity data model.
- **Entity Container Name** The name of the custom entity container created to hold all instances of entities. This class will derive from the *ObjectContext* class.
- **Lazy Loading Enabled** Set this property to *true* to perform just-in-time loading of data.
- **Metadata Artifact Processing** This property controls whether the model and mapping files (.csdl, .ssdl, and .msl files) are embedded in the compiled assembly (default) or simply copied to the output directory.
- **Namespace** This is the conceptual model's XML namespace.

NOTE CHANGING THE NAMESPACE

To change the namespace of your generated classes, in Solution Explorer, click the EDMX file. In Properties, set the Custom Tool Namespace. In Visual Basic, this entry is concatenated to the default namespace of the project. In C#, this is the absolute namespace of the generated classes.

- **Pluralize New Objects** This setting specifies whether new entity set names and navigation property names are pluralized. Default is *true*.
- **Transform Related Text Templates On Save** A text template is related to an .edmx file by inserting the name of the EDMX file into the text template. When set to *true* (default), all text templates related to the EDMX file are processed when the EDMX file is saved. When the property is set to *false*, none of the related text templates are processed.
- **Validate On Build** This setting specifies whether the model is validated when the project is built. Default is *true*.

Implementing the Database First Model

To use the Database First model, add an ADO.NET Entity Data Model.edmx file to your project: Right-click the project node in Solution Explorer and then choose Add | New Item. Select ADO.NET Entity Data Model. This template produces a file with an EDMX extension. Provide a name for your file. This example (Figure 6-6) demonstrates the creation of a conceptual model for the Northwind database. After naming the file NorthwindModel.edmx, click Add to start the Entity Data Model Wizard.

The previous section covered the Code First model but also covered many of the settings you might want to change even if you implement the Database First model.

Figure 6-2 showed the first step in the Entity Data Model Wizard, which is to select whether you want to generate the model from the database (Database First) or start with an empty model (Code First). Select Generate The Model from the database and click Next. The next step prompts you for a database connection. Configure the connection to reference the Northwind database and click Next. At the prompt, choose Select Your Database Objects, select All Tables, and click Finish.

Examining the Generated Model

Figure 6-6 shows the generated conceptual model of the Northwind database with the cardinality of each association. Notice some associations have a one-to-many relationship, whereas other associations have a zero-to-one-to-many relationship. This varies based on whether the foreign key allows null values.

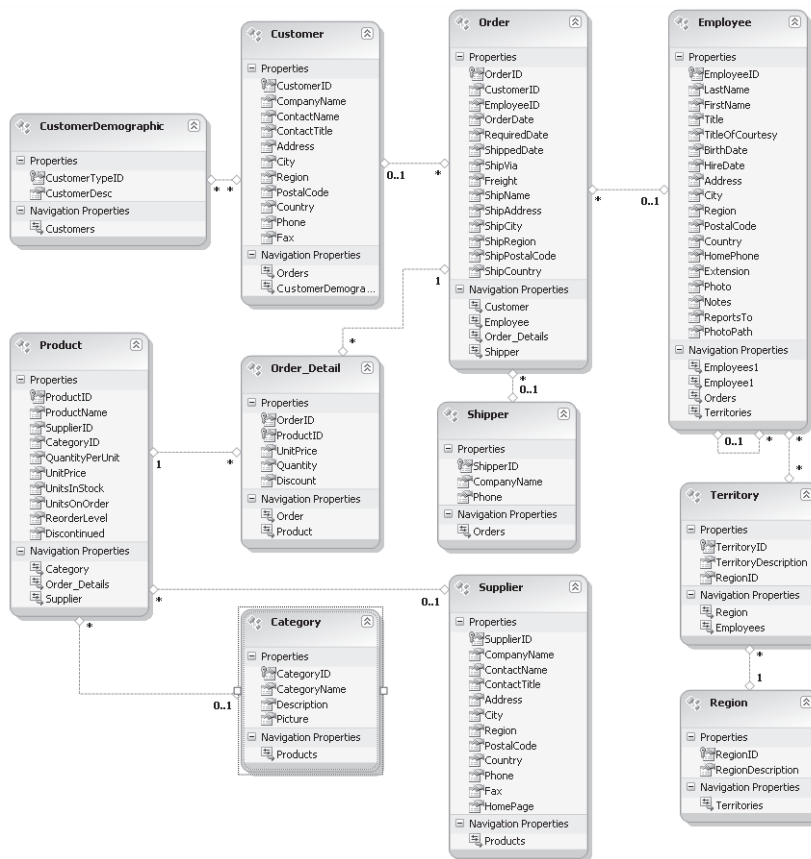


FIGURE 6-6 The Northwind conceptual model.

If you count the entities in Figure 6-6, you'll find that you have eleven, but you have thirteen tables in the database. The reason you're missing two entities is due to the many-to-many association. One of the tables not shown is the CustomerCustomerDemo table between Customer and CustomerDemographic. The other missing table is the EmployeeTerritories table between Employee and Territory. Because these tables exist simply to implement the many-to-many relationship, they are not shown on the conceptual model. You can see the missing table by clicking the many-to-many relationship and viewing the settings in the Mapping Details window. Figure 6-7 shows the selection of the relationship between Customer and CustomerDemographic. When this relationship is selected, the Mapping Details window shows that the CustomerCustomerDemo table implements this relationship.

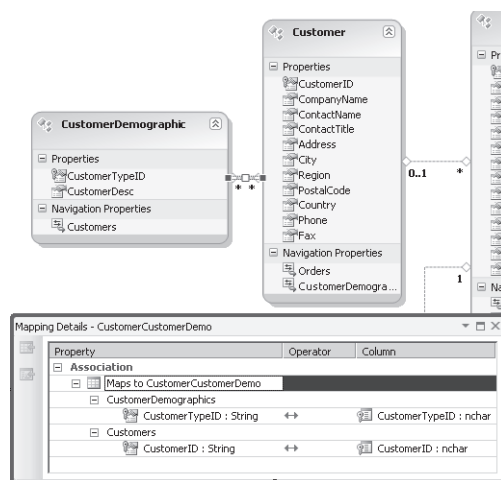


FIGURE 6-7 Select the relationship between Customer and CustomerDemographic to see the CustomerCustomerDemo table in the Mapping Details window.

Keys and Relationships

In Figure 6-7, you can see that the Mapping Details window shows the many-to-many relationship and the keys that join the tables in the relationship. This window is used whenever the join table is hidden when viewing the conceptual model. In addition, if you look at the Properties window, as shown in Figure 6-8, you will see the following properties:

- **Association Set Name** The name of the table implemented to perform a many-to-many relationship or the name of the relationship that joins tables in a relationship other than many-to-many such as one-to-many. For example, in Figure 6-8, this property contains the CustomerCustomerDemo table name, but if you were to click the relationship between Customer and Order, you would find that this setting contains FK_Orders_Customers, which is the name of the database relationship that joins the Customers and Orders tables.
- **Documentation** You can enter XML comments for this property. The *summary* and *LongDescription* elements are supported.

- **End1 Multiplicity** Specifies the number of entities at this end of the association. Valid values are *0* (zero), *0..1* (zero-to-one), or *** (many).
- **End1 Navigation Property** Specifies the name of the *Navigation* property that returns values at this end. For example, if this property is set to *Customers*, there will be a property in the current entity called *Customers*, and you can use this property to retrieve a list of customers that relate to the current entity.
- **End1 OnDelete** Specifies the action to perform on delete of an entity from *End1*. Look at *End1 Role Name* to identify the table to which table *End1* refers. Valid values are either *None*, which will not perform any delete action, or *Cascade*, which will delete the related items on *End2* when an entity on *End1* is deleted.
- **End1 Role Name** Specifies the name of the table to which this end of the relationship is joined.
- **End2 Multiplicity** Specifies the number of entities at this end of the association. Valid values are *0* (zero), *0..1* (zero-to-one), or *** (many).
- **End2 Navigation Property** Specifies the name of the *Navigation* property that returns values at this end. For example, if this property is set to *Orders*, there will be a property in the current entity called *Orders*, and you will be able to use this property to retrieve a list of orders that relate to the current entity.
- **End2 OnDelete** Specifies the action to perform on delete of an entity from *End2*. Look at *End2 Role Name* to identify the table to which table *End2* refers. Valid values are either *None*, which will not perform any delete action, or *Cascade*, which will delete the related items on *End1* when an entity on *End2* is deleted.
- **End2 Role Name** Specifies the name of the table to which this end of the relationship is joined.
- **Name** The name of this relationship object.
- **Referential Constraint** Shows the direction of the relationship.

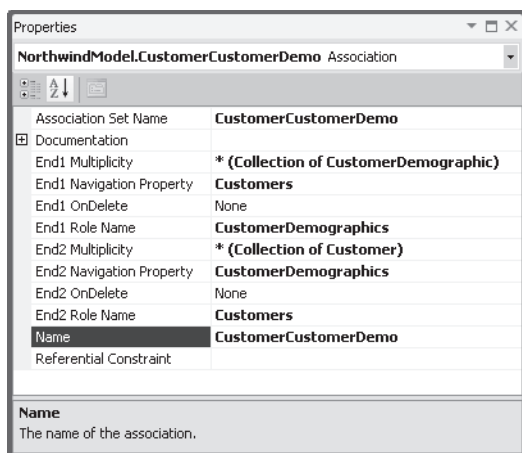


FIGURE 6-8 The relationship has settings in the Properties window that you can view and edit.

Managing your Database Connection and Context Using *ObjectContext*

So far, you've been introduced to data modeling using the Entity Framework, but you will inevitably want to retrieve some data. This is when *ObjectContext* is used. In this section, you will learn about *ObjectContext* and its features.

Retrieving Data with *ObjectContext*

You have created a conceptual model, and now you want to use it in your application. Start by instantiating *ObjectContext*. In the Northwind example, a *NorthwindEntities* class was created, which inherits from *ObjectContext*. You can instantiate the *NorthwindEntities* class and retrieve data very easily, as shown in the following sample code that retrieves the customers and assigns them to *DataGridView*.

Sample of Visual Basic Code

```
Private Sub UsingTheObjectContextToolStripMenuItem_Click( _
    ByVal sender As System.Object, ByVal e As System.EventArgs) _
    Handles UsingTheObjectContextToolStripMenuItem.Click
    Dim db As New NorthwindEntities()
    gv.DataSource = (From c In db.Customers
        Select New With
            {
                .CompanyID = c.CustomerID,
                .CompanyName = c.CompanyName
            }).ToList()
End Sub
```

Sample of C# Code

```
private void usingTheObjectContextToolStripMenuItem_Click(
    object sender, EventArgs e)
{
    var db = new NorthwindEntities();
    gv.DataSource = (from c in db.Customers
        select new
        {
            c.CustomerID,
            c.CompanyName
        }).ToList();
}
```

ObjectContext is the main object in the Entity Framework. The *NorthwindEntities* class derives from *ObjectContext* and adds properties for each of the entities collections, so, in the code sample, the *NorthwindEntities* object contains a *Customers* collection that can be queried.

How Entity Framework Connects to Your Database

The previous code sample used the *NorthwindEntities* class, which inherits from *ObjectContext*, to retrieve data by using a LINQ to Entities query and display the result in a data grid. You might be wondering where the connection is. The following is an abbreviated version of the source code that was generated by the Entity Framework designer to create the *NorthwindEntities* class.

Sample of Visual Basic Code

```
Public Partial Class NorthwindEntities
    Inherits ObjectContext

    Public Sub New()
        MyBase.New("name=NorthwindEntities", "NorthwindEntities")
        MyBase.ContextOptions.LazyLoadingEnabled = true
        OnContextCreated()
    End Sub

    Public Sub New(ByVal connectionString As String)
        MyBase.New(connectionString, "NorthwindEntities")
        MyBase.ContextOptions.LazyLoadingEnabled = true
        OnContextCreated()
    End Sub

    Public Sub New(ByVal connection As EntityConnection)
        MyBase.New(connection, "NorthwindEntities")
        MyBase.ContextOptions.LazyLoadingEnabled = true
        OnContextCreated()
    End Sub

    Public ReadOnly Property Categories() As ObjectSet(Of Category)
        Get
            If (_Categories Is Nothing) Then
                _Categories = MyBase.CreateObjectSet(Of Category)("Categories")
            End If
            Return _Categories
        End Get
    End Property

    Private _Categories As ObjectSet(Of Category)

    Public Function CustOrderHist(customerID As Global.System.String) As ObjectResult(Of CustOrderHist_Result)
        Dim customerIDParameter As ObjectParameter
        If (customerID IsNot Nothing)
            customerIDParameter = New ObjectParameter("CustomerID", customerID)
        Else
            customerIDParameter = New ObjectParameter("CustomerID", GetType(Global.
System.String))
        End If

        Return MyBase.ExecuteFunction(Of CustOrderHist_Result)("CustOrderHist",
customerIDParameter)
```

End Function

End Class

Sample of C# Code

```
public partial class NorthwindEntities :ObjectContext
{
    public NorthwindEntities() : base("name=NorthwindEntities", "NorthwindEntities")
    {
        this.ContextOptions.LazyLoadingEnabled = true;
        OnContextCreated();
    }

    public NorthwindEntities(string connectionString)
        : base(connectionString, "NorthwindEntities")
    {
        this.ContextOptions.LazyLoadingEnabled = true;
        OnContextCreated();
    }

    public NorthwindEntities(EntityConnection connection)
        : base(connection, "NorthwindEntities")
    {
        this.ContextOptions.LazyLoadingEnabled = true;
        OnContextCreated();
    }

    public ObjectSet<Category> Categories
    {
        get
        {
            if (_Categories == null)
            {
                _Categories = base.CreateObjectSet<Category>("Categories");
            }
            return _Categories;
        }
    }
    private ObjectSet<Category> _Categories;

    public ObjectResult<CustOrderHist_Result> CustOrderHist(global::System.String
customerID)
    {
        ObjectParameter customerIDParameter;
        if (customerID != null)
        {
            customerIDParameter = new ObjectParameter("CustomerID", customerID);
        }
        else
        {
            customerIDParameter = new ObjectParameter("CustomerID",
typeof(global::System.String));
        }
    }
}
```

```

        return base.ExecuteFunction<CustOrderHist_Result>("CustOrderHist",
customerIDParameter);
    }
}

```

From this code sample, you can see that the *NorthwindEntities* class inherits from *ObjectContext*. There are three constructors for the class. The first constructor is the parameterless constructor, which is the one that was used in the previous code sample. This constructor calls the *ObjectContext* constructor and passes the name of the connection string stored in the App.Config file and the name of the default entity container. The entity container is a logical container for entity sets, association sets, and function imports. It's defined in the EDMX file. If you want to change the connection string, simply locate the connection information in the App.Config file and make the change.

The second constructor enables you to pass an explicit connection string as a parameter, which is passed to the *ObjectContext* constructor. This works well when you want to modify the connection string in your code and pass the modified connection string to *ObjectContext*.

The third constructor accepts an actual connection object and passes it to the *ObjectContext* constructor. This is especially useful when you are executing a local transaction and you want *ObjectContext* to join the ongoing transaction.

Provider and Connection String Information

Connection strings contain information passed from a data provider to a data source to initialize the connection. The connection string information varies, based on the provider. When working with the Entity Framework, connection strings contain information to connect to the underlying ADO.NET data provider that supports the Entity Framework.

The connection string also contains information about the model and mapping files that the *EntityClient* class uses to access the model, and mapping metadata when connecting to the data source. The following sample is the *NorthwindEntities* connection string.

Sample of Connection String

```

<add name="NorthwindEntities"
  connectionString="metadata=res:/*/*NorthwindModel.csd|
    res:/*/*NorthwindModel.ssd|
    res:/*/*NorthwindModel.msl;
    provider=System.Data.SqlClient;
    provider connection string="
    Data Source=.;Initial Catalog=Northwind;
    Integrated Security=True;
    MultipleActiveResultSets=True";
    providerName="System.Data.EntityClient" />

```

In this connection string, the metadata section references the .csdl, .ssdl, and .msl files, which are resources embedded in the compiled assembly. The format for the resource is as follows.

```
Metadata=res://<assemblyFullName>/<resourceName>.
```

AssemblyFullName is the full name of an assembly with the embedded resource. The name includes the simple name, version name, supported culture, and public key, as follows:

ResourceLib, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null

Resources can be embedded in any assembly that is accessible by the application. You can also specify a wildcard (*) for *AssemblyFullName*, and the Entity Framework run time will search for resources in the following locations, in this order:

1. The calling assembly
2. The referenced assemblies
3. The assemblies in the bin directory of an application

If the files are not in one of these locations, an exception is thrown.

If your resources are in the current assembly, there is no harm in using the wildcard (*), but when you use the wildcard (*) to locate a resource in a different assembly, the Entity Framework has to look through all the assemblies for resources with the correct name. You can improve performance by specifying the assembly name instead of using the wildcard.

Storing Information about Objects and Their State

When a query is executed, the returned objects must be monitored for changes in state for the Entity Framework to have knowledge of the object's need to be persisted. The change-tracking information for the returned objects is stored in *ObjectStateEntry* objects, which *ObjectContext* creates for each retrieved or attached object. The *ObjectStateEntry* class is in the *System.Data.Objects* namespace. Table 6-3 shows the information the *ObjectStateEntry* object stores.

TABLE 6-3 The *ObjectStateEntry* Properties

PROPERTY NAME	DESCRIPTION
<i>CurrentValues</i>	Gets the current property values of the object or relationship associated with this <i>ObjectStateEntry</i>
<i>Entity</i>	Gets a reference to the tracked object that's associated with this <i>ObjectStateEntry</i>
<i>EntityKey</i>	Gets the entity key that's associated with this <i>ObjectStateEntry</i>
<i>EntitySet</i>	Gets the <i>EntitySetBase</i> class of the entity or relationship that's associated with this <i>ObjectStateEntry</i>
<i>IsRelationship</i>	Gets a Boolean value that indicates whether the entity associated with this entity is a relationship
<i>ObjectStateManager</i>	Gets the <i>ObjectStateManager</i> class for this <i>ObjectStateEntry</i>
<i>Original Values</i>	Gets the read-only version of the original property values of the object or relationship associated with this <i>ObjectStateEntry</i> . Objects in an <i>Added</i> state don't have original values

<i>RelationshipManager</i>	Gets a reference to the <i>RelationshipManager</i> class instance for the object represented by the entry
<i>State</i>	Gets the state of the entity associated with this <i>ObjectStateEntry</i>

ObjectContext has a *SaveChanges* method that you can call when you're ready to save all changed objects. To find whether the value of a property has changed between the calls to *SaveChanges*, query the collection of changed property names returned by the *GetModifiedProperties* method on the *ObjectStateEntry* object. The *GetModifiedProperties* method returns a generic *IEnumerable* of *string* containing the names of the properties that have changed.

The state of an entity object can be any of the values described in Table 6-4. To gain access to the state information for a given entity object, use *ObjectContext*, which has an *ObjectStateManager* property. *ObjectStateManager* has a *GetObjectStateEntry* method that accepts an entity parameter and returns the *ObjectStateEntry* object containing a *State* property of type *EntityState* that can take the values defined in Table 6-4.

TABLE 6-4 The *EntityState* Enumeration

MEMBER	DESCRIPTION
<i>Added</i>	The object has been instantiated and has been added to <i>ObjectContext</i> , and the <i>SaveChanges</i> method has not been called. After the changes are saved, the object state changes to <i>Unchanged</i> . Objects in the <i>Added</i> state do not have original values in <i>ObjectStateEntry</i> .
<i>Deleted</i>	The object has been deleted from <i>ObjectContext</i> . After the changes are saved, the object state changes to <i>Detached</i> .
<i>Modified</i>	At least one scalar property on the object has been modified, and the <i>SaveChanges</i> method has not been called. After the changes are saved, the object state changes to <i>Unchanged</i> .
<i>Detached</i>	The object exists but is not being tracked by <i>ObjectContext</i> . An entity is in this state immediately after it has been instantiated and before it is added to <i>ObjectContext</i> . If an entity has been removed from <i>ObjectContext</i> by calling the <i>Detach</i> method, it is also in this state. Also, an object might be in this state if it is loaded by using <i>NoTracking MergeOption</i> . No <i>ObjectStateEntry</i> instance is associated with objects in the <i>Detached</i> state.
<i>Unchanged</i>	The object has not been modified since it was attached to the context or since the last time the <i>SaveChanges</i> method was called.



EXAM TIP

Expect to be tested on the various entity states of an entity object.

ObjectContext Life Cycle

One of the things you must consider is the lifetime of your *ObjectContext*. When *ObjectContext* is referencing many objects to provide change tracking, its memory footprint could become quite large. Otherwise, change tracking and object caching provide value. Here are some guidelines to consider when you're developing an application that requires *ObjectContext*:

- **Memory Usage** The memory footprint must be considered. *ObjectContext* grows in size as it is used because it holds a reference to all the entities it's tracking. Although *ObjectContext* grows, it might be better to have a single shared *ObjectContext* than to have many *ObjectContext* objects in your application.
- **Dispose** *ObjectContext* does implement the *IDisposable* interface, so you might consider instantiating the object within a *using* block. Thus, you would use the *ObjectContext* object to perform a single action. This might not give you the functionality you want because you'll lose object tracking. If you can possibly retrieve your data, modify it, and send the changes back to the database within a single method call, you can use the *using* block within the method. If you can't use the *using* block, you can still benefit from explicitly calling the *Dispose* method whenever you know you're done with the object.
- **Cost Of Construction** Don't be too concerned with the cost to reconstruct *ObjectContext* because this cost is small.
- **Thread Safety** If you are trying to share an *ObjectContext* object across threads, don't. *ObjectContext* is not thread safe. You could explicitly synchronize access to a shared *ObjectContext* object, but you must analyze your scenario and decide whether it's better to have an *ObjectContext* object per thread or incur the cost to synchronize a shared *ObjectContext* object.
- **Stateless** If you are creating a stateless environment, such as a web service, you should not reuse *ObjectContext* across method calls. Therefore, this scenario lends itself to implementing the *using* block, but the contents of the *using* block might be code to query the data that was retrieved in a previous call, so modify the data and then save it.

