

56

LINQ to SQL

WHAT'S IN THIS CHAPTER?

- Working with LINQ to SQL using Visual Studio 2010
- Mapping LINQ to SQL objects to database entities
- Building LINQ to SQL operations without the O/R Designer
- Using the O/R Designer with custom objects
- Querying the SQL Server database using LINQ
- Stored procedures and LINQ to SQL

You will probably find the .NET Language Integrated Query Framework (LINQ) in C# 2010 to be one of the more exciting features the language has to offer. Basically, what LINQ provides is a lightweight façade over programmatic data integration. This is such a big deal because *data is king*.

Pretty much every application deals with data in some manner, whether that data comes from memory (in-memory data), databases, XML files, text files, or something else. Many developers find it very difficult to move from the strongly typed object-oriented world of C# to the data tier where objects are second-class citizens. The transition from one world to the next was a kludge at best and was full of error-prone actions.

In C#, programming with objects means a wonderful, strongly typed ability to work with code. You can navigate very easily through the namespaces, work with a debugger in the Visual Studio IDE, and more. However, when you have to access data, you will notice that things are dramatically different.

You end up in a world that is not strongly typed, where debugging is a pain or even non-existent, and you end up spending most of the time sending strings to the database as commands. As a developer, you also have to be aware of the underlying data and how it is structured or how all the data points relate.

LINQ provides a strongly typed interface to the underlying data stores. It provides the means for developers to stay within the coding environment that they are used to and access the underlying data as objects that work with the IDE, IntelliSense, and even debugging.

With LINQ, the queries that you create now become first-class citizens within the .NET Framework alongside everything else you are used to. When you work with queries for your data store, you quickly realize that they now work and behave as if they are types in the system. This means that you can now use any .NET-compliant language and query the underlying data store as you never have before.



Chapter 11, “Language Integrated Query,” provides an introduction to LINQ.

Figure 56-1 shows LINQ’s place in querying data.

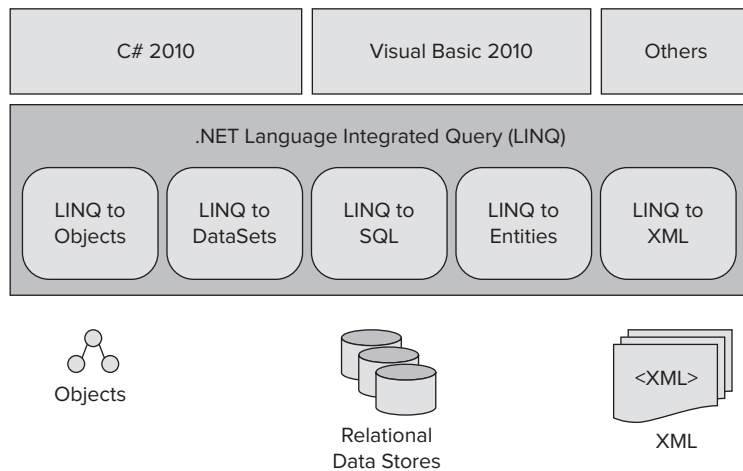


FIGURE 56-1

Looking at the figure, you can see that there are different types of LINQ capabilities, depending on the underlying data that you are going to work with in your application. From the list, you find the following LINQ technologies:

- LINQ to Objects
- LINQ to DataSets
- LINQ to SQL
- LINQ to Entities
- LINQ to XML

As a developer, you are given class libraries that provide objects that, using LINQ, can be queried as any other data store can. Objects are really nothing more than data that is stored in memory. In fact, your objects themselves might be querying data. This is where LINQ to Objects comes into play.

LINQ to SQL (the focus of this chapter), LINQ to Entities, and LINQ to DataSets provide the means to query relational data. Using LINQ, you can query directly against your database and even against the stored procedures that your database exposes. The last item from the diagram is the ability to query against your XML using LINQ to XML (this topic is covered in Chapter 33, “Manipulating XML”). The big thing that makes LINQ exciting is that it matters very little what you are querying against, because your queries will be quite similar.