Lazy Loading vs. Explicit Loading vs. Eager Loading

Lazy loading refers to delaying the loading of data until the data is needed. Lazy loading is also known as just-in-time loading, lazy initialization, on-demand loading, and deferred loading. In many applications, you will want lazy loading when the data is not retrieved until it's required. In Entity Framework 4, lazy loading is enabled by default, so if you execute

the following code sample that retrieves the first *Order* object for the customer whose CustomerID is ALFKI and then retrieves the list of order details for that *Order* object, the three order detail objects are displayed.

Sample of Visual Basic Code

```
Private Sub LazyLoadingToolStripMenuItem_Click( _  
    ByVal sender As System.Object, ByVal e As System.EventArgs) _  
    Handles LazyLoadingToolStripMenuItem.Click  
    MessageBox.Show("Don't forget to set the Lazy Loading Enabled " & _  
        "property on the EDMX file to true!")  
    Dim db As New NorthwindEntities()  
    Dim order = db.Orders.Where(Function(o) o.CustomerID = "ALFKI").First()  
    gv.DataSource = order.Order_Details.ToList()  
End Sub
```

Sample of C# Code

```
private void lazyLoadingToolStripMenuItem_Click(  
    object sender, EventArgs e)  
{  
    MessageBox.Show("Don't forget to set the Lazy Loading Enabled " +  
        "property on the EDMX file to true!");  
    var db = new NorthwindEntities();  
    var order = db.Orders.Where(o => o.CustomerID == "ALFKI").First();  
    gv.DataSource = order.Order_Details.ToList();  
}
```

In this example, the order detail objects were not retrieved until the *ToList* method was executed. This lazy loading worked properly—as you expected—because lazy loading is enabled.

From a best practices perspective, turn off lazy loading to ensure that you are not loading data inadvertently. To turn off lazy loading, open your EDMX file, click the designer surface, and, in the Properties window, set the *Lazy Loading Enabled* property to *false*.



EXAM TIP

For the exam, be sure you understand lazy loading. You can expect to get questions related to activating and deactivating lazy loading.

After turning off lazy loading, if you run the query again, no order details are displayed because no data was retrieved. Note that no exception was thrown, which means that you must pay attention to result sets to verify that the results of your queries match what is expected.

At this point, you must decide whether you want to load the order detail objects *explicitly* just before calling the *ToList* method or *eagerly* load the order detail objects when you run the initial query for the order.

To use eager loading, use the *Include* method to include the order detail objects when running the query for the order, as shown in the following code sample.

Sample of Visual Basic Code

```
Private Sub EagerLoadingToolStripMenuItem_Click( _  
    ByVal sender As System.Object, ByVal e As System.EventArgs) _  
    Handles EagerLoadingToolStripMenuItem.Click  
    MessageBox.Show("Don't forget to set the Lazy Loading Enabled " & _  
        "property on the EDMX file to false!")  
    Dim db As New NorthwindEntities()  
    Dim order = db.Orders.Include("Order_Details") _  
        .Where(Function(o) o.CustomerID = "ALFKI").First()  
    gv.DataSource = order.Order_Details.ToList()  
End Sub
```

Sample of C# Code

```
private void eagerLoadingToolStripMenuItem_Click(  
    object sender, EventArgs e)  
{  
    MessageBox.Show("Don't forget to set the Lazy Loading Enabled " +  
        "property on the EDMX file to false!");  
    var db = new NorthwindEntities();  
    var order = db.Orders.Include("Order_Details").Where(  
        o => o.CustomerID == "ALFKI").First();  
    gv.DataSource = order.Order_Details.ToList();  
}
```

To use explicit loading, use the *Load* method to load the order detail objects before executing the *ToList* method. Note that the *Load* method is a *Sub* (C# returns *void*), so the *Load* method can't be *chained* like many of the other query extension methods.

Sample of Visual Basic Code

```
Private Sub ExplicitLoadingToolStripMenuItem_Click( _  
    ByVal sender As System.Object, ByVal e As System.EventArgs) _  
    Handles ExplicitLoadingToolStripMenuItem.Click  
    MessageBox.Show("Don't forget to set the Lazy Loading Enabled " & _  
        "property on the EDMX file to false!")  
    Dim db As New NorthwindEntities()  
    Dim orders = db.Orders.Where(Function(o) o.CustomerID = "ALFKI").First()  
    orders.Order_Details.Load()  
    gv.DataSource = orders.Order_Details.ToList()  
End Sub
```

Sample of C# Code

```
private void explicitLoadingToolStripMenuItem_Click(  
    object sender, EventArgs e)  
{  
    MessageBox.Show("Don't forget to set the Lazy Loading Enabled " +  
        "property on the EDMX file to false!");  
    var db = new NorthwindEntities();  
    var order = db.Orders.Where(o => o.CustomerID == "ALFKI").First();  
    order.Order_Details.Load();  
    gv.DataSource = order.Order_Details.ToList();  
}
```

More Modeling and Design

In the next part of this chapter, you are introduced to more concepts that you need to understand in order to be able to implement a LINQ to Entities solution.

Working with Complex Types

Complex types are nonscalar properties of entities that enable scalar properties to be organized within entities. Complex types consist of scalar properties or other complex type properties. Complex types do not have keys, so they cannot be managed by the Entity Framework except through the parent object.

Classes generated by the Entity Framework tools inherit from *EntityObject*. Complex types that are generated inherit from *ComplexObject*. These classes inherit from a common base class, *StructuralObject*. Scalar properties of complex type objects can be accessed like other scalar properties.

You can create and modify complex types by using the Model Browser window, typically viewable when the EDMX file is open. In the Model Browser window, right-click the Complex Types folder and choose Create Complex Type. A new complex type is added to the folder with a default name, but the default name is selected, so you can simply type in the new name. In this example, a complex type called *Address* is created.

After the complex type has been added, you can add scalar or complex properties. For *Address*, five scalar properties will be added: *AddressLine1*, *AddressLine2*, *City*, *State*, and *PostalCode*. Figure 6-9 shows the completed complex type called *Address*.

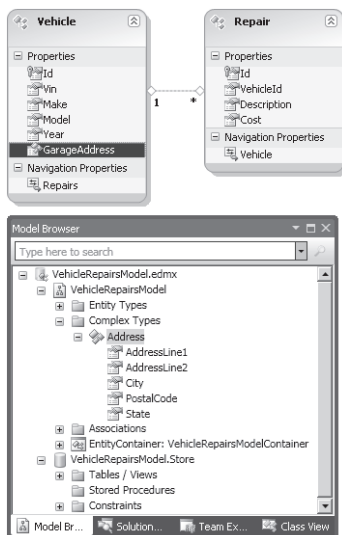


FIGURE 6-9 You can use the Model Browser window to create complex types.

After you create a complex type, you can use it on an entity or another complex type. In this example, a *GarageAddress* property is added to the *Vehicle* entity.

You can also create a complex type from existing properties on an existing entity. Using the Entity Framework designer, select one or more properties on an entity, right-click, and select Refactor Into New Complex Type. This adds a new complex type with the selected properties to the Model Browser. The complex type is given a default name, which you can rename. In addition, the complex property of the newly created type replaces the selected properties, and all property mappings are preserved.

Complex types do not support inheritance and cannot contain navigation properties. It's permissible for a complex type's scalar properties to be null, but a complex type property cannot be null. If *SaveChanges* is called on *ObjectContext* to persist a null complex type property, an *InvalidOperationException* is thrown.

When any property is changed anywhere in the object graph of a complex type, the property of the parent type is marked as changed, and all properties in the object graph of the complex type are updated when *SaveChanges* is called.

When the object layer is generated by the Entity Data Model tools, complex objects are instantiated when the complex type property is accessed and not when the parent object is instantiated.

Mapping Stored Procedures

There are two primary scenarios for mapping a stored procedure into a conceptual model. In one scenario, you simply want to expose a stored procedure and return either entities or complex types. In another scenario, you want to map insert, update, and delete operations for an entity type to stored procedures.

Adding a stored procedure to a conceptual model is also known as adding a function import, which enables you to call the corresponding stored procedure easily from your application code by executing a method on *ObjectContext*. A function import can return collections of simple types, entity types, or complex types, or return no value.

To add existing stored procedures to the conceptual model, right-click the Entity Framework designer surface and choose Update Model From Database. On the Add tab, select the stored procedures you want to add to the conceptual model and click Finish. In this example, all the Northwind stored procedures have been added.

In the Model Browser window, double-click the stored procedure you expose as a method, which opens the Add Function Import dialog box, as shown in Figure 6-10, when double-clicking the *CustOrderHist* stored procedure.

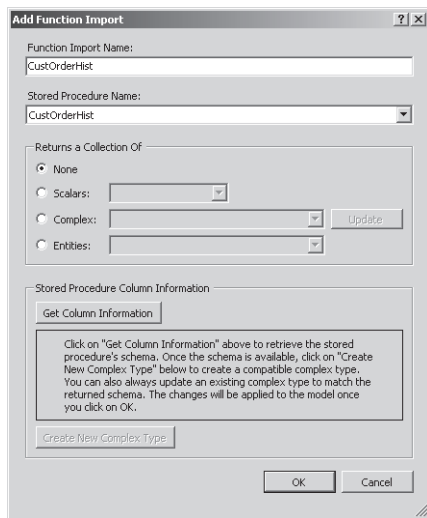


FIGURE 6-10 The Add Function Import dialog box maps a stored procedure to a function import.

In the Add Function Import dialog box, specify the return type of the stored procedure. You can specify a complex type or an entity. If the stored procedure executes a custom SQL statement, you can click Get Column Information to retrieve the schema information. After the schema information has been retrieved, you can click Create New Complex Type to create a complex type to match the returned schema. After you add the function import, you can use it as shown in the following code sample.

Sample of Visual Basic Code

```
Private Sub ExecuteStoredProcedureToolStripMenuItem_Click( _
    ByVal sender As System.Object, ByVal e As System.EventArgs) _
    Handles ExecuteStoredProcedureToolStripMenuItem.Click
    Dim db As New NorthwindEntities()
    gv.DataSource = db.CustOrderHist("ALFKI").ToList()
End Sub
```

Sample of C# Code

```
private void executeStoredProcedureToolStripMenuItem_Click(
    object sender, EventArgs e)
{
    var db = new NorthwindEntities();
    gv.DataSource = db.CustOrderHist("ALFKI");
}
```

The other scenario in which you might use stored procedures is for insert, update, and delete of entities. The *ObjectContext* class exposes a *SaveChanges* method that triggers updates to the underlying database. By default, these updates use SQL statements that are automatically generated, but the updates can use stored procedures that you specify. The good news is that the application code you use to create, update, and delete entities is the same whether or not you use stored procedures to update the database.

To map stored procedures to entities, in the Entity Framework designer, right-click the entity and choose Stored Procedure Mapping. In the Mapping Details window, as shown in Figure 6-11, assign a stored procedure for insert, update, and delete.

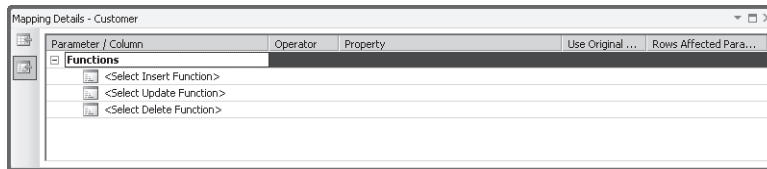


FIGURE 6-11 The Mapping Details window stores procedure mapping

Partial Classes and Methods

The Entity Framework generates classes that contain the properties defined in the conceptual model and do not contain any methods. These generated classes are partial, which means that you can extend these classes by adding your own methods and properties in a separate source file. When you add your code to a separate source file, you don't have to worry about losing your customization if the files are regenerated.

Consider the Northwind database, which has an Order Details table; this table has columns for Quantity, UnitPrice, and Discount. You want to display this information in a data grid, but you also want to include the total price for each row. You might start with a LINQ to Entities query that looks like the following.

Sample of Visual Basic Code

```
Private Sub orderDetailsWoCustomMethodToolStripMenuItem_Click( _
    ByVal sender As System.Object, ByVal e As System.EventArgs) _
    Handles orderDetailsWoCustomMethodToolStripMenuItem.Click
    Dim db As New NorthwindEntities()
    gv.DataSource = (From od In db.Order_Details.AsEnumerable()
        Where od.Order.CustomerID = "ALFKI"
        Select New With
            {
                od.OrderID,
                od.ProductID,
                od.Quantity,
                od.Discount,
                .DetailTotal = (1 - CType(od.Discount, Decimal)) *
                    (od.Quantity * od.UnitPrice)
            }).ToList()
End Sub
```

Sample of C# Code

```
private void orderDetailsWoCustomMethodToolStripMenuItem_Click(
    object sender, EventArgs e)
{
    var db = new NorthwindEntities();

    gv.DataSource = (from od in db.Order_Details.AsEnumerable()
```

```

        where od.Order.CustomerID == "ALFKI"
        select new
        {
            od.OrderID,
            od.ProductID,
            od.Quantity,
            od.Discount,
            DetailTotal = (1 - (decimal)od.Discount) *
                          (od.Quantity * od.UnitPrice)
        }).ToList();
    }

```

In this example, the total for each order detail is calculated in the LINQ to Entities query and assigned to a *DetailTotal* property on the anonymous type in the query. However, you might want access to *DetailTotal* in many places within the application.

A solution to this problem is to add a new property, *DetailTotal*, to the *Order_Detail* class. To add the new method without the risk of losing the method when the classes are regenerated, add a new file to your project that contains a partial class that has the same name as the class that is generated and that must be in the same namespace. In the following example, a new file has been added called *NorthwindExtensions.vb* (C# *NorthwindExtensions.cs*) that contains the *Order_Detail* partial class and the *DetailTotal* method.

Sample of Visual Basic Code

```

Partial Public Class Order_Detail
    Public ReadOnly Property DetailTotal() As Decimal
    Get
        Return (1 - CType(Discount, Decimal)) *
               (Quantity * UnitPrice)
    End Get
End Property
End Class

```

Sample of C# Code

```

namespace EntityFrameworkSampleCode
{
    public partial class Order_Detail
    {
        public double DetailTotal
        {
            get
            {
                return (1 - Discount) *
                       (double)(Quantity * UnitPrice);
            }
        }
    }
}

```

Now that you've add this file with the *Order_Detail* partial class to your project, you can change your LINQ query to use the new property. The following code sample shows the revised query, using the new *DetailTotal* property.

Sample of Visual Basic Code

```
Private Sub orderDetailsWithCustomMethodToolStripMenuItem_Click( _  
    ByVal sender As System.Object, ByVal e As System.EventArgs) _  
    Handles orderDetailsWithCustomMethodToolStripMenuItem.Click  
    Dim db As New NorthwindEntities()  
    gv.DataSource = (From od In db.Order_Details.AsEnumerable()  
        Where od.Order.CustomerID = "ALFKI"  
        Select New With  
            {  
                od.OrderID,  
                od.ProductID,  
                od.Quantity,  
                od.Discount,  
                od.DetailTotal  
            }).ToList()  
End Sub
```

Sample of C# Code

```
private void orderDetailsWithCustomMethodToolStripMenuItem_Click(  
    object sender, EventArgs e)  
{  
    var db = new NorthwindEntities();  
    gv.DataSource = (from od in db.Order_Details.AsEnumerable()  
        where od.Order.CustomerID == "ALFKI"  
        select new  
        {  
            od.OrderID,  
            od.ProductID,  
            od.Quantity,  
            od.Discount,  
            od.DetailTotal  
        }).ToList();  
}
```

Another area of extensibility is with the partial methods created on each entity type. There is a pair of partial methods called *OnXxxChanging* and *OnXxxChanged* for each property, in which *Xxx* is the name of the property. The *OnXxxChanging* method executes before the property has changed, and the *OnXxxChanged* method executes after the property has changed. To implement any of the partial methods, create a partial class as shown in the previous example and add the appropriate partial method with implementation code. In the following code sample, the *OnQuantityChanging* and *OnQuantityChanged* methods have been implemented in the partial class from the previous example.

Sample of Visual Basic Code

```
Private Sub OnQuantityChanging(ByVal value As Global.System.Int16)  
    Debug.WriteLine(String.Format( _  
        "Changing quantity from {0} to {1}", Quantity, value))  
End Sub  
  
Private Sub OnQuantityChanged()  
    Debug.WriteLine(String.Format( _  
        "Changed quantity to {0}", Quantity))  
End Sub
```


End Sub

Sample of C# Code

```
partial void OnQuantityChanging(global::System.Int16 value)
{
    Debug.WriteLine(string.Format(
        "Changing quantity from {0} to {1}", Quantity, value));
}

partial void OnQuantityChanged()
{
    Debug.WriteLine(string.Format(
        "Changed quantity to {0}", Quantity));
}
```

Implementing Inheritance in the Entity Framework

One of the problems with connecting an object-oriented application to a relational database is that the relational database has no concept of inheritance. There are three basic ways to solve this problem when implementing an object-relational mapping (ORM) solution: Table per Class Hierarchy (TPH), Table per Type (TPT), and Table per Concrete Class (TPC). This section covers all three solutions.

TPH

Of the different solutions, probably the simplest and easiest to implement is TPH, which is also known as Single Table Inheritance. To implement this solution, all concrete types in the inheritance hierarchy are stored on one table. The Entity Framework needs to know which type each row is, so you must define a discriminator column that identifies the concrete type to which a specific row is mapped.

Although this solution is simplistic from a database administrator's perspective, this model is not normalized because you typically have columns that have many null values. This is because some types need columns that others don't need, so you must mark columns that are unique per data type as being nullable even if they are not nullable when you are working with a type that requires the column.

This approach is not as efficient regarding disk space consumed, but the trade-off is simplicity and fewer joins to other tables, which can yield better performance.

To demonstrate this solution, consider the scenario in which your application tracks vehicles, and there are many types of vehicles. To keep the example manageable, there is a *Vehicle* base class, a *Car* object that inherits from *Vehicle*, and a *Boat* object that inherits from *Vehicle*. *Vehicle* is abstract and contains *Id*, *Vin*, *Make*, *Model*, and *Year* properties. *Car* inherits from *Vehicle* and has a *TireSize* property. *Boat* inherits from *Vehicle* and has a *PropellerSize* property. All this data is stored in a single table called *Vehicles*, as shown in Figure 6-12.


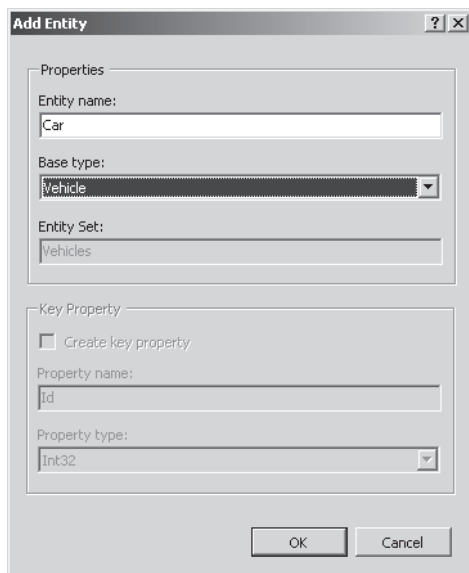
Vehicles			
	Column Name	Data Type	Allow Nulls
	Id	int	<input type="checkbox"/>
	Type	varchar(10)	<input type="checkbox"/>
	Vin	char(17)	<input type="checkbox"/>
	Make	varchar(50)	<input type="checkbox"/>
	Model	varchar(50)	<input type="checkbox"/>
	Year	int	<input type="checkbox"/>
	TireSize	varchar(50)	<input checked="" type="checkbox"/>
	PropSize	varchar(50)	<input checked="" type="checkbox"/>
			<input type="checkbox"/>

FIGURE 6-12 The Vehicles table contains data for the *Vehicle*, *Car*, and *Boat* entities.

The *Id* property is the primary key for all types that this table contains. The *Type* column is the column discriminator, which specifies the object a row represents. *Vin*, *Make*, *Model*, and *Year* are common to all vehicles, so they are contained in a *Vehicle* base class with the *Id*. The *TireSize* column is mandatory for all instances of *Car* and is defined on the *Car* class, but because *Boat* instances don't have tires, this column allows nulls. *PropSize* is mandatory for all *Boat* instances and is defined on the *Boat* class, but *Car* objects don't have propellers, so the *PropSize* column must allow nulls.

After the table is created, a *TablePerHierarchyModel.edmx* ADO.NET Entity Data Model is added, and, in the wizard, *Generate From Database* is selected, the connection and database are selected, and the *Vehicles* table is selected.



The 'Add Entity' dialog box is shown with the following fields:

- Properties:**
 - Entity name:
 - Base type:
 - Entity Set:
- Key Property:**
 - ☐ Create key property
 - Property name:
 - Property type:

Buttons: OK, Cancel

FIGURE 6-13 The Add Entity screen for *Car*.

The Entity Framework designer is displayed with the *Vehicle* entity. Right-click the designer surface and choose Add | Entity. Set Entity Name to *Car* and Base Type to *Vehicle*, as shown in Figure 6-13. Right-click the designer surface again and choose Add | Entity. Set Entity Name to *Boat* and Base Type to *Vehicle*.

At this point, you should be able to see the inheritance hierarchy, but the *TireSize* and *PropSize* properties are still in the *Vehicle* class. Right-click *TireSize* and select Cut; right-click the properties node on the *Car* entity and choose Paste. Next, right-click *PropSize* and choose Cut; right-click the properties node on the *Boat* entity and choose Paste. Your inheritance hierarchy should look like Figure 6-14.

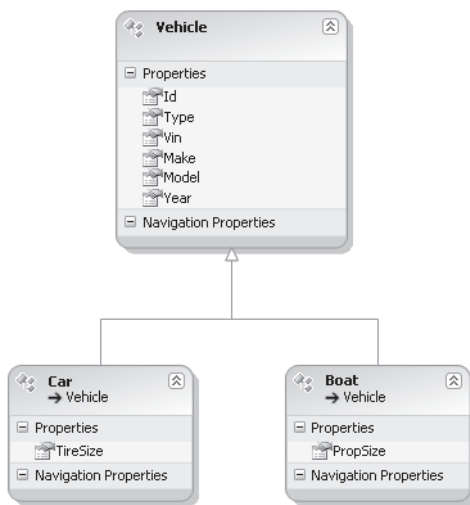


FIGURE 6-14 The inheritance hierarchy is shown with the *TireSize* and *PropSize* properties on the correct entities.

Now that the properties are on the correct entities, it's time to set up the mapping of the *TireSize* and *PropSize* properties. Select the *TireSize* property and note that the Mapping Details window is blank. Under the *Tables* node, click Add A Table Or View. From the drop-down list, select the *Vehicles* table. The *TireSize* column is automatically mapped to the *TireSize* property. You must also specify when a row should map to a *Car* entity. Click Add A Condition and, in the drop-down list, select *Type*. Next, click <Empty String> and type **Car**. Your mapping should look like Figure 6-15.

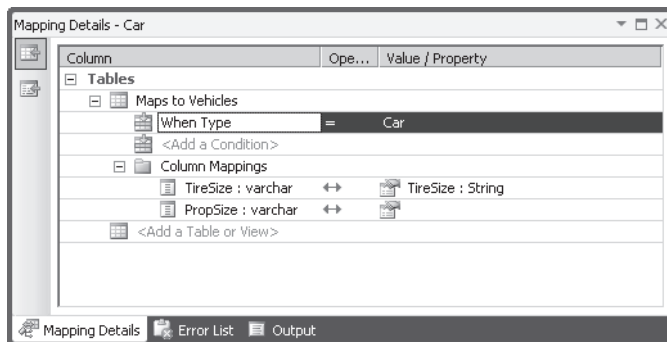


FIGURE 6-15 The Mapping Details window shows the mapping of *Car* to the *Vehicles* table.

Select the *PropSize* property and note that the Mapping Details window is blank. Under the *Tables* node, click *Add A Table Or View*. From the drop-down list, select the *Vehicles* table. The *PropSize* column is automatically mapped to the *PropSize* property. You must also specify when a row should map to a *Boat* entity. Click *Add A Condition* and, in the drop-down list, select the type. Next, click *<Empty String>* and type **Boat**.

Ensure that your generated classes are in an isolated namespace. In Solution Explorer, click the EDMX file. In the Properties window, set the Custom Tool Namespace for Visual Basic to *TablePerHierarchy* or, for C#, to *EntityFrameworkSampleCode.TablePerHierarchy*.

If you try to build your application, you will see the following build error: "Error 159: EntityType 'TablePerHierarchyModel.Vehicle' has no key defined." Define the key for EntityType. Click the *Id* property and, in the Properties window, set the *Entity Key* property to *true*.

If you try to build again, you will see a new error: Error 3023: "Problem in mapping fragments starting at lines 56, 66, 73: Column Vehicles. Type has no default value and is not nullable. A column value is required to store entity data. An Entity with Key (PK) will not round-trip when PK is in 'Vehicles' EntitySet AND Entity is type [TablePerHierarchyModel.Vehicle]." To correct this error, select the *Vehicle* entity and, in the Properties window, set the *Abstract* property to *true*.

If you try to build again, you will see a new error: "Error 3032: Problem in mapping fragments starting at line 56: Condition member 'Vehicles.Type' with a condition other than 'IsNull=False' is mapped. Either remove the condition on Vehicles.Type or remove it from the mapping." To correct this problem, remove the *Type* property from the *Vehicle* entity because the discriminator property should not be mapped. You should then be able to build the project.

To test this, either enter the following data into the table as shown in Table 6-5 or use the sample code that contains this data.

TABLE 6-5 Sample Data in the Vehicles Table

ID	TYPE	VIN	MAKE	MODEL	YEAR	TIRESIZE	PROPSIZE
1	<i>Car</i>	ABC123	BMW	Z-4	2009	225/45R17	NULL
2	<i>Boat</i>	DEF234	SeaRay	SunDeck	2005	NULL	14.75 x 21 SS
3	<i>Car</i>	GHI345	VW	Bug	2007	205/55R16	NULL
4	<i>Boat</i>	JKL456	Harris	FloatBoat	2000	NULL	14-1/2" x 18" LH

The following code sample will display the *Car* objects in a data grid.

Sample of Visual Basic Code

```
Private Sub tPHDisplayCarsToolStripMenuItem_Click( _
    ByVal sender As System.Object, ByVal e As System.EventArgs) _
    Handles tPHDisplayCarsToolStripMenuItem.Click
    Dim db = New TablePerHierarchy.TablePerHierarchyEntities()
    gv.DataSource = (From b In db.Vehicles.OfType(Of TablePerHierarchy.Car)()
        Select b).ToList()
End Sub
```

Sample of C# Code

```
private void tPHDisplayCarsToolStripMenuItem_Click(object sender, EventArgs e)
{
    var db = new TablePerHierarchy.TablePerHierarchyEntities();
    gv.DataSource = (from c in db.Vehicles.OfType<TablePerHierarchy.Car>()
        select c).ToList();
}
```

With a small tweak to change the type to *Boat*, you can retrieve the *Boat* objects as shown in the following code sample.

Sample of Visual Basic Code

```
Private Sub tPHDisplayBoatsToolStripMenuItem_Click( _
    ByVal sender As System.Object, ByVal e As System.EventArgs) _
    Handles tPHDisplayBoatsToolStripMenuItem.Click
    Dim db = New TablePerHierarchy.TablePerHierarchyEntities()
    gv.DataSource = (From b In db.Vehicles.OfType(Of TablePerHierarchy.Boat)()
        Select b).ToList()
End Sub
```

Sample of C# Code

```
private void tPHDisplayBoatsToolStripMenuItem_Click(object sender, EventArgs e)
{
    var db = new TablePerHierarchy.TablePerHierarchyEntities();
    gv.DataSource = (from b in db.Vehicles.OfType<TablePerHierarchy.Boat>()
        select b).ToList();
}
```

TPT

Of the different inheritance solutions, probably the most efficient is TPT. To implement this solution, each type in the inheritance hierarchy is stored in its own table. With a one-to-one mapping of type to table, this solution might also feel the most logical. Also, because of the one-to-one mapping, this solution is typically the most normalized.

One of the potential drawbacks to this solution is the increase in the number of joins you need to get the data. Joins are expensive, but if you're careful about creating indexes on foreign keys where necessary, performance issues can be minimized.

To demonstrate this solution, consider the same scenario described previously in the TPH section, in which your application tracks vehicles and there are many types of vehicles. To keep the example manageable, there is a *Vehicle* base class, a *Car* object that inherits from *Vehicle*, and a *Boat* object that inherits from *Vehicle*. *Vehicle* is abstract and contains *Id*, *Vin*, *Make*, *Model*, and *Year* properties. *Car* inherits from *Vehicle* and has a *TireSize* property. *Boat* inherits from *Vehicle* and has a *PropellerSize* property. All this data is stored in three tables named *Vehicles*, *Cars*, and *Boats*, as shown in Figure 6-16.

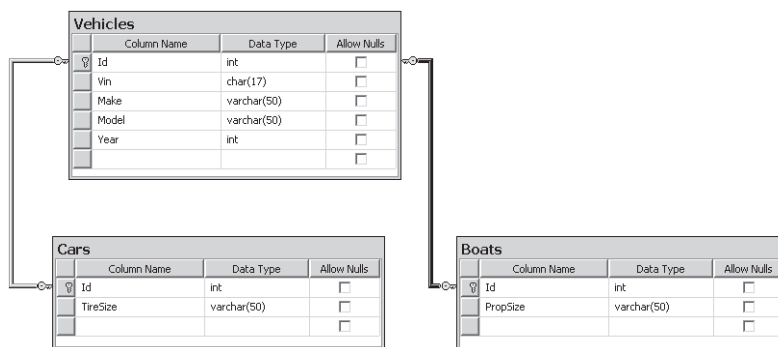


FIGURE 6-16 The tables with their relationships.

Id is the primary key for all types these tables contain. On the *Vehicles* table, the *Id* property is configured to be an auto-number (identity) column. On the *Car* and *Boat* tables, the *Id* property is not an auto-number column because *Car Id* and *Boat Id* get their values from the *Vehicle Id* column. *Vin*, *Make*, *Model*, and *Year* are common to all vehicles, so they are contained in a *Vehicle* base class with the *Id* property. *TireSize* is mandatory for all instances of *Car* and is defined on the *Car* class and on the *Cars* table. *Null* values cannot exist in *TireSize*. *PropSize* is mandatory for all *Boat* instances and is defined on the *Boat* class on the *Boats* table. *PropSize* does not allow null values.

After the tables are created, an ADO.NET Entity Data Model, *TablePerTypeModel.edmx*, is added in the wizard, and the *Vehicles*, *Cars*, and *Boats* tables are selected. The result is shown in Figure 6-17.

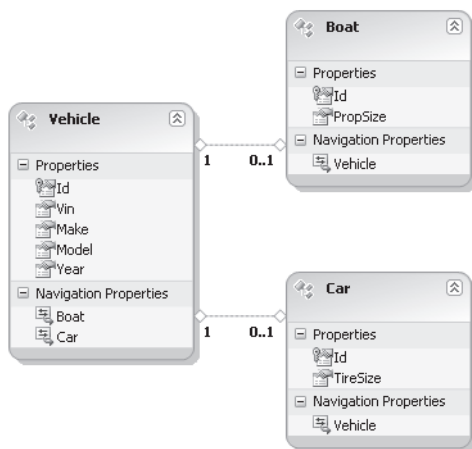


FIGURE 6-17 Updating the model from the database produces entities with associations.

The conceptual model generated from the database shows associations where you want inheritance instead. If you select the *Boat* entity, in the Properties window you will be able to set the *Base Type* property to *Vehicles*. Do the same for *Car* and then click the two associations and delete them. Also delete the *Id* property from *Boat* and *Car* because *Id* is inherited. Click the *TablePerTypeModel.edmx* file and, in the Properties window, set Custom Tool Namespace to *TablePerType*. Last, click the vehicle and set the *Abstract* property to *true*. Figure 6-18 shows the completed conceptual model.

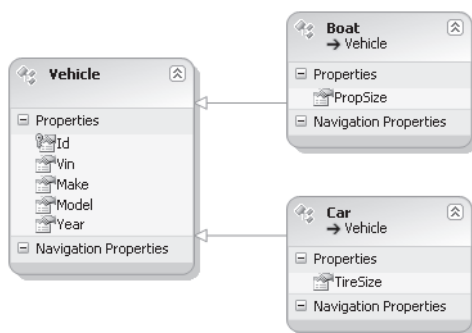


FIGURE 6-18 The completed TPT conceptual model.

The following code sample displays the *Car* objects in a data grid.

Sample of Visual Basic Code

```
Private Sub TPTDisplayCarsToolStripMenuItem_Click( _
    ByVal sender As System.Object, ByVal e As System.EventArgs) _
    Handles TPTDisplayCarsToolStripMenuItem.Click
    Dim db = New TablePerType.TablePerTypeEntities()
    gv.DataSource = (From b In db.Vehicles.OfType(Of TablePerType.Car)()
        Select b).ToList()
```

```
End Sub
```

Sample of C# Code

```
private void tPTDisplayCarsToolStripMenuItem_Click(
    object sender, EventArgs e)
{
    var db = new TablePerType.TablePerTypeEntities();
    gv.DataSource = (from c in db.Vehicles.OfType<TablePerType.Car>()
                     select c).ToList();
}
```

With a small tweak to change the type to *Boat*, you can retrieve the *Boat* objects as shown in the following code sample.

Sample of Visual Basic Code

```
Private Sub TPTDisplayBoatsToolStripMenuItem_Click( _
    ByVal sender As System.Object, ByVal e As System.EventArgs) _
    Handles TPTDisplayBoatsToolStripMenuItem.Click
    Dim db = New TablePerType.TablePerTypeEntities()
    gv.DataSource = (From b In db.Vehicles.OfType(Of TablePerType.Boat)()
                    Select b).ToList()
End Sub
```

Sample of C# Code

```
private void tPTDisplayBoatsToolStripMenuItem_Click(
    object sender, EventArgs e)
{
    var db = new TablePerType.TablePerTypeEntities();
    gv.DataSource = (from b in db.Vehicles.OfType<TablePerType.Boat>()
                     select b).ToList();
}
```

TPC

TPC is another method of implementing inheritance with the Entity Framework. In this solution, you provide a table for each concrete class but no table for the abstract class. To clarify terminology, the abstract class is the base class from which the concrete child classes inherit. Typically, you would instantiate the concrete class in your application. If there is no table for the abstract class, where is the data that is in the abstract class stored? It is included with each of the concrete classes, so you will have the same base-class columns definition in each of the concrete classes.

One of the potential drawbacks to this solution is the duplication of columns in each concrete class. If you add a new property to the abstract class, you must add a column to every concrete class table. Also, if you need to access only values that are in the abstract class, you must create a query that performs a union of all concrete tables. For example, if the *Vin*, *Make*, *Model*, and *Year* are in the abstract class, and you want a list of all vehicles whose year is 2000, you must union all the concrete tables together to get the desired result. This inheritance solution is not used often, relative to TPH and TPT.

To demonstrate this solution, consider the same scenario described in the TPH and TPT sections, in which your application tracks vehicles and there are many types of vehicles. To keep the example manageable, there will be a *Vehicle* base class, a *Car* object that inherits from *Vehicle*, and a *Boat* object that inherits from *Vehicle*. *Vehicle* is abstract and contains *Id*, *Vin*, *Make*, *Model*, and *Year* properties. *Car* inherits from *Vehicle* and has a *TireSize* property. *Boat* inherits from *Vehicle* and has a *PropellerSize* property. All this data is stored in two tables, *Cars* and *Boats*, as shown in Figure 6-19.

Id
Vin
Make
Model
Year
TireSize

Id
Vin
Make
Model
Year
PropSize

FIGURE 6-19 Implementing TPC shows tables with duplicate column names for base class storage.

The *Id* property on both tables is the primary key and is configured to be an auto-number (identity) column. The *Vin*, *Make*, *Model*, and *Year* properties are common to all vehicles, so they are contained in a *Vehicle* base class with *Id* even if the base class data is stored with the concrete class. The *TireSize* property is mandatory for all instances of *Car* and is defined on the *Car* class as well as on the *Cars* entity. Null values cannot exist in *TireSize*. *PropSize* is mandatory for all *Boat* instances and is defined on the *Boat* class and on the *Boats* table. *PropSize* does not allow null values.

TPC class solution is not supported in the Entity Framework designer but can be implemented by editing the EDMX file with an XML editor. Also, you'll find it quite difficult to find documentation on this, so here is a step-by-step walkthrough:

- **Create the database tables** In this example, a new folder called *TablePerConcrete* was added to the project. A service-based database was added to the *TablePerConcrete.mdf* folder; the *Cars* and *Boats* tables were added as shown in Figure 6-19, and these tables were populated with the data as defined in Table 6-4.
- **Add an ADO.NET Entity Data Model** Right-click the *TablePerConcrete* folder, choose *Add | New Item | ADO.NET Entity Data Model*, name it *TablePerConcreteModel.edmx*, and click *Add*. This starts the Entity Data Model Wizard, which displays the *Generate From Database or Empty Model* prompt. Select *Generate From Database* and click *Next*. On the *Choose Your Data Connection* page, select *TablePerConcrete.mdf* as the connection and click *Next*. If you are informed that the connection uses a local data file that's not in the project, click *Yes* to add the data file to the project. On the *Choose Your Database Objects* page, open the *Tables* node, select *Boats* and *Cars*, and click *Finish*. In *Solution Explorer*, select the *TablePerConcreteModel.edmx* file and, in the *Properties* window, set the *Custom Tool Namespace* property to *TablePerConcrete*. At this point, the Entity Framework designer shows the *Boat* and *Car* entities with

properties that match the columns in the tables. The problem is that you want to use inheritance for the columns that exist in both entities.

- **Add the Vehicle Abstract Entity** From the toolbox, drag an entity to the designer surface and name it *Vehicle*. While *Vehicle* is selected, in the Properties window, set the *Abstract* property to *true*. From either the *Boat* or the *Car* entity, select *Make*, *Model*, *Vin*, and *Year*, copy these properties, and then right-click the *Vehicle* entity and click Paste. Click the *Boat* entity and, in the Properties window, choose the Base Type and select *Vehicle*. Click the *Car* entity and, in the Properties window, choose the Base Type and select *Vehicle*. Now that *Boat* and *Car* inherit from *Vehicle*, you can delete the *Id*, *Vin*, *Make*, *Model*, and *Year* properties from the *Boat* and the *Car* columns. Your conceptual model should look like Figure 6-20.

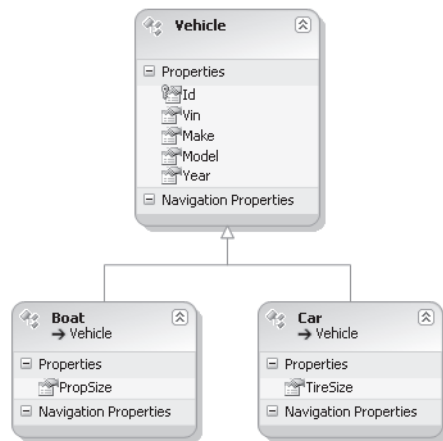


FIGURE 6-20 The completed TPC conceptual model is shown with the base *Vehicle* entity.

- **Edit the EDMX file with an XML Editor** If you click the *Boat* or *Car* entity and look at the Mapping Details window, you'll notice that all of the table columns are shown, but only the *PropSize* and *TireSize* properties are mapped. If you try to map the *Id*, *Vin*, *Make*, *Model*, or *Year* properties, you'll find that the designer won't let you map to a property in the base class. At this point, you've done as much as you can with the Entity Framework designer, so save and close the EDMX file. Next, right-click the EDMX file, choose Open With, and select XML (Text) Editor. Figure 6-21 shows the collapsed structure of the EDMX file.

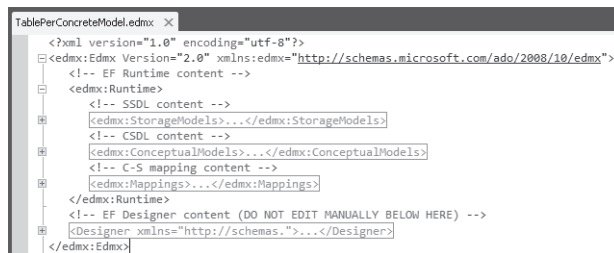


FIGURE 6-21 The collapsed view of the EDMX file shows the sections that comprise this file.

The Store Schema Definition Language (SSDL) content is good because this is the SSDL section that defines the database tables. The CSDL content needs only a small change because this is the CSDL section that represents the conceptual model you just modified in the designer. The following change must be made because the current conceptual model is defining a *Vehicles* collection on the *ObjectContext* class, but you need *Cars* and *Boats* collections instead.

Sample of Existing XML

```
<EntityContainer Name="TablePerConcreteEntities"
  annotation:LazyLoadingEnabled="true">
  <EntitySet Name="Vehicles" EntityType="TablePerConcreteModel.Vehicle" />
</EntityContainer>
```

Sample of New XML

```
<EntityContainer Name="TablePerConcreteEntities"
  annotation:LazyLoadingEnabled="true">
  <EntitySet Name="Cars" EntityType="TablePerConcreteModel.Car" />
  <EntitySet Name="Boats" EntityType="TablePerConcreteModel.Boat" />
</EntityContainer>
```

The CSDL-to-SSDL (C-S) mapping content needs to be changed. There are two issues with this section. First, it doesn't contain the mappings for all the columns of the *Cars* and *Boats* tables to the abstract class columns. Second, it shows the *Cars* and *Boats* columns nested inside *Vehicles*, so you would see only a *Vehicles* property on the *ObjectContext* class, but you want *Cars* and *Boats* to be exposed explicitly on the *ObjectContext* class. Modify this section to look like the following.

Sample of Updated C-S Mapping

```
<!-- C-S mapping content -->
<edm:Mappings>
  <Mapping Space="C-S"
    xmlns="http://schemas.microsoft.com/ado/2008/09/mapping/cs">
    <EntityContainerMapping
      StorageEntityContainer="TablePerConcreteModelStoreContainer"
      CdmEntityContainer="TablePerConcreteEntities">

      <EntitySetMapping Name="Cars">
        <EntityTypeMapping
          TypeName="IsTypeOf(TablePerConcreteModel.Car)">
```

```

    <MappingFragment StoreEntitySet="Cars">
      <ScalarProperty Name="Id" ColumnName="Id" />
      <ScalarProperty Name="Vin" ColumnName="Vin" />
      <ScalarProperty Name="Make" ColumnName="Make" />
      <ScalarProperty Name="Model" ColumnName="Model" />
      <ScalarProperty Name="Year" ColumnName="Year" />
      <ScalarProperty Name="TireSize" ColumnName="TireSize" />
    </MappingFragment>
  </EntityTypeMapping>
</EntitySetMapping>

<EntitySetMapping Name="Boats">
  <EntityTypeMapping
    TypeName="IsTypeOf(TablePerConcreteModel.Boat)">
    <MappingFragment StoreEntitySet="Boats">
      <ScalarProperty Name="Id" ColumnName="Id" />
      <ScalarProperty Name="Vin" ColumnName="Vin" />
      <ScalarProperty Name="Make" ColumnName="Make" />
      <ScalarProperty Name="Model" ColumnName="Model" />
      <ScalarProperty Name="Year" ColumnName="Year" />
      <ScalarProperty Name="PropSize" ColumnName="PropSize" />
    </MappingFragment>
  </EntityTypeMapping>
</EntitySetMapping>

</EntityContainerMapping>
</Mapping>
</edmx:Mappings>

```

- **Add Implementation Code to View Cars** The following code sample retrieves a list of cars and binds the return value to a data grid.