LINQ TO SQL USING VISUAL STUDIO 2010

LINQ to SQL in particular is a means to have a strongly typed interface against a SQL Server database. You will find the approach that LINQ to SQL provides is by far the easiest approach to querying SQL Server available at the moment. It is not just about querying single tables within the database. For instance, if you call the `Customers` table of the Northwind database and want to pull a customer's specific orders from the `Orders` table in the same database, LINQ will use the relations of the tables and make the query on your behalf. LINQ will query the database and load up the data for you to work with from your code (again, strongly typed).

It is important to remember that LINQ to SQL is not only about querying data, but you are also able to perform the Insert/Update/Delete statements needed.

You can also interact with the entire process and customize the operations performed to add your own business logic to any of the CRUD operations (Create/Read/Update/Delete).

Visual Studio 2010 comes into strong play with LINQ to SQL in that you will find an extensive user interface that allows you to design the LINQ to SQL classes you will work with.

The next section focuses on how to set up your first LINQ to SQL instance and pull items from the `Products` table of the Northwind database.

Calling the Products Table

For an example of using LINQ to SQL, this section starts by calling a single table from the Northwind database and using it to populate results to the screen.

To start off, create a console application (using .NET Framework 4) and add the Northwind database file to this project (`Northwind.MDF`).



The following example makes use of the `Northwind.mdf` SQL Server Express Database file. To get this database, search for "Northwind and pubs Sample Databases for SQL Server 2000." You can find this link at <http://www.microsoft.com/downloads/details.aspx?FamilyID=06616212-0356-46A0-8DA2-EEBC53A68034&displaylang=en>. When installed, you will find the `Northwind.mdf` file in the `C:\SQL Server 2000 Sample Databases` directory. To add this database to your application, right-click the solution you are working with and select `Add Existing Item`. From the provided dialog box, you are then able to browse to the location of the `Northwind.mdf` file that you just installed. If you are having trouble getting permissions to work with the database, make a data connection to the file from the Visual Studio Server Explorer and you will be asked to be made the appropriate user of the database. VS will make the appropriate changes on your behalf for this to occur.