Sample of Visual Basic Code

```

Private Sub TOCDisplayCarsToolStripMenuItem_Click( _
    ByVal sender As System.Object, ByVal e As System.EventArgs) _
    Handles TOCDisplayCarsToolStripMenuItem.Click
    Dim db = New TablePerConcrete.TablePerConcreteEntities()
    gv.DataSource = (From b In db.Cars
        Select b).ToList()
End Sub

```

Sample of C# Code

```

private void tPCDisplayCarsToolStripMenuItem_Click(
    object sender, EventArgs e)
{
    var db = new TablePerConcrete.TablePerConcreteEntities();
    gv.DataSource = (from c in db.Cars
        select c).ToList();
}

```

- **Add Implementation Code to View Boats** The following code sample retrieves a list of boats and binds the return value to a data grid.

Sample of Visual Basic Code

```
Private Sub TOCDisplayBoatsToolStripMenuItem_Click( _  
    ByVal sender As System.Object, ByVal e As System.EventArgs) _  
    Handles TOCDisplayBoatsToolStripMenuItem.Click  
    Dim db = New TablePerConcrete.TablePerConcreteEntities()  
    gv.DataSource = (From b In db.Boats  
                     Select b).ToList()  
End Sub
```

Sample of C# Code

```
private void tPCDisplayBoatsToolStripMenuItem_Click(  
    object sender, EventArgs e)  
{  
    var db = new TablePerConcrete.TablePerConcreteEntities();  
    gv.DataSource = (from b in db.Boats  
                     select b).ToList();  
}
```

Updating the Database Schema

If you make changes to the conceptual model, you will want to update the database schema to match the model. To update the database, right-click the Entity Framework designer surface and choose **Generate Database From Model**. The **Generate Database Wizard** produces a SQL script file that you can edit and execute.

Be aware that the **Generate Database Wizard** does not create a “differencing” script that simply sends only the modifications to the database. In the created script, you’ll see that the script drops the tables and re-creates them.

If you download the **Entity Designer Database Generation Power Pack**, you’ll find that this tool gives you more control of the database generation scripts.

The EntityObject Generator

When reading the database to generate entities, you might want more control over the entities that are generated. For example, maybe you want to implement a custom interface on all your entity classes, or you want every entity class to contain a custom constructor. You can create and edit templates by using **Text Template Transformation Toolkit (T4) Code Generation**. T4 Code Generation is built into Microsoft Visual Studio .NET so that you can use T4 Code Generation immediately. When you create a T4 template, it has a TT extension, which stands for **Text Template**.

Microsoft is including various T4 templates in Visual Studio .NET as a way of rapidly developing templates that help get the job done quickly. The **EntityObject Generator** is a T4 template you can use to generate entity classes that enable you to customize the entities that are created by actually modifying the template that generates the Visual Basic or C# code.

To use the **ADO.NET EntityObject Generator**, create or open an EDMX file and right-click the Entity Framework designer surface. Click **Add Code Generation Item**, as shown in Figure 6-22. The next screen shows the **ADO.NET EntityObject Generator**. When you select it, notice

that a file with a TT extension is created, which indicates that this uses T4 Code Generation. The next screen is a security warning stating that running the T4 template can cause harm to your computer. Clicking OK causes the T4 template to execute and create entities that can be customized.

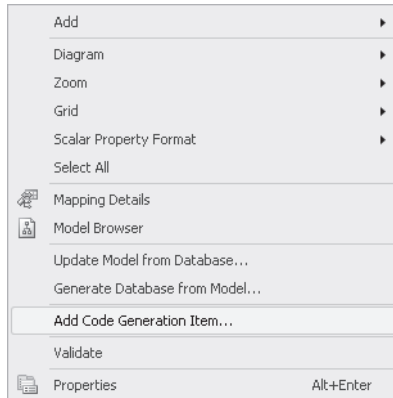


FIGURE 6-22 The Add Code Generation Item menu option enables you to add a T4 template to achieve more control over the generated entities.

Here are some notes about using the EntityObject Generator template:

- The name of the template file determines the name of the code file it generates. For example, if the text template is named `TextTemplate.tt`, the generated file will be named `TextTemplate.vb` or `TextTemplate.cs`.
- The *Custom Tool* property of the targeted .edmx file must be empty.
- The *SourceCSDLPath* initialization below must be set to either the path of the targeted .edmx or .csdl file or to the path of the targeted .edmx or .csdl file relative to the template path.

After you add the ADO.NET EntityObject Generator, look at the Visual Basic or C# file that normally exists under the EDMX file: The file is essentially empty except for a note stating that default code generation is disabled. The *Code Generation Strategy* property on the EDMX file can be changed from *None* back to *Default* if you want to discontinue using the EntityObject Generation template.

Where are the generated entities? They are in the Visual Basic or C# file located under the T4 text template file. What value did this provide? You can have the template that creates the entities so you can modify the template that creates the entities, which means that you get a great amount of power and flexibility that you didn't have before. Prior to the implementation of the T4 template, you had to rely entirely on the template that was built into Visual Studio .NET. Now, you can modify the EntityObject Generator text template, and code generation will be produced based on the modified text template contents.

The Self-Tracking Entity Generator

Until now, when working with the Entity Framework, you created entities that are tracked by *ObjectContext*. If you change any of your entities, *ObjectContext* knows about the change, and it can persist the change when you call the *SaveChanges* method on *ObjectContext*. What happens if *ObjectContext* is disposed? What happens if you serialize your entities and send them across tiers, such as when you have an N-tier application? In these scenarios, the entity objects and *ObjectContext* become disconnected.

The ADO.NET Self-Tracking Entity Generator text template generates the object-layer code that consists of a custom typed *ObjectContext* and entity classes that contain self-tracking state logic so that the entities themselves keep track of their state instead of *ObjectContext* doing so. Probably the best usage of self-tracking entities is when working with N-tier applications.

You add the ADO.NET Self-Tracking Entity (STE) Generator text template to an EDMX file the same way as you add the EntityObject Generator: with an open EDMX file, you right-click the designer surface, choose Add Code Generation Item, and then select the ADO.NET Self-Tracking Entity Generator. When you add the ADO.NET Self-Tracking Entity Generator, you get two text template (.tt) files. The first file is the <model name>.tt, which generates the self-tracking entity types. The second file is the <model name>.Context.tt, which generates the typed *ObjectContext* class.

After you add the STE Generator, the Visual Basic or C# file under the EDMX file will be empty. The STE classes are created as separate class files under the <model name>.tt file. If you open one of the STE class files, you'll find much more code than the amount of code that was in the original entity class files, because the STE classes contain the code to track the state of the object.

The following extension methods can be called on self-tracking entities:

- **StartTracking Method** This method instructs the change tracker on the entity to start recording any changes applied to scalar properties, collections, and references to other entities. Tracking starts automatically when an STE is deserialized into the client through the Windows Communication Foundation (WCF). Tracking is also turned on for newly created entities when a relationship is created between the new entity and an entity that is already tracking changes. Tracking is also turned on when any of the *MarkAs* methods are executed.
- **StopTracking Method** The *StopTracking* method stops recording changes.
- **MarkAs Methods** All the *MarkAs* methods turn tracking on. These extension methods facilitate changing the state of an entity explicitly to *Added*, *Modified*, *Deleted*, or *Unchanged*. The *MarkAs[State]* methods return the same entity to which they are applied, with the modified state.
- **MarkAsAdded Method** This method changes the state of the entity to *Added*. When new self-tracking entities are created, they are in the *Added* state with change tracking not enabled.

- **MarkAsDeleted Method** This method changes the state of the entity to *Deleted* and clears the navigation properties on the entity that is being marked for deletion, essentially severing the relationship(s). If the navigation property is a reference object, the property is set to *null*. If the navigation property represents a collection, its *Clear* method is called. When *MarkAsDeleted* is called on an object that is part of a collection, the object is removed from the collection. To mark each object in a collection as deleted, mark the objects in a copy of the collection. You can create a copy of a collection by calling the *ToArray()* or *ToList()* method on the collection.
- **MarkAsModified Method** This method changes the state of the entity to *Modified*. Modifying the value of a property on an entity that has change tracking enabled also sets the state to *Modified*.
- **MarkAsUnchanged Method** This method changes the state of the entity to *Unchanged*. *AcceptChanges* also clears the change-tracking information for an entity and moves its state to *Unchanged*.
- **AcceptChanges Method** This method clears the change-tracking information for an entity and changes its state to *Unchanged*. If you want to reset the state of a relationship, call *AcceptChanges* on both entities that participate in the relationship.

Commonly, you will be using self-tracking entities to transfer objects across tiers, using WCF. In this scenario, there are some items for which you should specifically watch. Consider the following items:

- Be sure that your client project has a reference to the assembly containing the entity types. If you don't have a reference to the assembly with the entity types, and you add the WCF service reference to the client project, the client project will use the WCF proxy types that were created by adding the service reference and not the actual self-tracking entity types. This is a problem because the WCF proxy types that are created contain only the data and don't contain any methods that the STEs have, so you will not get the automated notification features that manage the tracking of the entities on the client. If you intentionally do not want to include the entity types, you must set change-tracking information manually on the client for the changes to be sent back to the service.
- Calls to the service operation should be stateless and should create a new *ObjectContext* instance. *ObjectContext* implements *IDisposable*, so consider creating *ObjectContext* in a *using* block.
- You might have to send a modified object graph from the client to the service but then intend to continue working with the same graph on the client. In this scenario, you have to iterate through the object graph manually and call the *AcceptChanges* method on each object to reset the change tracker. If objects in your object graph contain properties that are populated with database-generated values such as identity values, the Entity Framework replaces values of these properties with the database-generated values after the *SaveChanges* method is called. You might want to implement your service operation to return saved objects or a list of generated property values for

the objects back to the client. The client could then replace the object instances or object property values with the objects or property values returned from the service operation.

- Merging object graphs from multiple service requests can introduce objects with duplicate key values in the resulting graph. The Entity Framework does not remove objects with duplicate keys. When you call the *ApplyChanges* method, an exception is thrown.
- When you change the relationship between objects by setting the foreign key property, the reference navigation property is set to *null* and not synchronized to the appropriate principal entity on the client. After the object graph is attached to the *ObjectContext* class and you call the *ApplyChanges* method, the foreign key properties and navigation properties will be synchronized.
- Not having a reference navigation property synchronized with the appropriate principal object is a problem if you have specified cascade delete on the foreign key relationship. If you delete the principal object, the delete is not propagated to the dependent objects. If you have cascade deletes specified, use the navigation properties to change relationships instead of setting the foreign key property.
- Self-tracking entities cannot perform lazy loading.
- Binary serialization and serialization to ASP.NET state management objects are not supported by self-tracking entities, but you can modify the text template to add binary serialization support as needed.

POCO Entities

POCO stands for Plain Old CLR Objects, which essentially means you can use the classes you create, and the Entity Framework enables you to use your classes without making any modifications to the classes themselves as long as the names of the entity types, complex types, and properties in the custom data classes match the names of the entity types, complex types, and properties in the conceptual model. This enables you to use domain objects with your data model.

When the Entity Framework was first released (.NET 3.5 SP1), many restrictions were imposed on entity classes. Entity classes had to be subclasses of *EntityObject* or had to implement a set of interfaces, referred to as IPOCO. These interfaces include *IEntityWithKey*, *IEntityWithChangeTracker*, and *IEntityWithRelationships*. The many restrictions became a problem when trying to create domain classes that were truly independent of persistence concerns.

Entity Framework 4.0 supports POCO types that don't need to inherit from a base class or implement any interfaces to get persistence. There is also no need for metadata or mapping attributes on type members, so you can create simple entity classes that are coded as shown. The following code sample defines POCO classes for working with Northwind Customers and Orders tables.

Sample of Visual Basic Code

```
Public Class Customer
    Public Property CustomerID As String
    Public Property CompanyName As String
    Public Property ContactName As String
    Public Property ContactTitle As String
    Public Property Address As String
    Public Property City As String
    Public Property Region As String
    Public Property PostalCode As String
    Public Property Country As String
    Public Property Phone As String
    Public Property Fax As String
    Public Property Orders() As List(Of Order)
End Class
```

```
Public Class Order
    Public Property OrderID As Integer
    Public Property CustomerID As String
    Public Property EmployeeID As Nullable(Of Integer)
    Public Property OrderDate As Nullable(Of DateTime)
    Public Property RequiredDate As Nullable(Of DateTime)
    Public Property ShippedDate As Nullable(Of DateTime)
    Public Property ShipVia As Nullable(Of Integer)
    Public Property Freight As Nullable(Of Decimal)
    Public Property ShipName As String
    Public Property ShipAddress As String
    Public Property ShipCity As String
    Public Property ShipRegion As String
    Public Property ShipPostalCode As String
    Public Property ShipCountry As String
    Public Property Customer As Customer
End Class
```

Sample of C# Code

```
public class Customer
{
    public string CustomerID { get; set; }
    public string CompanyName { get; set; }
    public string ContactName { get; set; }
    public string ContactTitle { get; set; }
    public string Address { get; set; }
    public string City { get; set; }
    public string Region { get; set; }
    public string PostalCode { get; set; }
    public string Country { get; set; }
    public string Phone { get; set; }
    public string Fax { get; set; }
    public List<Order> Orders { get; set; }
}

public class Order
{
    public int OrderID { get; set; }
```

```

    public string CustomerID { get; set; }
    public int? EmployeeID { get; set; }
    public DateTime? OrderDate { get; set; }
    public DateTime? RequiredDate { get; set; }
    public DateTime? ShippedDate { get; set; }
    public int? ShipVia { get; set; }
    public decimal? Freight { get; set; }
    public string ShipName { get; set; }
    public string ShipAddress { get; set; }
    public string ShipCity { get; set; }
    public string ShipRegion { get; set; }
    public string ShipPostalCode { get; set; }
    public string ShipCountry { get; set; }
    public Customer Customer { get; set; }
}

```

You can then use the Entity Framework to query and instantiate these types from the database, and you'll automatically get change tracking, updating, and all other services the .NET Framework offers. The compelling aspect of this scenario is that you might already have your POCO classes in your existing code, so adding an EDMX file and generating the conceptual model from the database can be quick and easy. Simply make sure that the names on your POCO classes match the names of your conceptual entities.

To get started with POCO classes, be aware of the following factors:

- An EDMX file that contains the conceptual model is still required.
- You must turn off the building of .NET classes on the EDMX file by clearing the *Custom Tool* property on the EDMX file. (In Solution Explorer, click the EDMX file to see this setting in the Properties window.)
- You cannot use non-POCO EDMX (an EDMX file with the *Custom Tool* property still set) files and POCO EDMX files in the same project because a non-POCO EDMX file generates an assembly-level attribute, *EdmSchemaAttribute*, that can't exist in an assembly that has a POCO EDMX file. The solution is to create a separate class library project for the POCO or non-POCO EDMX files. In the sample code for this chapter, many examples have already been presented with non-POCO EDMX files, so the POCO example code will be placed in a separate class library project, which is referenced by the sample code project.
- When using POCO, the Entity Framework will search the assembly that contains the POCO EDMX file to locate classes that have the same name as the conceptual model entities. This is done without considering the namespace the POCO classes are in. Therefore, if you have a *Customer* class that's in the *Data* namespace (*Data.Customer*) and another *Customer* class that's in the *ViewModel* namespace (*ViewModel.Customer*), the Entity Framework throws an exception, stating that there is an ambiguous match on the *Customer* class. Once again, the solution is to place your POCO classes and your POCO EDMX file in a separate assembly with a dedicated namespace and reference that assembly from your project.

Getting Started with POCO Entities

After you decide which project will be the home of your POCO EDMX file and POCO classes, you can right-click the project node, choose Add | New Item, and add the ADO.NET Entity Data Model file; in these examples, it's called NorthwindPoco.edmx. This starts the Entity Data Model Wizard, which prompts you to select either Generate From Database or Empty Model. In this example, Generate From Database is selected. The next page prompts for a database connection and a connection settings name. Select a connection to the Northwind database and name the NorthwindPocoEntities connection setting. On the next page, Choose Database Objects, select the Customers and Orders tables and set Model Name to NorthwindPocoModel. Click Finish. The conceptual model is displayed, as shown in Figure 6-23.

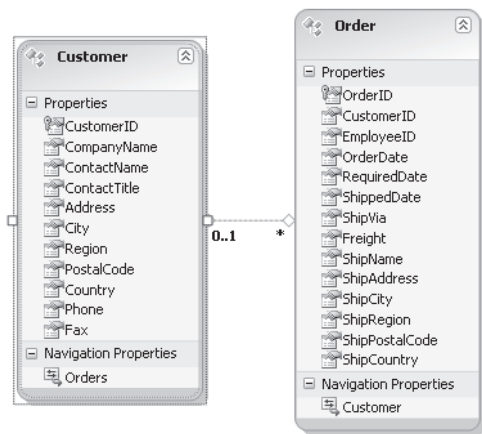


FIGURE 6-23 This is the Northwind POCO conceptual model.

Now that you've added the EDMX file, it's time to configure it. In Solution Explorer, click the EDMX file and then, in the Properties window, clear the *Custom Tool* property to turn off the automatic generation of .NET classes for your conceptual model.

Add the *Customer* and *Order* classes to this project. These classes contain the code from the previous code sample. Be sure they are in the same project as your EDMX file.

Create a custom *ObjectContext* class. The following code sample defines a simple *NorthwindPocoContext* class.

Sample of Visual Basic Code

```
Imports System.Data.Objects
```

```
Public Class NorthwindPocoContext
    Inherits ObjectContext
```

```
    Public Sub New()
        MyBase.New("name= NorthwindPocoEntities", "NorthwindPocoEntities")
        Me._customers = MyBase.CreateObjectSet(Of Customer)()
```

```

        Me._orders = MyBase.CreateObjectSet(Of Order)()
    End Sub

    Public ReadOnly Property Customers As ObjectSet(Of Customer)
        Get
            Return Me._customers
        End Get
    End Property
    Private _customers As ObjectSet(Of Customer)

    Public ReadOnly Property Orders As ObjectSet(Of Order)
        Get
            Return Me._orders
        End Get
    End Property
    Private _orders As ObjectSet(Of Order)
End Class

```

Sample of C# Code

```

public class NorthwindPocoContext :ObjectContext
{
    public NorthwindPocoContext()
        : base("name=NorthwindPocoEntities", "NorthwindPocoEntities")
    {
        _customers = CreateObjectSet<Customer>();
        _orders = CreateObjectSet<Order>();
    }

    public ObjectSet<Customer> Customers
    {
        get
        {
            return _customers;
        }
    }
    private ObjectSet<Customer> _customers;

    public ObjectSet<Order> Orders
    {
        get
        {
            return _orders;
        }
    }
    private ObjectSet<Order> _orders;
}

```

If you added the POCO EDMX file to a class library project, the connection information was stored in an App.Config file within that project. Copy your connection from the App.Config file in your library project to the App.Config file in your executable project.

This completes the setup of a simple POCO scenario. Use the following code to retrieve a list of customers and their orders.

Sample of Visual Basic Code

```
Private Sub pOCODisplayCustomersOrdersToolStripMenuItem_Click( _  
    ByVal sender As System.Object, ByVal e As System.EventArgs) _  
    Handles pOCODisplayCustomersOrdersToolStripMenuItem.Click  
    Dim db = New PocoLibrary.NorthwindPocoContext()  
    gv.DataSource = (From c In db.Customers  
                     Join o In db.Orders  
                     On c.CustomerID Equals o.CustomerID  
                     Select New With  
                         {  
                             c.CustomerID,  
                             c.CompanyName,  
                             o.OrderID,  
                             o.OrderDate  
                         }).ToList()  
End Sub
```

Sample of C# Code

```
private void pOCODisplayCustomersOrdersToolStripMenuItem_Click(  
    object sender, EventArgs e)  
{  
    var db = new PocoLibrary.NorthwindPocoContext();  
    gv.DataSource = (from c in db.Customers  
                     join o in db.Orders  
                     on c.CustomerID equals o.CustomerID  
                     select new  
                         {  
                             c.CustomerID,  
                             c.CompanyName,  
                             o.OrderID,  
                             o.OrderDate  
                         }).ToList();  
}
```

Other POCO Considerations

It was relatively easy to set up and configure POCO support by using the Entity Framework. Now that you have a working POCO sample, you might consider the following:

- Instead of manually creating the POCO classes, you could set up a T4 text template to generate POCO classes for you.
- Your POCO class getters and setters can have any access modifier (*public*, *private*, and so on), but none of the mapped properties can be *overrideable* (C# *virtual*), and you can't specify that you require partial trust support.
- Any collection type that implements the generic *ICollection* interface can be used as a collection when creating your POCO types. If you don't initialize a collection when instantiating a POCO class that has a collection, a generic *List* will be created.
- In many object-oriented applications, it's common to have only one-way navigability between objects. For example, from the *Customer* object, you can navigate to the

Order objects the customer has, but you can't navigate to the *Customer* object from an *Order* object. This one-way navigability is supported as long as your conceptual model shows the same behavior.

- POCO does support lazy (just-in-time) data loading, but only through the creation of proxy types that implement lazy-loading behavior on top of your POCO classes.

Model-Defined Functions

You create a model-defined function within the conceptual model of your EDMX file to provide extra functionality within your LINQ to Entity queries.

An example of a model-defined function would be an instance when you want to retrieve the total price of a line from the Order Details table in the Northwind database. This table contains a row for each line in an order but provides only the quantity, unit price, and discount of each item, not the total price for the line. The same scenario was considered earlier in this chapter in the "Partial Classes and Methods" section. Although a solution was proposed, it used a client-side extension method and required the *AsEnumerable* extension method to call the method. In this example, a model-defined method is created, and the function executes server-side.

To add a model-defined function, open the EDMX file with the XML editor by right-clicking the EDMX file, which, in this case, is the NorthwindModel.edmx file, choose Open With | XML (Text) Editor, and click OK. The EDMX file is displayed as XML.

Scroll down until you locate the conceptual model, which is the *edmx:ConceptualModels* tag. Just inside that tag is a *Schema* tag. You add model-defined functions inside the *Schema* tag. Add the following XML inside the *Schema* tag.

Sample of XML

```
<Function Name="DetailTotal" ReturnType="Decimal">
  <Parameter Name="od" Type="NorthwindModel.Order_Detail" />
  <DefiningExpression>
    (od.UnitPrice * od.Quantity) * CAST(1 - od.Discount AS DECIMAL)
  </DefiningExpression>
</Function>
```

This function is called *DetailTotal*, and it returns a decimal value. The function accepts one parameter, *od*, that is of *NorthwindMode.Order_Detail* type. The *DefiningExpression* tag contains the function's expression. In this expression, the SQL *CAST* function, a model-defined function, will be run server-side. Close and save the EDMX file.

The model-defined function has been created in the conceptual model, but you still need a way to connect your code to it. To do so, add a function into your Visual Basic or C# code, which will have to be annotated with the *EdmFunctionAttribute* attribute. This function can be another instance method of the class itself, but best practice is to create a separate class and define this method as *shared* (C# *static*). In the following sample code, a *ModelDefinedFunctions* class has been created, and a *DetailTotal* method has been added.

Sample of Visual Basic Code

```
Public Class ModelDefinedFunctions
    <EdmFunction("NorthwindModel", "DetailTotal")>
    Public Shared Function DetailTotal(ByVal orderDetail As Order_Detail) As Decimal
        Throw New NotSupportedException( _
            "DetailTotal can only be used in a LINQ to Entities query")
    End Function
End Class
```

Sample of C# Code

```
public class ModelDefinedFunctions
{
    [EdmFunction("NorthwindModel", "DetailTotal")]
    public static decimal DetailTotal(Order_Detail orderDetail)
    {
        throw new NotSupportedException(
            "DetailTotal can only be used in a LINQ to Entities query");
    }
}
```

The completed model-defined function can be used in a LINQ to Entities query, and the function will be executed server-side. The following code sample demonstrates the query for order details and uses the *DetailTotal* function.

Sample of Visual Basic Code

```
Private Sub ModelDefinedFunctionsToolStripMenuItem_Click( _
    ByVal sender As System.Object, ByVal e As System.EventArgs) _
    Handles ModelDefinedFunctionsToolStripMenuItem.Click
    Dim db As New NorthwindEntities()
    gv.DataSource = (From od In db.Order_Details
        Where od.Order.CustomerID = "ALFKI"
        Select New With
            {
                od.OrderID,
                od.ProductID,
                od.Quantity,
                od.Discount,
                od.UnitPrice,
                .DetailTotal = ModelDefinedFunctions.DetailTotal(od)
            }).ToList()
End Sub
```

Sample of C# Code

```
private void modelDefinedFunctionsToolStripMenuItem_Click(
    object sender, EventArgs e)
{
    var db = new NorthwindEntities();
    gv.DataSource = (from od in db.Order_Details
        where od.Order.CustomerID == "ALFKI"
        select new
        {
            od.OrderID,
            od.ProductID,
```



```

        od.Quantity,
        od.Discount,
        od.UnitPrice,
        DetailTotal = ModelDefinedFunctions.DetailTotal(od)
    }).ToList();
}

```

In some scenarios, model-defined functions can increase performance. For example, in the previous code sample, the quantity, unit price, and discount were displayed. If you need to display only the value returned by the *DetailTotal* function, the quantity, unit price, and discount don't need to be sent to your client application. When this scenario was covered earlier in the chapter, in the "Partial Classes and Methods" section, the quantity, unit price, and discount were needed because the extension method was being executed in your client application. In that example, you needed those columns, even if they were not displayed.

Design a Database to Track Music

In this practice, you design a database for tracking your music by album and song name. You use the Code First model to design the conceptual model and then generate the database from the conceptual model.

This practice generates the data-access layer by using the Entity Framework designer, but the result of this practice will be used as the starting point for the next practice.

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

EXERCISE Creating the Project and the Data-Access Layer

In this exercise, you create a Windows Presentation Foundation (WPF) application project and use the ADO.NET Entity Data Model to configure the conceptual model and generate the database.

1. In Visual Studio .NET 2010, choose File | New | Project.
2. Select your desired programming language and then select the WPF Application template. For the project name, enter **MusicTracker**. Be sure to select a desired location for this project. For the solution name, enter **MusicTrackerSolution**. Be sure that Create Directory For Solution is selected and then click OK. After Visual Studio .NET finishes creating the project, the home page, *MainWindow.xaml*, is displayed.

NOTE CHECK YOUR VISUAL STUDIO .NET SETTINGS

If you don't see a prompt for the location, it's because your Visual Studio .NET settings are set up to enable you to abort the project and 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.

- 3 You won't add anything to the `MainWindow.xaml` file in this practice, so you can close the file.
4. Add a service-based database to the project by right-clicking the *Project* node in Solution Explorer and choose `Add | New Item | Data | Service-Based Database`. Name the database **MusicDatabase.mdf** and click `Add`. This starts the Data Source Configuration Wizard.
5. On the Data Source Configuration page, select `Entity Data Model` and click `Next`. This starts the Entity Data Model Wizard.
6. The next prompt is to choose either `Generate From Database` or `Empty Model`.
Be careful here; although you will be working with an empty model, selecting `Generate From Database` will attach the database connection string to the EDMX file.
7. Select `Generate From Database` and click `Next`.
8. The next page prompts for a connection, and the `MusicDatabase.mdf` file is already selected. Click `Next`.
9. On the `Choose Your Database Objects` page, click `Finish` because the database is currently empty.
The `MusicDatabase.mdf` and `Model1.edmx` files are now in your project, and the Entity Data Model designer is open.
10. In Solution Explorer, change to the name of the new `Model1.edmx` file to **MusicModel.edmx**.
11. Drag an entity from the toolbox and drop it onto the designer surface. Change the name to **Album**.
12. Drag another entity from the toolbox and drop it onto the designer surface. Change the name to **Song**.
13. On these entities, right-click each entity, choose `Add | Scalar Property`, and name the new property **Name**.
14. In the toolbox, select `Association`, click the *Album* entity, and then click the *Song* entity. You should see the new association. In the Properties window, change *End1 Multiplicity* to `0..1` and *End2 Multiplicity* to an asterisk.
An album can have many songs, and a song can be on one album, many albums, or no album. Your conceptual model should look Figure 6-24.

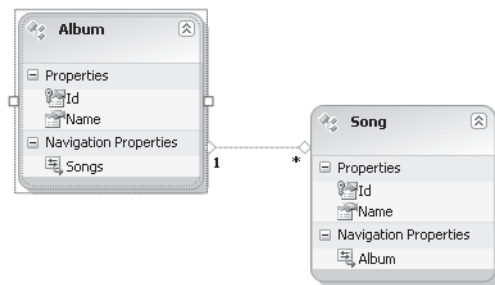


FIGURE 6-24 The MusicTracker conceptual model.

15. You're ready to generate the database from your conceptual model. Right-click an empty location on the Entity Data Model designer surface and click **Generate Database From Model**.
16. On the next page, the **Generate Database Wizard** shows the SQL script that creates the database. Click **Finish**. The SQL script is open for editing and execution.
17. Because you are working with an automatically mounted database, you don't need the *USE* statement in the script, so locate and delete the following statement:

```
USE [MusicDatabase];
```

18. Before you execute the script, devise a connection string that references the database in your project.
In the *App.Config* file, you'll see a reference to `|DataDirectory|` before *MusicDatabase.mdf*, which indicates that at run time, ADO.NET looks for the database in the folder that contains your executable file. If this were a web application, ADO.NET would look in the *Data* directory for the database. This location is good at run time, but when the database is generated, you must reference the *MusicDatabase* physical location in the project.
19. To get a proper connection string for database generation, double-click the *MusicDatabase.mdf* file in *Solution Explorer*. This opens the database in *Server Explorer*.
20. In *Server Explorer*, select *MusicDatabase.mdf* and, in the *Properties* window, copy the contents of the connection string to the clipboard. You can copy the connection string even though the property is read-only. You're now ready to execute the SQL script.
21. Right-click an empty part of the SQL Script file and then click **Execute SQL**. When the **Connect To Database** screen is displayed, click **Options**, click the **Additional Connection Parameters** tab, paste the connection string into this window, and click **Connect**. This executes your SQL script, and you should see a message stating that the execution succeeded. Figure 6-25 shows the created *MusicDatabase* with two tables.

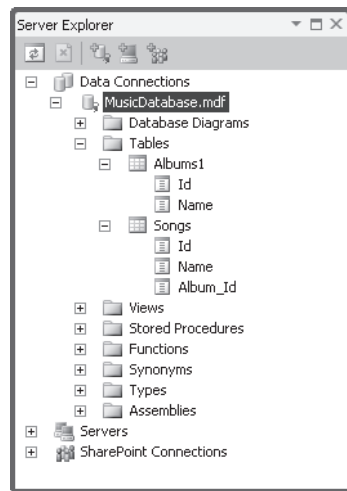


FIGURE 6-25 The created database shows the new tables and includes the join tables.

22. Build your solution; it should be successful.

You now have a database with an Entity Data Model that can query and modify data.

Lesson Summary

This lesson provided detailed information about the Entity Framework, covering basic architecture, data modeling, and connectivity with *ObjectContext*.

- The Object Services layer converts LINQ to Entities and some ESQL queries to a pure Entity SQL command tree.
- EntityClient provider handles the mapping from the conceptual model to the storage model to create an ADO.NET command tree.
- ADO.NET is the underlying technology to communicate to the database.
- You can implement a Code First model by which to create the conceptual model and then generate the database from the conceptual mode.
- You can implement a Database First model by which to generate the conceptual model from the existing database.
- *ObjectContext* is the central object to use for executing queries and updates to the database.
- Lazy loading of data is also called just-in-time loading, when the data doesn't load until you reference it.
- Eager loading is using the *Include* extension method to load child table data when the parent table is being loaded.
- Explicit loading of data is using the *Load* extension method explicitly to load child table data before you use it.

- The Entity Framework supports mapping to complex types and stored procedures.
- The generated entity classes are partial classes and include partial methods.
- The Entity Framework supports Table per Class Hierarchy (TPH) and Table per Type (TPT) inheritance.
- The Entity Framework supports Table per Concrete Class (TPC) inheritance, but the Entity Framework designer does not support it.
- You can modify the conceptual model and send the changes back to the database.
- You can use the Text Template Transformation Toolkit (T4) to generate your entity classes, and Visual Studio .NET provides the T4 EntityObject Generator template by which you can control the entity object generation. Visual Studio .NET also provides the T4 SelfTracking Entity Generator template by which you can create and control the self-tracking entity classes.
- The Entity Framework provides support for the creation of Plain Old CLR Objects (POCO) that can be tracked.
- The Entity Framework enables you to create model-defined functions by which you can create server-side functions that can be called from within a LINQ query.

Lesson Review

You can use the following questions to test your knowledge of the information in Lesson 1, “What Is the ADO.NET Entity Framework?” 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. You are creating a new application that will use a database that already exists. Which model will you use to create your Entity Data Model?
 - A. The Code First model
 - B. The Database First model
2. You have a created *Customer* entity but it comprises properties that are spread across two tables in the database. What should you do?
 - A. Use LINQ to SQL classes.
 - B. Use LINQ to XML.
 - C. Use the Entity Framework.
3. You have created entity classes from your conceptual model, but you can’t add a custom method to one of the classes. Which would be the best approach to adding this method?

- A.** Create a T4 text template and the method.
 - B.** Add an EntityObject Generator to your project and add the new modification to the text template.
 - C.** Add a new partial class to your project and add the method to this class.
 - D.** Open the created entity class and add the method.
- 4.** In your application, you have existing classes that represent the conceptual model. How can the Entity Framework be configured to use these classes?
 - A.** You can use these classes by making the classes inherit from *ObjectContext*.
 - B.** You can use these classes, but you must make the EDMX-generated entity classes inherit from each of your existing classes.
 - C.** You can implement a POCO EDMX file.
- 5.** You have created a conceptual model manually by using the Entity Framework designer. Can you use the Entity Framework designer to generate a new database from the completed conceptual model?
 - A.** Yes
 - B.** No
- 6.** In your application, you want to define a function to be called in your LINQ to Entities query and executed server-side, but you don't have permission to create objects in the database server. How can you solve this problem?
 - A.** Create a model-defined function.
 - B.** Extend the entity class and add a method to it.
 - C.** Create a stored procedure.
- 7.** In the Entity Framework, which is the primary object you use to query and modify data?
 - A.** *ObjectContext*
 - B.** *DataContext*
 - C.** *ObjectStateEntry*
 - D.** *EntityObject*
- 8.** Adding a call to the *Include* extension method in a query for data is an example of what kind of loading strategy?
 - A.** Lazy loading
 - B.** Eager loading
 - C.** Explicit loading

Lesson 2: Querying and Updating with the Entity Framework

In Lesson 1, you learned how the Entity Framework enables you to create a conceptual model of your data and provides the mapping between the entity classes and the storage. Now you will use the Entity Framework to query for data.

After this lesson, you will be able to:

- Execute LINQ to Entity queries.
- Execute Entity SQL queries.
- Execute queries using the database's query language.
- Create a LINQ query to perform an inner join on two element sequences.

Estimated lesson time: 30 minutes

Using LINQ to Entities to Query Your Database

In Chapter 3, "Introducing LINQ," you learned about LINQ and query extension methods. Everything you learned in Chapter 3 is applicable when working with LINQ to Entities.

Query Execution

When you execute a LINQ query, LINQ to Entities converts LINQ queries to command-tree queries. The command-tree query is executed against *ObjectContext*, and the returned objects can be used by both the Entity Framework and LINQ.

The procedure the Entity Framework follows when executing a LINQ query is as follows.

1. *ObjectContext* creates an *ObjectQuery* instance. The *ObjectQuery* class is a generic class that represents a query that returns zero to many typed objects. The *ObjectQuery* instance belongs to the *ObjectContext* class and gets connection information from *ObjectContext*. The *ObjectQuery* class implements the *IQueryable* interface, which defines many methods that can be used to create an object that queried and enumerated to get the query result. This *IQueryable* interface is a generic interface that inherits from the generic *IEnumerable* interface.
2. Using the *ObjectQuery* instance, *ObjectContext* composes a LINQ to Entities query in Visual Basic or C#. The following code sample demonstrates the use of the *ObjectQuery* object and the *IQueryable* interface.

Sample of Visual Basic Code

```
Private Sub ObjectQueryToolStripMenuItem_Click( _  
    ByVal sender As System.Object, ByVal e As System.EventArgs) _  
    Handles ObjectQueryToolStripMenuItem.Click  
    Dim db As New NorthwindEntities()
```

```

Dim customers As ObjectQuery(Of Customer) = db.Customers
Dim result As IQueryable(Of Customer) =
    From c In customers
    Where c.CustomerID.StartsWith("S")
    Select c
gv.DataSource = result.ToList()
End Sub

```

Sample of C# Code

```

private void objectQueryToolStripMenuItem_Click(
    object sender, EventArgs e)
{
    var db = new NorthwindEntities();
    ObjectQuery<Customer> customers = db.Customers;
    IQueryable<Customer> result = from c in customers
                                where c.CustomerID.StartsWith("S")
                                select c;
    gv.DataSource = result.ToList();
}

```

3. The LINQ query operators and expressions are converted to command-tree representations that can be executed against the Entity Framework. LINQ to Entities queries contain query operators and expressions; example query operators are *select* and *where*. Example expressions are $x \geq 0$ and *c.CustomerName.StartsWith("S")*. In LINQ, operators are defined as methods on a class, whereas expressions can contain anything allowed by the types defined in the *System.Linq.Expressions* namespace and anything that can be represented by a lambda function. The Entity Framework allows a subset of LINQ, basically allowing only operations that are run on the database and supported by *ObjectQuery*. Be aware that LINQ defines a number of query operators that are not supported by LINQ to Entities. One example of an unsupported LINQ query operator is the *Where* query extension method, which has an overload that passes in two parameters: the generic item and the corresponding index in the sequence. However, only the single parameter *Where* query extension method typically used is supported. If you attempt to use an unsupported operator, an exception will be thrown.
4. The command-tree representation of the query is executed against the data source. Like LINQ, LINQ to Entities queries provide deferred execution.
5. Return the query results to the client as objects. This is known as *materialization*. Materialization never returns data rows to the client; instead, CLR objects are always instantiated (materialized) and returned.

Difference in Query Execution

Just like LINQ, the Entity Framework supports both LINQ to Entities (also known as query expression syntax) queries and extension method (also known as method-based query syntax) queries. Because LINQ to Entities is a subset of LINQ, it's appropriate to explore things that are unsupported in LINQ to Entities. The following are areas of unsupported items:

- **Projections** The transforming of returned data to a desired result is known as a projection. Projections are supported by the *Select* and *SelectMany* query extension methods. Most overloads of *Select* and *SelectMany* are supported in LINQ to Entities, with the exception of those that accept a positional index argument.
- **Filters** The *Where* query extension method provides the ability to filter. Most overloads of *Where* are supported in LINQ to Entities except those that accept a positional index argument. The following sample code attempts to return every other customer by using the index number of the current customer but throws a *NotSupportedException* because the *Where* overload that has the positional index argument is not supported.

Sample of Visual Basic Code

```
Private Sub UnsupportedToolStripMenuItem1_Click( _
    ByVal sender As System.Object, ByVal e As System.EventArgs) _
    Handles UnsupportedToolStripMenuItem1.Click
    Dim db As New NorthwindEntities()
    Dim result = db.Customers.Where(Function(c, i) i Mod 2 = 0)
    gv.DataSource = result.ToList()
End Sub
```

Sample of C# Code

```
private void unsupportedToolStripMenuItem_Click(
    object sender, EventArgs e)
{
    var db = new NorthwindEntities();
    var result = db.Customers.Where((c, i) => i % 2 == 0);
    gv.DataSource = result.ToList();
}
```

- **Sorting** The query extension methods that support sorting are *OrderBy*, *OrderByDescending*, *ThenBy*, *ThenByDescending*, and *Reverse*. Most overloads of these extension methods are supported except overloads that accept an *IComparer* because this can't be translated to the data source.
- **Joining** The query extension methods that support joining are *Join* and *GroupJoin*. Most overloads of *Join* and *GroupJoin* are supported except overloads that accept an *IEqualityComparer* because the *IEqualityComparer* can't be translated to the data source.
- **Paging** A paging operation returns a single, specific element from a sequence. The supported methods are *First*, *FirstOrDefault*, *Skip*, and *Take*. The unsupported methods are *ElementAt*, *ElementAtOrDefault*, *Last*, *LastOrDefault*, *Single*, *SingleOrDefault*, *SkipWhile*, and *TakeWhile*.
- **Grouping** The grouping method is *GroupBy*. Most overloads of the grouping methods are supported, with the exception of those that use an *IEqualityComparer* because the comparer cannot be translated to the data source.

If you attempt to use any of the unsupported methods, a *NotSupportedException* will be thrown. In many cases, you might be able to execute the desired operation client-side

by adding *AsEnumerable* to the beginning of the query, but using *AsEnumerable* could cause performance problems if executing a method client-side means copying much more data to the client or making many more server calls. The following code sample uses the *AsEnumerable* method to work around the exception that was thrown in the previous example.

Sample of Visual Basic Code

```
Private Sub SupportedToolStripMenuItem1_Click( _
    ByVal sender As System.Object, ByVal e As System.EventArgs) _
    Handles SupportedToolStripMenuItem1.Click
    Dim db As New NorthwindEntities()
    Dim result = db.Customers.AsEnumerable() _
        .Where(Function(c, i) i Mod 2 = 0)
    gv.DataSource = result.ToList()
End Sub
```

Sample of C# Code

```
private void supportedToolStripMenuItem_Click(
    object sender, EventArgs e)
{
    var db = new NorthwindEntities();
    var result = db.Customers.AsEnumerable()
        .Where((c, i) => i % 2 == 0);
    gv.DataSource = result.ToList();
}
```

Although this code sample fixes the problem, you'll find that the SQL statement, as follows, simply queries for all customers. That means SQL Server sent twice as many rows to the client.

SQL Query

```
SELECT
[Extent1].[CustomerID] AS [CustomerID],
[Extent1].[CompanyName] AS [CompanyName],
[Extent1].[ContactName] AS [ContactName],
[Extent1].[ContactTitle] AS [ContactTitle],
[Extent1].[Address] AS [Address],
[Extent1].[City] AS [City],
[Extent1].[Region] AS [Region],
[Extent1].[PostalCode] AS [PostalCode],
[Extent1].[Country] AS [Country],
[Extent1].[Phone] AS [Phone],
[Extent1].[Fax] AS [Fax]
FROM [dbo].[Customers] AS [Extent1]
```

This problem could be solved by implementing a stored procedure, as shown in the following SQL code sample.