By default now, when creating many of the application types provided in .NET Framework 4 within Visual Studio 2010, you will notice that you already have the proper references in place to work with LINQ. When creating a console application, you will get the following using statements in your code:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
```

From this, you can see that the LINQ reference required is already in place.

Adding a LINQ to SQL Class

The next step is to add a LINQ to SQL class. When working with LINQ to SQL, one of the big advantages you will find is that Visual Studio 2010 does an outstanding job of making it as easy as possible. Visual Studio provides an object-relational mapping designer, called the O/R Designer, which allows you to visually design the object to database mapping.

To start this task, right-click your solution and select Add New Item from the provided menu. From the items in the Add New Item dialog box, you will find LINQ to SQL Classes as an option. This is presented in Figure 56-2.

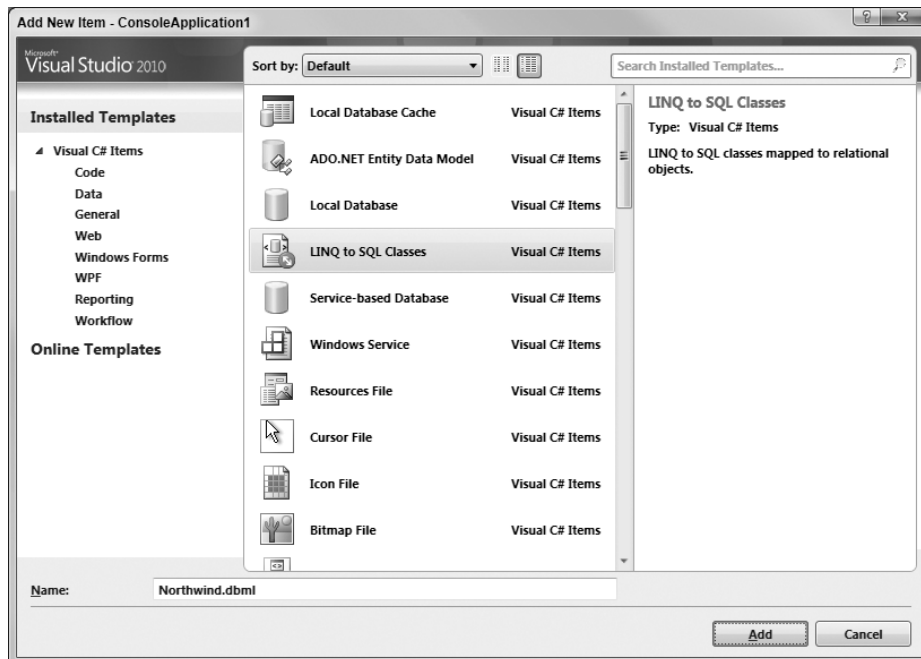


FIGURE 56-2

Because this example is using the Northwind database, name the file `Northwind.dbml`. Click the Add button, and you will see that this operation creates a couple of files for you. Figure 56-3 presents the Solution Explorer after adding the Northwind .dbml file.

A number of things were added to your project with this action. The `Northwind.dbml` file was added and it contains two components. Because the LINQ to SQL class that was added works with LINQ, the following references were also added on your behalf: `System.Core`, `System.Data.DataSetExtensions`, `System.Data.Linq`, and `System.Xml.Linq`.

Introducing the O/R Designer

Another big addition to the IDE that appeared when you added the LINQ to SQL class to your project (the `Northwind.dbml` file), was a visual

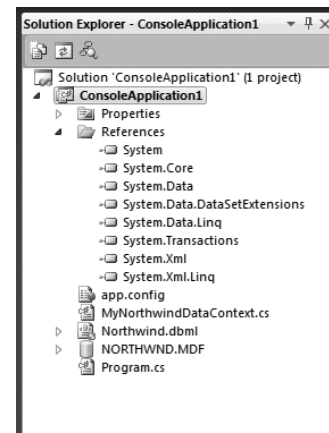


FIGURE 56-3

representation of the .dbml file. The new O/R Designer will appear as a tab within the document window directly in the IDE. Figure 56-4 shows a view of the O/R Designer when it is first initiated.