SQL Stored Procedure

```
CREATE PROC EveryOtherCustomer
AS
WITH RowData as
(
    SELECT ROW_NUMBER() OVER( ORDER BY CustomerID) as Row, *
```

```

        FROM Customers
    )
    SELECT CustomerID ,CompanyName ,ContactName
        ,ContactTitle ,Address ,City
        ,Region ,PostalCode ,Country
        ,Phone ,Fax
    FROM RowData
    WHERE Row % 2 = 0

```

This stored procedure uses the *WITH* statement to create a common table expression that feeds into the next statement. The next statement returns the Customer rows but filters out every other row. After adding the stored procedure to the EDMX file, you can call the stored procedure, as shown in the following code sample.

Sample of Visual Basic Code

```

Private Sub StoredProcedureToolStripMenuItem_Click( _
    ByVal sender As System.Object, ByVal e As System.EventArgs) _
    Handles StoredProcedureToolStripMenuItem.Click
    Dim db As New NorthwindEntities()
    gv.DataSource = db.EveryOtherCustomer().ToList()
End Sub

```

Sample of C# Code

```

private void storedProcedureToolStripMenuItem_Click(
    object sender, EventArgs e)
{
    var db = new NorthwindEntities();
    gv.DataSource = db.EveryOtherCustomer().ToList();
}

```

SQL Statement

```
exec [dbo].[EveryOtherCustomer]
```

This was an example of how you might run into incompatibilities and what to do to solve this problem. In addition to some of the more common differences described in this section, there are many other incompatibilities and differences between LINQ and LINQ to Entities due to the difference in behavior between the CLR and the data source. For more information, be sure to investigate the MSDN documentation.

Introducing Entity SQL

The Entity Framework enables developers to write object-based queries that target the Entity Data Model without the need to know the logical schema of the database, which could have varying degrees of normalization, as well as different schema styles and different naming conventions for tables and columns, based on who creates the schema. The Entity Data Model insulates developers from the low-level details of the logical model and frees them to focus on the problem at hand.

SQL is the time-tested query language for data access, but the Entity Data Model introduces an enhanced data model that builds on entities, rich types, and relationships. Entity

SQL (ESQL) provides a query language that enables programmers to write queries in terms of Entity Data Model abstractions. ESQL was designed to address this need by supporting types in a clean and expressive way. If you already know SQL, working with ESQL will feel natural. One of the best uses of ESQL is to create complex dynamic queries.

ESQL is not SQL, but you can use ESQL to pass standard SQL queries to the database. It provides access to data as *EntitySet* collections of a given entity type. Using the Northwind database and EDMX file that was created in Lesson 1, *NorthwindEntities.Customers* is a valid query that returns the collection of all *Customers*, whereas *{1,2,3}* is an inline collection of integers. Tables in the database are just collections of a given entity type. Collections can be created, nested, and projected just like any other Entity Data Model type.

When you're working with ESQL, you are working with the EntityClient data provider, as shown in Figure 6-26. Because you are at an abstract level, you can expect ESQL to be provider-independent, so queries written in ESQL can be reused across different database products.

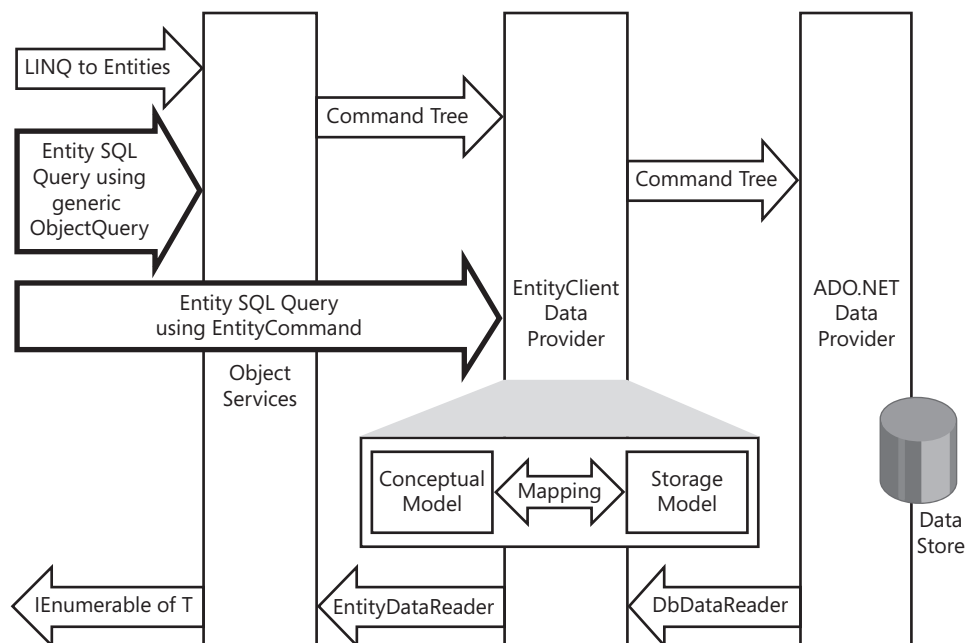


FIGURE 6-26 ESQL can target Object Services and the EntityClient data provider.

When using ESQL, there are two ways of executing queries: You can use *EntityCommand* or the generic *ObjectQuery* class. Both of these are covered in this lesson, but you must first know how to open an entity connection.

Opening an ESQL Connection

To use ESQL, you need an *EntityConnection* object. *EntityConnection* is in the *System.Data.EntityClient* namespace. The connection needs a connection string as well. If the connection is created dynamically or from the beginning, you can use *EntityConnectionStringBuilder* to create the connection string. The connection string must include the metadata that references the .csdl, .ssdl, and .msl files, a subject that was covered in the Lesson 1 section, “Provider and Connection String Information,” of this chapter.

If you already have a connection string stored in your application’s config file, you can set the connection string easily by using “name=NorthwindEntities” as the connection string, where NorthwindEntities is the name of the connection in the config file. The following code sample creates a connection that can be used with ESQL.

Sample of Visual Basic Code

```
Private Sub EntitySQLConnectionToolStripMenuItem_Click( _  
    ByVal sender As System.Object, ByVal e As System.EventArgs) _  
    Handles EntitySQLConnectionToolStripMenuItem.Click  
    Using conn As New EntityConnection("name=NorthwindEntities")  
        conn.Open()  
        MessageBox.Show("Connected!")  
    End Using  
End Sub
```

Sample of C# Code

```
private void entitySQLConnectionToolStripMenuItem_Click(  
    object sender, EventArgs e)  
{  
    using (EntityConnection conn =  
        new EntityConnection("name=NorthwindEntities"))  
    {  
        conn.Open();  
        MessageBox.Show("Connected!");  
    }  
}
```

The *EntityCommand* Object

The *EntityCommand* class inherits from the *DbCommand* class that you learned about in Chapter 2, “ADO.NET Connected Classes,” so coding with an *EntityCommand* object looks like traditional ADO.NET programming. Queries written using the *EntityCommand* object are targeted toward the EntityClient data provider, as shown in Figure 6-26. When you use *EntityCommand*, there is no entity materialization. This means that querying for all customers will not return *Customer* objects; you simply get a *DbDataReader* object, which gives you a stream of row data. The following code sample demonstrates the use of the *EntityCommand* class.

Sample of Visual Basic Code

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

```

        Handles EntityCommandToolStripMenuItem.Click
Using conn As New EntityConnection("name = NorthwindEntities")
    conn.Open()
    Dim myQuery = "SELECT c.CustomerID FROM NorthwindEntities.Customers AS c"
    Using cmd As New EntityCommand(myQuery, conn)
        Dim customerIds = New List(Of String)()
        Dim reader = cmd.ExecuteReader(CommandBehavior.SequentialAccess)
        While reader.Read()
            customerIds.Add(reader.GetString(0))
        End While
        gv.DataSource = (From id In customerIds
                        Select New With {.ID = id}).ToList()
    End Using
End Using
End Sub

```

Sample of C# Code

```

private void entityCommandToolStripMenuItem_Click(object sender, EventArgs e)
{
    using (EntityConnection conn = new EntityConnection("name = NorthwindEntities"))
    {
        conn.Open();
        string myQuery = "SELECT c.CustomerID FROM NorthwindEntities.Customers AS c";
        using (EntityCommand cmd =
            new EntityCommand(myQuery, conn))
        {
            var customerIds = new List<string>();
            var reader = cmd.ExecuteReader(CommandBehavior.SequentialAccess);
            while (reader.Read())
            {
                customerIds.Add((string)reader[0]);
            }
            gv.DataSource = (from id in customerIds
                            select new {ID=id}).ToList();
        }
    }
}

```

This example contains an ESQL query that retrieves a list of customer IDs by using the *NorthwindEntities* container that was defined in the Entity Data Model. You can use dot notation to access the Customers *EntitySet* collection. When you create an alias, you must use the AS keyword. In the example, the reader populates a generic list of string objects called *customerIds*. To display *customerIds* in the data grid, a LINQ query creates a generic list of anonymous types that have an *ID* property. In ESQL, you can't use the asterisk (*) to indicate all columns. When *ExecuteReader* is called to retrieve a stream of rows, you must include *CommandBehavior.SequentialAccess*.

The *ObjectQuery* Class

The generic *ObjectQuery* class targets the object services layer, as shown in Figure 6-26, which enables you to write queries that use your conceptual model entities. Unlike the *EntityCommand* class, you can use the *ObjectQuery* class to retrieve materialized entities,

so you can query for a list of customers and get a collection of *Customer* objects. Using the query from the previous code sample, you can run the same query as with *ObjectQuery*, as follows.

Sample of Visual Basic Code

```
Private Sub ObjectQueryToolStripMenuItem1_Click( _
    ByVal sender As System.Object, ByVal e As System.EventArgs) _
    Handles ObjectQueryToolStripMenuItem1.Click
    Using ctx = NewObjectContext("name=NorthwindEntities")
        ctx.Connection.Open()
        Dim myQuery = "SELECT c.CustomerID FROM NorthwindEntities.Customers AS c"
        Dim customerIds = New ObjectQuery(Of DbDataRecord)(myQuery, ctx).ToList()
        gv.DataSource = (From id In customerIds
            Select New With {.ID = id.GetString(0)}).ToList()
    End Using
End Sub
```

Sample of C# Code

```
private void objectQueryToolStripMenuItem1_Click(object sender, EventArgs e)
{
    using (ObjectContext ctx = new ObjectContext("name=NorthwindEntities"))
    {
        ctx.Connection.Open();
        string myQuery = "SELECT c.CustomerID FROM NorthwindEntities.Customers AS c";
        var customerIds = new ObjectQuery<DbDataRecord>(myQuery, ctx).ToList();
        gv.DataSource = (from id in customerIds
            select new { ID = id.GetString(0) }).ToList();
    }
}
```

This code sample shows that *ObjectQuery* is a bit more simplistic than *EntityCommand*. The query string is exactly the same as the previous code sample, but this sample is easier to execute.

The ROW Function

The Entity Data Model defines three kinds of types: primitive types such as *Int32* and *String*; nominal types that are defined in the schema, such as *Entity* and *Relationship*; and transient types that are anonymous types, such as *collection*, *row*, and *ref*.

In your query, you can create a *row* instance by using the *ROW* function. This enables you to construct rows without accessing a table. The following code sample demonstrates the creation of a *row* instance.

Sample of Visual Basic Code

```
Private Sub RowToolStripMenuItem_Click( _
    ByVal sender As System.Object, ByVal e As System.EventArgs) _
    Handles RowToolStripMenuItem.Click
    Using ctx = NewObjectContext("name=NorthwindEntities")
        ctx.Connection.Open()
        Dim myQuery = "ROW(1 AS MyIndex, 'ALFKI' AS MyId)"
        Dim myStuff = New ObjectQuery(Of DbDataRecord)(myQuery, ctx).ToList()
```

```

gv.DataSource = (From i In myStuff
                  Select New With
                  {
                      .Index = i.GetInt32(0),
                      .Id = i.GetString(1)
                  }).ToList()

End Using
End Sub

```

Sample of C# Code

```

private void rowToolStripMenuItem_Click(object sender, EventArgs e)
{
    using (ObjectContext ctx = new ObjectContext("name=NorthwindEntities"))
    {
        ctx.Connection.Open();
        string myQuery = "ROW(1 AS MyIndex, 'ALFKI' AS MyId)";
        var myStuff = new ObjectQuery<DbDataRecord>(myQuery, ctx).ToList();
        gv.DataSource = (from item in myStuff
                        select new
                        {
                            Index = item.GetInt32(0),
                            Id = item.GetString(1)
                        }).ToList();
    }
}

```

The *MULTISET* Collection

Collections can be created by using the *MULTISET* keyword. For example, *MULTISET(1,2,3,4,5)* creates a collection of five integers. You can also create collections simply by using curly braces {}. For example, {1,2,3,4,5} is the same as *MULTISET(1,2,3,4,5)*. You can use a collection in your query, as shown in the following code sample.

Sample of Visual Basic Code

```

Private Sub MultiSetToolStripMenuItem_Click( _
    ByVal sender As System.Object, ByVal e As System.EventArgs) _
    Handles MultiSetToolStripMenuItem.Click
    Using ctx = New ObjectContext("name=NorthwindEntities")
        ctx.Connection.Open()
        Dim myQuery = "SELECT i FROM {1,2,3,4,5} AS i"
        Dim myStuff = New ObjectQuery(Of DbDataRecord)(myQuery, ctx).ToList()
        gv.DataSource = (From i In myStuff
                        Select New With
                        {
                            .Index = i.GetInt32(0)
                        }).ToList()
    End Using
End Sub

```

Sample of C# Code

```

private void multiSetToolStripMenuItem_Click(object sender, EventArgs e)
{

```



```

using (ObjectContext ctx = new ObjectContext("name=NorthwindEntities"))
{
    ctx.Connection.Open();
    string myQuery = "SELECT i FROM {1,2,3,4,5} AS i";
    var myStuff = new ObjectQuery<DbDataRecord>(myQuery, ctx).ToList();
    gv.DataSource = (from item in myStuff
                     select new
                     {
                         Index = item.GetInt32(0),
                     }).ToList();
}
}

```

Working with the *REF*, *CREATEREF*, and *DEREF* Functions

The *REF* function returns an entity reference to a persisted entity. An entity reference is a form of lightweight entity in which you don't need to consume resources to create and maintain the full entity state until it is really necessary. An entity reference consists of the entity key and an entity set name. Different entity sets can be based on the same entity type, so a particular entity key can appear in multiple entity sets, but an entity reference is always unique. If the input expression represents a persisted entity, a reference to this entity is returned. If the input expression is not a persisted entity, a null reference is returned.

When you are ready to use the referenced entity, you can dereference it by using the *DEREF* function explicitly, or you can dereference it implicitly by just invoking a property of the entity. The following code sample uses *REF* to obtain a reference to a customer and then implicitly dereferences it back to a customer by using the dot (.) to access *CompanyName*.

Sample of Visual Basic Code

```

Private Sub RefDerefToolStripMenuItem_Click( _
    ByVal sender As System.Object, ByVal e As System.EventArgs) _
    Handles RefDerefToolStripMenuItem.Click
    Using ctx = New ObjectContext("name=NorthwindEntities")
        ctx.Connection.Open()
        Dim myQuery = _
            "SELECT REF(c).CompanyName FROM NorthwindEntities.Customers as c"
        Dim myStuff = New ObjectQuery(Of DbDataRecord)(myQuery, ctx).ToList()
        gv.DataSource = (From i In myStuff
                        Select New With
                        {
                            .Name = i.GetString(0)
                        }).ToList()
    End Using
End Sub

```

Sample of C# Code

```

private void refCreateRefDerefToolStripMenuItem_Click(
    object sender, EventArgs e)
{
    using (ObjectContext ctx = new ObjectContext("name=NorthwindEntities"))
    {
        ctx.Connection.Open();
    }
}

```

```

        string myQuery =
            "SELECT REF(c).CompanyName FROM NorthwindEntities.Customers as c";
        var myStuff = new ObjectQuery<DbDataRecord>(myQuery, ctx).ToList();
        gv.DataSource = (from item in myStuff
            select new
            {
                Name = item.GetString(0),
            }).ToList();
    }
}

```

The *CREATEREF* function also returns a reference to a persisted entity, but this function requires an additional parameter, which is the key of the entity you want to locate. The key parameter is passed as a *ROW* function. The following sample code demonstrates the use of *CREATEREF* to create a reference to a single customer by passing in the primary key value as the second parameter. The code then uses the *DEREF* function to return the actual *Customer* object.

Sample of Visual Basic Code

```

Private Sub CreateRefDerefToolStripMenuItem_Click( _
    ByVal sender As System.Object, ByVal e As System.EventArgs) _
    Handles CreateRefDerefToolStripMenuItem.Click
    Using ctx = NewObjectContext("name=NorthwindEntities")
        ctx.Connection.Open()
        Dim myQuery = _
            "DEREF(CREATEREF(NorthwindEntities.Customers, ROW('ALFKI')))"
        gv.DataSource = New ObjectQuery(Of Customer)(myQuery, ctx).ToList()
    End Using
End Sub

```

Sample of C# Code

```

private void createRefDerefToolStripMenuItem_Click(object sender, EventArgs e)
{
    using (ObjectContext ctx = new ObjectContext("name=NorthwindEntities"))
    {
        ctx.Connection.Open();
        string myQuery = "DEREF(CREATEREF(NorthwindEntities.Customers, ROW('ALFKI')))"
        gv.DataSource = new ObjectQuery<Customer>(myQuery, ctx).ToList();
    }
}

```

Working with Entity Sets

Entities are at the core of the Entity Data Model, and an entity set is like a table except that it's within the Entity Data Model. An entity container is like a database. If you want to retrieve all entities in an entity set, you can write a simple ESQ such as the following.

Sample of Visual Basic Code

```

Private Sub EntitySetToolStripMenuItem_Click( _
    ByVal sender As System.Object, ByVal e As System.EventArgs) _
    Handles EntitySetToolStripMenuItem.Click
    Using ctx = NewObjectContext("name=NorthwindEntities")

```

```

        ctx.Connection.Open()
        Dim myQuery = "NorthwindEntities.Customers"
        gv.DataSource = New ObjectQuery(Of Customer)(myQuery, ctx).ToList()
    End Using
End Sub

```

Sample of C# Code

```

private void entitySetToolStripMenuItem_Click(object sender, EventArgs e)
{
    using (ObjectContext ctx = new ObjectContext("name=NorthwindEntities"))
    {
        ctx.Connection.Open();
        string myQuery = "NorthwindEntities.Customers";
        gv.DataSource = new ObjectQuery<Customer>(myQuery, ctx).ToList();
    }
}

```

This example simply specified the name of the container (*NorthwindEntities*) and the entity set (*Customers*). When executed as *ObjectQuery*, the results were materialized, meaning that *Customer* objects were created.

Using Query Builder Methods

The generic *ObjectQuery* object has many extension methods that can perform projection. When you use these methods, you can use the *it* keyword to reference the current entity. The following code sample demonstrates the use of query builder methods to filter the *Customers* entity set and then sort and create a projection that returns only the company name and contact name.

Sample of Visual Basic Code

```

Private Sub QueryBuilderMethodsToolStripMenuItem_Click( _
    ByVal sender As System.Object, ByVal e As System.EventArgs) _
    Handles QueryBuilderMethodsToolStripMenuItem.Click
    Using ctx = New ObjectContext("name=NorthwindEntities")
        ctx.Connection.Open()
        Dim myQuery = "NorthwindEntities.Customers"
        Dim results = New ObjectQuery(Of DbDataRecord)(myQuery, ctx) _
            .Where("StartsWith(it.CompanyName, 'A')") _
            .Select("it.CompanyName, it.ContactName") _
            .OrderBy("it.ContactName") _
            .ToList()
        gv.DataSource = (From i In results
            Select New With
            {
                .Company = i(0),
                .Contact = i(1)
            }).ToList()
    End Using
End Sub

```

Sample of C# Code

```

private void queryBuilderMethodsToolStripMenuItem_Click(object sender, EventArgs e)
{

```

```

using (ObjectContext ctx = new ObjectContext("name=NorthwindEntities"))
{
    ctx.Connection.Open();
    string myQuery = "NorthwindEntities.Customers";
    var results = new ObjectQuery<DbDataRecord>(myQuery, ctx)
        .Where("StartsWith(it.CompanyName, 'A')")
        .Select("it.CompanyName, it.ContactName")
        .OrderBy("it.ContactName")
        .ToList();

    gv.DataSource = (from i in results
        select new
        {
            Company=(string)i[0],
            Contact=(string)i[1]
        }).ToList();
}
}

```

Using *ObjectContext* to Submit Changes to the Database

This chapter has examined the Entity Data Model and various ways of retrieving data by using the Entity Framework. The following section examines modifying data. The balance of this chapter focuses on submitting changes to the database.

Modifying Existing Entities

In Lesson 1, you learned about the *ObjectContext* object and how it tracks changes. You also learned how to model and retrieve entities. So how do you change one of the entities you've retrieved? You simply modify the entity and call the *SaveChanges* method on the *ObjectContext* object you used to retrieve the entity. The following code sample retrieves a single customer whose CustomerID is ALFKI and modifies the *ContactTitle* property. Next, the *SaveChanges* method is called on *ObjectContext* to persist the change back to the database.

Sample of Visual Basic Code

```

Private Sub ModifyExistingDataToolStripMenuItem_Click( _
    ByVal sender As System.Object, ByVal e As System.EventArgs) _
    Handles ModifyExistingDataToolStripMenuItem.Click
    Dim db = New NorthwindEntities()
    Dim customer = (From c In db.Customers
        Where c.CustomerID = "ALFKI"
        Select c).First()
    customer.ContactTitle = "President " + DateTime.Now
    db.SaveChanges()
    MessageBox.Show("Customer Saved")
End Sub

```

Sample of C# Code

```

private void updateExistingDataToolStripMenuItem_Click(
    object sender, EventArgs e)
{

```

```

var db = new NorthwindEntities();
var customer = (from c in db.Customers
                where c.CustomerID=="ALFKI"
                select c).First();
customer.ContactTitle = "President " + DateTime.Now;
db.SaveChanges();
MessageBox.Show("Customer Saved");
}

```

When the *SaveChanges* method is called, the *ObjectContext* object locates all entities whose *EntityState* property is *Added*, *Deleted*, or *Modified* and executes the appropriate *Insert*, *Delete*, or *Update* command to the database. The valid values for *EntityState* are shown in Table 6-4.

The Entity Data Model Generator creates classes in which each property contains *OnPropertyChanging* and *OnPropertyChanged* events. You can extend the entity class to add code that subscribes to these events as needed. Any change to a scalar property causes the entity's *EntityState* property to be *Modified*.

Entities in the *Detached* state are not persisted. You must attach the entity to an *ObjectContext* object to enable change tracking.

Adding New Entities to an *ObjectContext* Class

Before you save a new entity to the database, you must create it. You can use the *new* keyword to instantiate the class and populate all non-nullable properties before saving. If you're working with the Entity Framework-generated classes, they have a static *CreateObjectName* method that can instantiate the class. This method accepts parameters for each non-nullable property. The *CreateObjectName* method does not attach the newly created object to an *ObjectContext* object.

When creating POCO objects, you should use the *CreateObject* method of *ObjectContext* because this creates a proxy object wrapper that inherits from the entity you are creating. The proxy object handles the object tracking.

When you create a new entity such as *Employee*, its initial *EntityState* will be *Detached*. You can populate the *Employee* object with data, but its *EntityState* will still be *Detached* until you attach the entity to an *ObjectContext* object. You can use any of the following methods to attach an entity to an *ObjectContext* object:

- **AddToXxx** Use the *AddToXxx* method on *ObjectContext*, where Xxx is the entity collection name, and the method accepts a single typed entity parameter.
- **AddObject** The *AddObject* method of *ObjectContext* accepts two parameters. The first parameter is a string containing the name of the entity set. The second parameter's type is *object* and references the entity you want to add.
- **AddObject** The *AddObject* method of *ObjectSet*, which is the entity set's type, accepts a single typed parameter containing the entity to be added to an *ObjectSet*.
- **Add** The *Add* method on the *EntityCollection*, which is the type used for a property that represents the "many" end of a relationship, accepts one typed entity parameter.

Before you can save a new entity, you must set all properties that do not support null values. The Entity Framework generates a temporary key value for every new object when it's created. After *SaveChanges* is called, the temporary key value is replaced by the identity value assigned at the database.

If the database is not configured to generate a key value, you should assign a unique value. If multiple objects have the same user-specified key value, an *InvalidOperationException* is thrown when *SaveChanges* is called.

In the following code sample, a new *Employee* entity is created using the *CreateObjectName* method to populate the non-nullable properties. The new entity is passed to the *AddObject* method of the *Employees ObjectSet* to attach the new entity to the *ObjectContext* object. Finally, the *SaveChanges* method is called to persist the new entity to the database.

Sample of Visual Basic Code

```
Private Sub AddNewEntitiesToolStripMenuItem_Click( _
    ByVal sender As System.Object, ByVal e As System.EventArgs) _
    Handles AddNewEntitiesToolStripMenuItem.Click
    Dim db = New NorthwindEntities()
    Dim emp = Employee.CreateEmployee(-1, "John", "Smith")
    MessageBox.Show(emp.EntityState.ToString())
    db.Employees.AddObject(emp)
    MessageBox.Show(emp.EntityState.ToString())
    db.SaveChanges()
    MessageBox.Show("Employee Saved")
    MessageBox.Show(emp.EntityState.ToString())
End Sub
```

Sample of C# Code

```
private void addingNewEntitesToolStripMenuItem_Click(
    object sender, EventArgs e)
{
    var db = new NorthwindEntities();
    var emp = Employee.CreateEmployee(-1, "John", "Smith");
    MessageBox.Show(emp.EntityState.ToString());
    db.Employees.AddObject(emp);
    MessageBox.Show(emp.EntityState.ToString());
    db.SaveChanges();
    MessageBox.Show("Employee Saved");
    MessageBox.Show(emp.EntityState.ToString());
}
```

When this code executes, an *Employee* object is created, and its *EntityState* is displayed as *Detached*. The *Employee* object is added to the *Employees* entity set, which attaches it to the *ObjectContext*, and the *EntityState* is displayed as *Added*. Finally, the *SaveChanges* method is called, and the *Employee* object is saved. You are notified that the *Employee* object has been saved, and the *EntityState* is displayed as *Unchanged*. The *Unchanged* state indicates that no changes have been made to the *Employee* object since it was saved.

Notice that a temporary value is supplied for the *Employee* object, but after the save *EmployeeID* would contain the value assigned to the *Employee* object at the database, which is a positive number.

Attaching Entities to an *ObjectContext*

When a new entity is created, you use one of the *Add* methods described in the previous section to attach the new entity to the *ObjectContext*. This procedure works well if the key has never been assigned, but if the key already has a value, an exception will be thrown. This can happen when an entity is retrieved from the database with the *NoTracking* option or if the entity was detached using the *Detach* method. You might also attach an entity to an *ObjectContext* when the entity comes from a different *ObjectContext* object.

You can use any of the following methods to attach an entity that has a key to an *ObjectContext* object:

- **Attach** Use the *Attach* method of *ObjectContext* where the method accepts a single typed entity parameter.
- **AttachTo** The *AttachTo* method of *ObjectContext* accepts two parameters. The first parameter is a string containing the name of the entity set. The second parameter's type is *object* and references the entity you want to add.
- **Attach** The *Attach* method of *ObjectSet*, which is the entity set's type, accepts a single typed parameter containing the entity to be added to the *ObjectSet*.

The following code is an example of retrieving an entity, detaching it, and then attaching it to a new *ObjectContext* object so that the entity is modified and saved.

Sample of Visual Basic Code

```
Private Sub AttachToolStripMenuItem_Click( _  
    ByVal sender As System.Object, ByVal e As System.EventArgs) _  
    Handles AttachToolStripMenuItem.Click  
    Dim db = New NorthwindEntities()  
    Dim customer = (From c In db.Customers  
                    Where c.CustomerID = "ALFKI"  
                    Select c).First()  
    MessageBox.Show(customer.EntityState.ToString())  
    db.Detach(customer)  
    MessageBox.Show(customer.EntityState.ToString())  
    db = New NorthwindEntities()  
    db.Customers.Attach(customer)  
    MessageBox.Show(customer.EntityState.ToString())  
    customer.ContactTitle = "Pres " + DateTime.Now  
    MessageBox.Show(customer.EntityState.ToString())  
    db.SaveChanges()  
    MessageBox.Show("Customer Saved")  
    MessageBox.Show(customer.EntityState.ToString())  
End Sub
```

Sample of C# Code

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

```

{
    var db = new NorthwindEntities();
    var customer = (from c in db.Customers
                    where c.CustomerID == "ALFKI"
                    select c).First();
    MessageBox.Show(customer.EntityState.ToString());
    db.Detach(customer);
    MessageBox.Show(customer.EntityState.ToString());
    db = new NorthwindEntities();
    db.Customers.Attach(customer);
    MessageBox.Show(customer.EntityState.ToString());
    customer.ContactTitle = "Pres " + DateTime.Now;
    MessageBox.Show(customer.EntityState.ToString());
    db.SaveChanges();
    MessageBox.Show("Customer Saved");
    MessageBox.Show(customer.EntityState.ToString());
}

```

When the sample code is executed, an *ObjectContext* object is created (of type *NorthwindEntities*), a customer is retrieved, and a pop-up shows *EntityState* as *Unchanged*. The customer is then detached from the *ObjectContext*, and a pop-up shows *EntityState* as *Detached*. Next, a new *ObjectContext* object is created, and the customer is attached by using the *Attach* method on the *ObjectSet Customers* property of this new *ObjectContext*. A pop-up shows *EntityState* is now *Unchanged*. Attaching a detached entity always changes *EntityState* to *Unchanged* even if the entity was changed while detached. The customer is modified, and a pop-up shows its *EntityState* property as *Modified*. Finally, the *SaveChanges* method is called on the *ObjectContext*, and a pop-up shows the customer was saved. Another pop-up shows that *EntityState* is back to *Unchanged*, which implies that the state of the customer is in sync with the database.

When working with POCO, you must call the *DetectChanges* method on the *ObjectContext* to attach the POCO entity to the *ObjectContext*. Be sure to call *DetectChanges* prior to calling *SaveChanges*.



EXAM TIP

For the exam, expect to be tested on saving POCO entities. Remember that you must call the *DetectChanges* method on the *ObjectContext*.

Deleting Entities

When it's time to delete an entity, call the *DeleteObject* method of the *ObjectSet* property or the *DeleteObject* method of the *ObjectContext* to mark the specified entity for deletion. The row is not deleted from the database until *SaveChanges* is called. The following code sample shows the deletion of an *Order_Detail* object.

Sample of Visual Basic Code

```

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

```



```

        Handles DeleteToolStripMenuItem.Click
Dim db = New NorthwindEntities()
Dim itemToDelete = db.Order_Details.First()
MessageBox.Show(itemToDelete.EntityState.ToString())
db.Order_Details.DeleteObject(itemToDelete)
MessageBox.Show(itemToDelete.EntityState.ToString())
db.SaveChanges()
MessageBox.Show("Order Detail Deleted")
MessageBox.Show(itemToDelete.EntityState.ToString())
End Sub

```

Sample of C# Code

```

private void deleteToolStripMenuItem_Click(
    object sender, EventArgs e)
{
    var db = new NorthwindEntities();
    var itemToDelete = db.Order_Details.First();
    MessageBox.Show(itemToDelete.EntityState.ToString());
    db.Order_Details.DeleteObject(itemToDelete);
    MessageBox.Show(itemToDelete.EntityState.ToString());
    db.SaveChanges();
    MessageBox.Show("Order Detail Deleted");
    MessageBox.Show(itemToDelete.EntityState.ToString());
}

```

After *SaveChanges* is called, if you still hold a reference to the deleted entity, its *EntityState* will be *Detached*. You should not reuse the entity, because if you attach the entity, its *EntityState* will be *Unchanged*, and if you then modify the entity and call *SaveChanges*, an attempt to modify a nonexistent row will throw an *OptimisticConcurrencyException*.

Cascading Deletes

When an entity has associations to other entities, and you are trying to delete the entity, consider the following factors.

If the entity you deleted has dependent entities (foreign key association), and the foreign key on the dependent entities is nullable, the Entity Framework sets the foreign key of the dependent entities to *null* when the principal entity is deleted.

If a primary key of the principal entity is part of the primary key of the dependent entity, deleting the principal entity also deletes all the loaded dependent entities, which is essentially an implicit cascading delete for which you don't need to define a cascading delete rule on the relationship.

You can also set up a delete rule in your EDMX file. In the Entity Data Model designer, click the desired relationship and, in the Properties window, locate the *1* or *0..1* multiplicity end of the relationship. Set the *Delete* property to *Cascade* as shown in Figure 6-27, in which a cascading delete is being configured on the relationship between *Customers* and *Orders*.

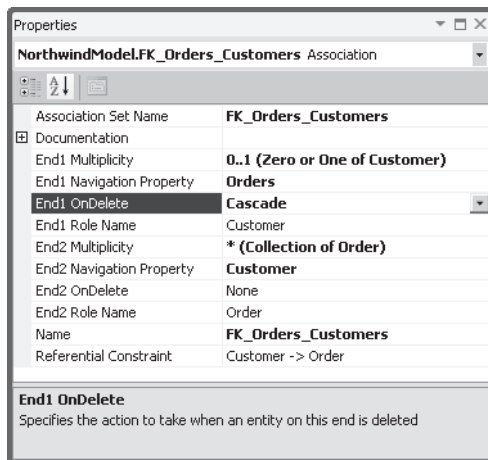


FIGURE 6-27 A Cascading delete is configured on the 1 or 0..1 multiplicity end.

It's important to note that a cascading delete in the Entity Framework works only if the dependent entity objects are loaded. This can be accomplished using the *Include* or *Load* methods. If you don't load the dependent entities, an *UpdateException* will be thrown. Also, don't forget to load dependents of dependents and so on. For example, if you set up *Customers* to cascade the deletion of *Orders*, deleting a customer will fail if you haven't loaded the dependent *Orders* and the dependent *Order_Details* as well.

To demonstrate a cascading delete, one has been configured for the relationship between *Customers* and *Orders*, as shown in Figure 6-27, and the following code sample will delete the first customer, which will delete the customer's orders, which will delete the orders' *Order_Details* values.

Sample of Visual Basic Code

```
Private Sub CascadeDeleteToolStripMenuItem_Click( _
    ByVal sender As System.Object, ByVal e As System.EventArgs) _
    Handles CascadeDeleteToolStripMenuItem.Click
    Dim db = New NorthwindEntities()
    Dim itemToDelete = db.Customers _
        .Include("Orders") _
        .Include("Orders.Order_Details") _
        .First()
    db.Customers.DeleteObject(itemToDelete)
    db.SaveChanges()
    MessageBox.Show("Customer Deleted!")
End Sub
```

Sample of C# Code

```
private void cascadingDeleteToolStripMenuItem_Click(
    object sender, EventArgs e)
{
    var db = new NorthwindEntities();
    var itemToDelete = db.Customers
```

```

        .Include("Orders")
        .Include("Orders.Order_Details")
        .First();
db.Customers.DeleteObject(itemToDelete);
db.SaveChanges();
MessageBox.Show("Customer Deleted!");
}

```

Notice that the *Include* method loads the *Order* objects for each *Customer* by passing "Orders" to the *Include* method. The *Include* method is used again to load *Order_Details* for each *Customer* by passing "Orders.Order_Details" to the *Include* method.

Using Stored Procedures

You've learned that the Entity Framework generates a class that inherits from *ObjectContext* to represent the entity container in the conceptual model. In most of the previous examples, this is the *NorthwindEntities* class. You've also learned that *ObjectContext* exposes a *SaveChanges* method that sends updates to the underlying database. So far, these updates have been using SQL statements that are automatically generated by the system, but you can develop and use custom stored procedures instead. Your application code will stay the same; you're simply remapping the update operations to use stored procedures instead of dynamic SQL.

When you decide to use stored procedures, you must map all three of the insert, update, and delete operations of an entity to stored procedures. If you don't map all three operations, the unmapped operations will fail when executed at run time, and an *UpdateException* will be thrown.

To map the insert, update, and delete operations to stored procedures, open your EDMX file. Right-click the desired entity and click Stored Procedure Mapping. This opens the Mapping Details window in Function mapping view. After you select the stored procedure to map to one of the operations (insert, update, or delete), you'll see the parameter list by which you can map entity properties to each property. For each mapping, you can choose whether to use the current value or the original value.

Submitting Changes in a Transaction

When working with the Entity Framework, you might wonder what happens when you execute the *SaveChanges* method and an exception is thrown after half of the changes have already been sent to the database.

By default, the Entity Framework automatically sends all changes back to the database in the context of a transaction, so if an exception is thrown, all changes are rolled back. In some scenarios, you need more control over the transaction. The most common scenario is when you have multiple *ObjectContext* objects, and they all need to participate in a transaction. Why multiple *ObjectContext* objects? One *ObjectContext* object accesses mainframe data while the other *ObjectContext* object is accessing SQL Server data. Alternately, maybe your code was written to create the *ObjectContext* object in a just-in-time fashion, so the

method creates an *ObjectContext* object for its operation, and then the method creates a new *ObjectContext* object for its operation. This scenario is addressed in the following sample code.

In your application, you might have an *addToInventory* method that accepts a *productId* parameter and a *quantity* parameter. This method creates an *ObjectContext* object, retrieves the product, and adds to its *UnitsInStock* property based on the quantity passed to the method. You might have a matching *removeFromInventory* method that does the same thing except that it subtracts from *UnitsInStock*. Also, a helper *displayUnitsInStock* method enables you to easily display the current value of the *UnitsInStock* property. The following sample code shows these methods.

Sample of Visual Basic Code

```
Private Sub displayUnitsInStock(ByVal productId As Integer)
    Dim db = New NorthwindEntities()
    Dim Product = db.Products _
        .Where(Function(p) p.ProductID = productId) _
        .First()
    MessageBox.Show(String.Format( _
        "Product ID: {0} UnitsInStock: {1}", _
        productId, Product.UnitsInStock))
End Sub

Private Sub addToInventory( _
    ByVal productId As Integer, ByVal quantity As Short)
    Dim db = New NorthwindEntities()
    Dim Product = db.Products _
        .Where(Function(p) p.ProductID = productId) _
        .First()
    Product.UnitsInStock += quantity
    db.SaveChanges()
End Sub

Private Sub removeFromInventory( _
    ByVal productId As Integer, ByVal quantity As Short)
    Dim db = New NorthwindEntities()
    Dim Product = db.Products _
        .Where(Function(p) p.ProductID = productId) _
        .First()
    Product.UnitsInStock -= quantity
    db.SaveChanges()
End Sub
```

Sample of C# Code

```
private void displayUnitsInStock(int productId )
{
    var db = new NorthwindEntities();
    var product = db.Products
        .Where(p => p.ProductID == productId)
        .First();
    MessageBox.Show(String.Format(
        "Product ID: {0} UnitsInStock: {1}",
```

```

        productId, Product.UnitsInStock));
    }

    private void addToInventory(int productId, short quantity)
    {
        var db = new NorthwindEntities();
        var product = db.Products
            .Where(p => p.ProductID == productId)
            .First();
        product.UnitsInStock += quantity;
        db.SaveChanges();
    }

    private void removeFromInventory(int productId, short quantity)
    {
        var db = new NorthwindEntities();
        var product = db.Products
            .Where(p => p.ProductID == productId)
            .First();
        product.UnitsInStock -= quantity;
        db.SaveChanges();
    }
}

```

Each of these methods modifies *UnitsInStock* in the context of its own transaction, but you want to add a new method, *transferInventory*, that accepts *fromProductId* and *toProductId* parameters and a quantity. This enables a customer to exchange one product for another. You want this operation to be treated as a transaction so you can use the *TransactionScope* class to wrap the complete transfer in a transaction. The *TransactionScope* class was covered in more detail in Chapter 2.

In the following code sample, a *transferInventory* method makes the call to the *addToInventory* and *removeFromInventory* methods to perform the transaction. These calls are wrapped in a *TransactionScope* object, which is in a *try/catch* block. For the purpose of the demonstration, *transferInventory* will also accept a Boolean value to indicate whether a failure should be simulated.

Sample of Visual Basic Code

```

Private Sub FailToolStripMenuItem_Click( _
    ByVal sender As System.Object, ByVal e As System.EventArgs) _
    Handles FailToolStripMenuItem.Click
    transferInventory(1, 2, 1, True)
End Sub

Private Sub SuccessToolStripMenuItem_Click( _
    ByVal sender As System.Object, ByVal e As System.EventArgs) _
    Handles SuccessToolStripMenuItem.Click
    transferInventory(1, 2, 1, False)
End Sub

Private Sub transferInventory( _
    ByVal fromProductId As Integer, _
    ByVal toProductId As Integer, _

```

```

        ByVal quantity As Integer, _
        ByVal causeError As Boolean)

    displayUnitsInStock(fromProductId)
    displayUnitsInStock(toProductId)

    Try
        Using tran As New TransactionScope()
            removeFromInventory(fromProductId, quantity)
            addToInventory(toProductId, quantity)
            If causeError Then
                Throw New EntitySqlException("Simulated Exception")
            End If
            tran.Complete()
            MessageBox.Show("Success")
        End Using
    Catch ex As Exception
        MessageBox.Show(ex.Message)
    End Try

    displayUnitsInStock(fromProductId)
    displayUnitsInStock(toProductId)
End Sub

```

Sample of C# Code

```

private void failToolStripMenuItem_Click(object sender, EventArgs e)
{
    transferInventory(1,2,1,true);
}

private void successToolStripMenuItem_Click(object sender, EventArgs e)
{
    transferInventory(1, 2, 1, false);
}

private void transferInventory(
    int fromProductId,
    int toProductId,
    short quantity,
    bool causeError)
{
    displayUnitsInStock(fromProductId);
    displayUnitsInStock(toProductId);

    try
    {
        using (var tran = new TransactionScope())
        {
            removeFromInventory(fromProductId, quantity);
            addToInventory(toProductId, quantity);
            if(causeError)
            {
                throw new EntitySqlException("Simulated Exception");
            }
        }
    }
}

```

```

        }
        tran.Complete();
        MessageBox.Show("Success");
    }
}
catch (Exception ex)
{
    MessageBox.Show(ex.Message);
}

displayUnitsInStock(fromProductId);
displayUnitsInStock(toProductId);
}

```

Retrieve and Save Data

In this practice, you continue the design of an application and database for tracking your music. You want to track your music by album, song name, artist, and genre. You use the Code First model to design the conceptual model and then generate the database from the conceptual model.

The previous practice generated the data-access layer by using the Entity Framework designer, and the result of that practice is the starting point for this practice.

In this practice, you create the graphical user interface (GUI) and provide code to retrieve data from and save data to the database.

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

EXERCISE Creating the GUI

In this exercise, you modify the WPF application you created in Lesson 1 by creating the GUI.

1. In Visual Studio .NET 2010, select 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, double-click the MainWindow.xaml file to open it in the WPF Form Designer window.
3. In the *Window* tag, add a *Loaded* event handler. When prompted for a new event handler, double-click New EventHandler. Your XAML should look like the following.

Sample of Visual Basic XAML

```

<Window x:Class="MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Loaded="Window_Loaded"
        Title="MainWindow" Height="350" Width="525">
    <Grid>
    </Grid>
</Window>

```

Sample of C# XAML

```
<Window x:Class="MusicTracker.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Loaded="Window_Loaded"
        Title="MainWindow" Height="350" Width="525">
    <Grid>
    </Grid>
</Window>
```

4. The XAML contains a *Grid* definition. Inside the grid, add three row definitions. The first row should have its *Height* property set to *Auto*, and the last two rows should have their *Height* property set to *"*"*. Regardless of your programming language, your XAML for the grid should look like the following.

Sample of XAML

```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto" />
        <RowDefinition Height="*" />
        <RowDefinition Height="*" />
    </Grid.RowDefinitions>
</Grid>
```

5. In the XAML, before the end of the *Grid* tag name, add *Menu*. Inside the menu, add *MenuItem* elements for *Save*, called *mnuSave*, and for *Exit*, called *mnuExit*. After adding these items, double-click each *MenuItem* element to add the click event handler code. Your XAML should look like the following.

Sample of XAML

```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto" />
        <RowDefinition Height="*" />
        <RowDefinition Height="*" />
    </Grid.RowDefinitions>
    <Menu>
        <MenuItem Header="Save" Name="mnuSave" Click="mnuSave_Click" />
        <MenuItem Header="Exit" Name="mnuExit" Click="mnuExit_Click" />
    </Menu>
</Grid>
```

6. Add *Albums* to *Grid*. To display the Data Sources panel, select Data | Show Data Sources. The *MusicDatabaseEntitiesObjectContext* object is displayed, and it contains the *Albums* and *Songs* entity sets.
7. Drag *Albums* to the WPF designer surface and drop it in the second row. This adds resources and code to your project to wire the entity set to this data grid.
8. Resize *DataGrid* so it fits within the first row and then modify the *DataGrid* XAML code to clean up the sizing, hide the *Id* column, and set the *Name* column to take all

available space on the row. Your XAML for the data grid should look like the following, regardless of programming language.

Sample of XAML

```
<DataGrid AutoGenerateColumns="False" EnableRowVirtualization="True"
    ItemsSource="{Binding}" Name="AlbumsDataGrid" Margin="5"
    RowDetailsVisibilityMode="VisibleWhenSelected" Grid.Row="1">
    <DataGrid.Columns>
        <DataGridTextColumn x:Name="IdColumn" Binding="{Binding Path=Id}"
            Header="Id" Visibility="Hidden" />
        <DataGridTextColumn x:Name="NameColumn" Binding="{Binding Path=Name}"
            Header="Album Name" Width="*" />
    </DataGrid.Columns>
</DataGrid>
```

9. Add *Songs* for the current *Album* to the grid. (Don't use the *Songs* entity set directly under *MusicDatabaseEntities*.)
10. Click the plus sign beside Albums; Songs appears. This represents the songs an album contains. Drag that *Songs* entity set to the third row of the grid.
11. Resize *DataGrid* to fit within the bottom row. Modify the XAML to clean up the resizing, make the Id column invisible, and set the Name column to take all available space on the row. Your XAML for this data grid should look like the following, based on programming language.

Sample of Visual Basic XAML

```
<DataGrid AutoGenerateColumns="False" EnableRowVirtualization="True"
    Grid.Row="2" Margin="5"
    ItemsSource="{Binding Source={StaticResource AlbumsSongsViewSource}}"
    Name="SongsDataGrid" RowDetailsVisibilityMode="VisibleWhenSelected" >
    <DataGrid.Columns>
        <DataGridTextColumn x:Name="IdColumn1" Binding="{Binding Path=Id}"
            Header="Id" Visibility="Hidden" />
        <DataGridTextColumn x:Name="NameColumn1" Binding="{Binding Path=Name}"
            Header="Song Name" Width="*" />
    </DataGrid.Columns>
</DataGrid>
```

Sample of C# XAML

```
<DataGrid AutoGenerateColumns="False" EnableRowVirtualization="True"
    Grid.Row="2" Margin="5"
    ItemsSource="{Binding Source={StaticResource albumsSongsViewSource}}"
    Name="SongsDataGrid" RowDetailsVisibilityMode="VisibleWhenSelected" >
    <DataGrid.Columns>
        <DataGridTextColumn x:Name="IdColumn1" Binding="{Binding Path=Id}"
            Header="Id" Visibility="Hidden" />
        <DataGridTextColumn x:Name="NameColumn1" Binding="{Binding Path=Name}"
            Header="Song Name" Width="*" />
    </DataGrid.Columns>
</DataGrid>
```

A *Windows.Resources* element contains the resource information to which your *DataGrid* controls are binding. The XAML is a bit different between programming languages, so here is the completed XAML, based on programming language.

Sample of Visual Basic XAML

```
<Window x:Class="MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Loaded="Window_Loaded"
    Title="MainWindow" Height="350" Width="525" mc:Ignorable="d"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:my="clr-namespace:MusicTracker">
    <Window.Resources>
        <CollectionViewSource x:Key="AlbumsViewSource" d:DesignSource=
            "{d:DesignInstance my:Album, CreateList=True}" />
        <CollectionViewSource x:Key="AlbumsSongsViewSource" Source=
            "{Binding Path=Songs, Source={StaticResource AlbumsViewSource}}" />
    </Window.Resources>
    <Grid DataContext="{StaticResource AlbumsViewSource}">
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto" />
            <RowDefinition Height="*" />
            <RowDefinition Height="*" />
        </Grid.RowDefinitions>
        <Menu>
            <MenuItem Header="Save" Name="mnuSave" Click="mnuSave_Click" />
            <MenuItem Header="Exit" Name="mnuExit" Click="mnuExit_Click" />
        </Menu>
        <DataGrid AutoGenerateColumns="False" EnableRowVirtualization="True"
            ItemsSource="{Binding}" Name="AlbumsDataGrid" Margin="5"
            RowDetailsVisibilityMode="VisibleWhenSelected" Grid.Row="1">
            <DataGrid.Columns>
                <DataGridTextColumn x:Name="IdColumn"
                    Binding="{Binding Path=Id}"
                    Header="Id" Visibility="Hidden" />
                <DataGridTextColumn x:Name="NameColumn"
                    Binding="{Binding Path=Name}"
                    Header="Album Name" Width="*" />
            </DataGrid.Columns>
        </DataGrid>
        <DataGrid AutoGenerateColumns="False" EnableRowVirtualization="True"
            Grid.Row="2" Margin="5" ItemsSource=
            "{Binding Source={StaticResource AlbumsSongsViewSource}}"
            Name="SongsDataGrid"
            RowDetailsVisibilityMode="VisibleWhenSelected" >
            <DataGrid.Columns>
                <DataGridTextColumn x:Name="IdColumn1"
                    Binding="{Binding Path=Id}"
                    Header="Id" Visibility="Hidden" />
                <DataGridTextColumn x:Name="NameColumn1"
                    Binding="{Binding Path=Name}"
                    Header="Song Name" Width="*" />
            </DataGrid.Columns>
    </Grid>
```

```

        </DataGrid>
    </Grid>
</Window>

```

Sample of C# XAML

```

<Window x:Class="MusicTracker.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Loaded="Window_Loaded"
        Title="MainWindow" Height="350" Width="525" mc:Ignorable="d"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:my="clr-namespace:MusicTracker">
    <Window.Resources>
        <CollectionViewSource x:Key="albumsViewSource" d:DesignSource=
            "{d:DesignInstance my:Album, CreateList=True}" />
        <CollectionViewSource x:Key="albumsSongsViewSource" Source=
            "{Binding Path=Songs, Source={StaticResource albumsViewSource}}" />
    </Window.Resources>
    <Grid DataContext="{StaticResource albumsViewSource}">
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto" />
            <RowDefinition Height="*" />
            <RowDefinition Height="*" />
        </Grid.RowDefinitions>
        <Menu>
            <MenuItem Header="Save" Name="mnuSave" Click="mnuSave_Click" />
            <MenuItem Header="Exit" Name="mnuExit" Click="mnuExit_Click" />
        </Menu>
        <DataGrid AutoGenerateColumns="False" EnableRowVirtualization="True"
            ItemsSource="{Binding}" Name="AlbumsDataGrid" Margin="5"
            RowDetailsVisibilityMode="VisibleWhenSelected" Grid.Row="1">
            <DataGrid.Columns>
                <DataGridTextColumn x:Name="IdColumn" Binding="{Binding Path=Id}"
                    Header="Id" Visibility="Hidden" />
                <DataGridTextColumn x:Name="NameColumn"
                    Binding="{Binding Path=Name}"
                    Header="Album Name" Width="*" />
            </DataGrid.Columns>
        </DataGrid>
        <DataGrid AutoGenerateColumns="False" EnableRowVirtualization="True"
            Grid.Row="2" Margin="5"
            ItemsSource=
                "{Binding Source={StaticResource albumsSongsViewSource}}"
            Name="SongsDataGrid"
            RowDetailsVisibilityMode="VisibleWhenSelected" >
            <DataGrid.Columns>
                <DataGridTextColumn x:Name="IdColumn1"
                    Binding="{Binding Path=Id}"
                    Header="Id" Visibility="Hidden" />
                <DataGridTextColumn x:Name="NameColumn1"
                    Binding="{Binding Path=Name}"
                    Header="Song Name" Width="*" />
            </DataGrid.Columns>
        </DataGrid>
    </Grid>

```

```

        </Grid>
    </Window>

```

12. Double-click the Save menu item. This takes you to the code-behind file. Code is already in this file from dropping Albums and Songs on the XAML designer surface. *Window_Loaded* contains code to instantiate *MusicDatabaseEntities*, but you need access to that object from within the *mnuSave_Click* method, so cut that line and paste it just inside the class. Your code should look like the following.

Sample of Visual Basic Code

```

Class MainWindow
    Dim MusicDatabaseEntities As MusicTracker.MusicDatabaseEntities = _
        New MusicTracker.MusicDatabaseEntities()

    Private Sub Window_Loaded(ByVal sender As System.Object, _
        ByVal e As System.Windows.RoutedEventArgs)

        'Load data into Albums. You can modify this code as needed.
        Dim AlbumsViewSource As System.Windows.Data.CollectionViewSource = _
            CType(Me.FindResource("AlbumsViewSource"), _
                System.Windows.Data.CollectionViewSource)
        Dim AlbumsQuery As System.Data.Objects.ObjectQuery( _
            Of MusicTracker.Album) _
            = Me.GetAlbumsQuery(MusicDatabaseEntities)
        AlbumsViewSource.Source = AlbumsQuery.Execute( _
            System.Data.Objects.MergeOption.AppendOnly)
    End Sub

    Private Sub mnuSave_Click( _
        ByVal sender As System.Object, _
        ByVal e As System.Windows.RoutedEventArgs)

    End Sub

    Private Sub mnuExit_Click( _
        ByVal sender As System.Object, _
        ByVal e As System.Windows.RoutedEventArgs)

    End Sub

    Private Function GetAlbumsQuery( _
        ByVal MusicDatabaseEntities As MusicTracker.MusicDatabaseEntities) _
        As System.Data.Objects.ObjectQuery(Of MusicTracker.Album)

        Dim AlbumsQuery As System.Data.Objects.ObjectQuery( _
            Of MusicTracker.Album) = MusicDatabaseEntities.Albums
        'Update the query to include Songs data in Albums.
        'You can modify this code as needed.
        AlbumsQuery = AlbumsQuery.Include("Songs")
        'Returns an ObjectQuery.
        Return AlbumsQuery
    End Function
End Class

```

Sample of C# Code

```
using System;
using System.Collections.Generic;
using System.Collections.ObjectModel;
using System.Linq;
using System.Text;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;

namespace MusicTracker
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        MusicTracker.MusicDatabaseEntities musicDatabaseEntities =
            new MusicTracker.MusicDatabaseEntities();

        public MainWindow()
        {
            InitializeComponent();
        }

        private void Window_Loaded(object sender, RoutedEventArgs e)
        {
            // Load data into Albums. You can modify this code as needed.
            System.Windows.Data.CollectionViewSource albumsViewSource =
                ((System.Windows.Data.CollectionViewSource)
                (this.FindResource("albumsViewSource")));
            System.Data.Objects.ObjectQuery<MusicTracker.Album> albumsQuery =
                this.GetAlbumsQuery(musicDatabaseEntities);
            albumsViewSource.Source =
                albumsQuery.Execute(System.Data.Objects.MergeOption.AppendOnly);
        }

        private void mnuSave_Click(object sender, RoutedEventArgs e)
        {
        }

        private void mnuExit_Click(object sender, RoutedEventArgs e)
        {
        }

        private System.Data.Objects.ObjectQuery<Album> GetAlbumsQuery(
```

```

        MusicDatabaseEntities musicDatabaseEntities)
    {
        // Auto generated code

        System.Data.Objects.ObjectQuery<MusicTracker.Album> albumsQuery =
            musicDatabaseEntities.Albums;
        // Update the query to include Songs data in Albums.
        //You can modify this code as needed.
        albumsQuery = albumsQuery.Include("Songs");
        // Returns an ObjectQuery.
        return albumsQuery;
    }
}

```

- 13.** Add the following code to the *mnuSave_Click* method to save all changes to the database and display a saved message or an error message.

Sample of Visual Basic Code

```

Private Sub mnuSave_Click( _
    ByVal sender As System.Object, _
    ByVal e As System.Windows.RoutedEventArgs)

    Try
        MusicDatabaseEntities.SaveChanges()
        MessageBox.Show("Saved")
    Catch ex As Exception
        MessageBox.Show(ex.Message)
    End Try
End Sub

```

Sample of C# Code

```

private void mnuSave_Click(object sender, RoutedEventArgs e)
{
    try
    {
        musicDatabaseEntities.SaveChanges();
        MessageBox.Show("Saved");
    }
    catch (Exception ex)
    {
        MessageBox.Show(ex.Message);
    }
}

```

- 14.** Add code to the *mnuExit_Click* method to exit the application. The exit method first saves all changes and then exits. If an error is thrown during the save operation, display the error but continue exiting the application. Your code should look like the following.

Sample of Visual Basic Code

```

Private Sub mnuExit_Click( _
    ByVal sender As System.Object, _
    ByVal e As System.Windows.RoutedEventArgs)

```

```

Try
    MusicDatabaseEntities.SaveChanges()
Catch ex As Exception
    MessageBox.Show(ex.Message)
End Try
Application.Current.Shutdown()
End Sub

```

Sample of C# Code

```

private void mnuExit_Click(object sender, RoutedEventArgs e)
{
    try
    {
        musicDatabaseEntities.SaveChanges();
    }
    catch (Exception ex)
    {
        MessageBox.Show(ex.Message);
    }
    Application.Current.Shutdown();
}

```

15. Press F5 to run this application. Your main window should look the one in Figure 6-28. You should be able to enter albums by simply typing the album name and pressing Enter. As you add albums, you might want to add songs; simply type each song name and press Enter.

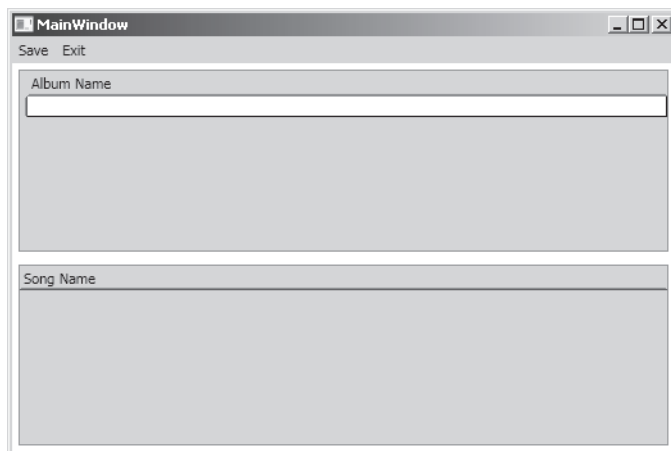


FIGURE 6-28 This is the completed GUI.

Lesson Summary

This lesson provided detailed information about querying the Entity Frameworks and covered LINQ to Entities and ESQL querying.

- LINQ to Entities queries are converted to command trees that are executed against *ObjectContext* objects.
- You use *ObjectQuery* to compose a LINQ to Entities query.
- LINQ to Entities provides deferred execution of queries.
- *Materialization* is the creation of objects that represent the result.
- Not all implementations of LINQ operators are supported in LINQ to Entities, due to the varying types of data stores that LINQ to Entities supports.
- Use the *AsEnumerable* query extension method to cast to the generic *IEnumerable* interface, which bypasses LINQ to Entities, but code executes client-side.
- Instead of using *AsEnumerable*, you can implement a stored procedure that executes server-side and might perform better to solve a problem .
- ESQL provides an SQL-like syntax for building queries that are database independent.
- One of the biggest benefits of using ESQL is the ability to create dynamic queries that target any supported database.
- ESQL queries can target Object Services and the EntityClient data provider.
- *ObjectContext* can send changes back to the database.
- The *SaveChanges* method on the *ObjectContext* object locates all changed objects and sends the appropriate *Insert*, *Update*, or *Delete* command to the database.
- An entity has an *EntityState* property that can be set to *Detached*, *Unchanged*, *Modified*, *Added*, or *Deleted*.
- If an existing entity (entity that has a primary key) is detached, you can use the *Attach* method to attach the entity to an *ObjectContext* object.
- Deleting an entity simply marks the entity for deletion. The entity is not deleted until you call the *SaveChanges* method on the *ObjectContext* object.
- You can configure a relationship to perform cascading deletes by setting the *1* or *1..0* end of a relationship to *Cascade*.
- You can map stored procedures to insert, delete, and change operations.
- *ObjectContext* automatically sends the updates within a transaction context.
- You can wrap operations that use multiple *ObjectContext* objects within a *TransactionScope* class.

Lesson Review

You can use the following questions to test your knowledge of the information in Lesson 2, “Querying and Updating with the Entity Framework.” 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. When working with the Entity Framework, which method do you execute to send changes back to the database?
 - A. *SubmitChanges*
 - B. *SaveChanges*
2. You are writing an Entity SQL query that requires you to construct a row from various data. Which function will you use to construct the row?
 - A. *CREATEREF*
 - B. *MULTISET*
 - C. *ROW*
3. You have just set up the Entity Framework cascading delete, and you are now trying to use it, but you keep getting an *UpdateException*. What is the most logical cause of this exception?
 - A. You must make sure that all dependent objects are loaded.
 - B. Your database does not support cascading deletes.
 - C. You have not defined a cascading delete at the database server.
4. You are working with an *ObjectContext* object that targets the mainframe and another *ObjectContext* object that targets SQL Server. When it's time to save the changes, you want all changes to be sent to the mainframe and to SQL Server as one transaction. How can you accomplish this?
 - A. Just save both *ObjectContext* objects because they automatically join the same transaction.
 - B. Save to the mainframe and use an *if* statement to verify that the changes were successful. If successful, save to SQL Server.
 - C. Wrap the saving of both *ObjectContext* objects within a *TransactionScope* object that is implemented in a *using* statement in which the last line executes the *Complete* method on the *TransactionScope* class.
 - D. Use a Boolean flag to indicate the success of each save, which will tell you whether the save was successful.

Case Scenarios

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

Case Scenario 1: Choosing an Object-Relational Mapper

You have been assigned the task of choosing an object-relational mapper, and your choices are LINQ to SQL or LINQ to Entities. The application will be using SQL Server 2008 R2, and the database will have a small number of tables (fewer than 10) but will also occasionally need to send transactions to an existing Oracle server. Your team will be designing the database while the application is being created. Your application takes advantage of inheritance and polymorphism wherever possible.

1. Does anything in this scenario force you to choose a particular ORM?
2. Which ORM would you choose?
3. What are the reasons for your choice?

Case Scenario 2: Using the Entity Framework

You are creating a new application, but you have to connect to an existing database. However, your team has no control over that schema. Your team wants to work in an object-centric environment in which the focus is on the domain model, not on the data model. Answer the following questions regarding this application and the use of the Entity Framework.

1. Can you describe a feature of the Entity Framework that will help you when the database already exists?
2. Can you describe a feature of the Entity Framework that will help you when you have no control of the database schema?
3. Can you describe a feature of the Entity Framework that will help you when you see that different kinds of employees (managers, salespeople) have additional data in other tables?

Suggested Practices

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

Create an Application That Uses LINQ to Entities Queries

Create at least one application that uses LINQ to Entities. This can be accomplished by performing the practices at the ends of Lesson 1 and Lesson 2, or you can complete the following Practice 1.

- **Practice 1** Create an application that requires you to query a database for data from at least two tables by using LINQ to Entities when the tables are related. This could be movies that have actors, artists who record music, or people who have vehicles. Add LINQ to Entities queries to search and filter the data.
- **Practice 2** Complete Practice 1 and then add LINQ to Entities queries that join the tables. Be sure to provide both inner and outer joins. Also, add queries that perform grouping and aggregates. Try at least one Entity SQL query.

Create an Application That Modifies Data by Using LINQ to Entities

Create at least one application that uses LINQ to Entities to modify data in the database. This can be accomplished by performing the practices at the ends of Lesson 1 and Lesson 2 or by completing the following practice.

- **Practice** Create an application that requires you to collect data into at least two database tables when the data tables are related. This could be movies that have actors, artists who record music, or people who have vehicles. Use LINQ to Entities to add, delete, and modify the data.

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.