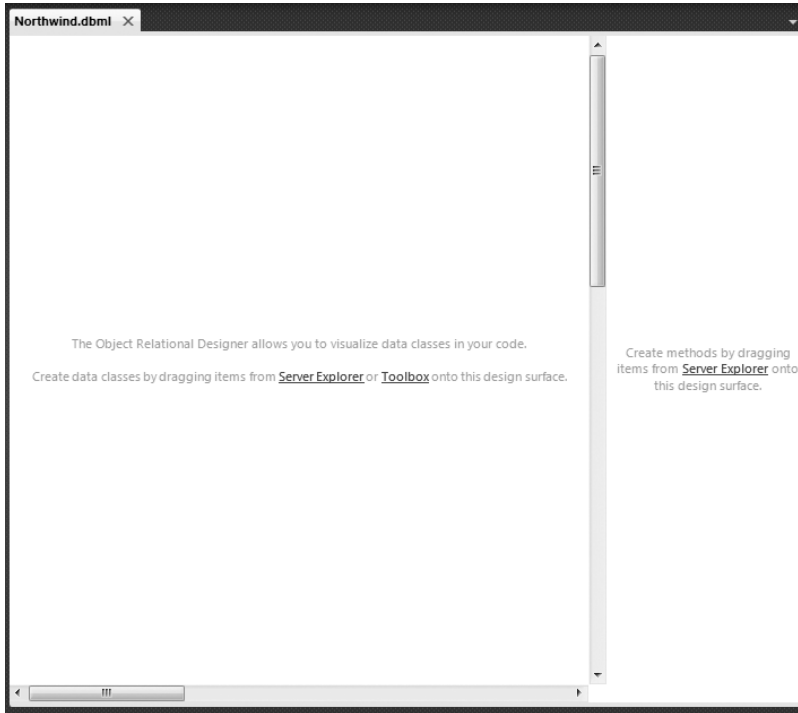


FIGURE 56-4

The O/R Designer is made up of two parts. The first part is for data classes, which can be tables, classes, associations, and inheritances. Dragging such items on this design surface will give you a visual representation of the object that can be worked with. The second part (on the right) is for methods, which map to the stored procedures within a database.

When viewing your .dbml file within the O/R Designer, you will also have an Object Relational Designer set of controls in the Visual Studio toolbox. The toolbox is presented in Figure 56-5.

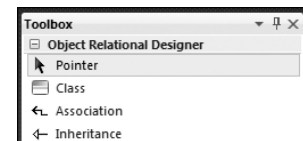


FIGURE 56-5

Creating the Product Object

For this example, you want to work with the `Products` table from the Northwind database, which means that you have to create a `Products` table that will use LINQ to SQL to map to this table. Accomplishing this task is simply a matter of opening up a view of the tables contained within the database from the Server Explorer dialog within Visual Studio and dragging and dropping the `Products` table onto the design surface of the O/R Designer. This action's results are illustrated in Figure 56-6.

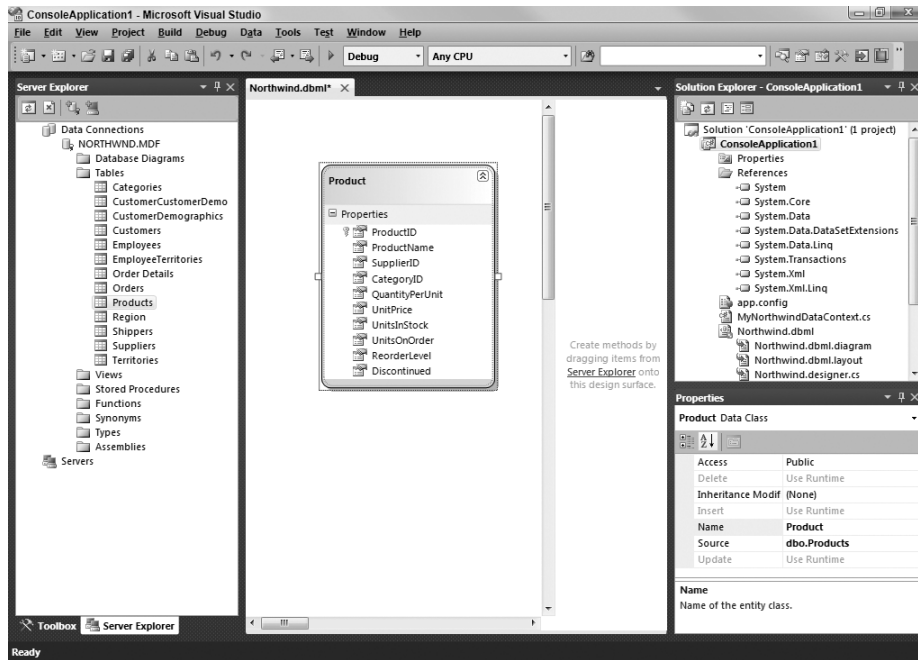


FIGURE 56-6

With this action, a bunch of code is added to the designer files of the .dbml file on your behalf. These classes will give you a strongly typed access to the `Products` table. For a demonstration of this, turn your attention to the console application's `Program.cs` file. The following shows the code that is required for this example:



```
using System;
using System.Linq;

namespace ConsoleApplication1
{
    class Class1
    {
        static void Main(string[] args)
        {
            NorthwindDataContext dc = new NorthwindDataContext();

            var query = dc.Products;

            foreach (Product item in query)
            {
                Console.WriteLine("{0} | {1} | {2}",
                    item.ProductID, item.ProductName, item.UnitsInStock);
            }

            Console.ReadLine();
        }
    }
}
```

code download ConsoleApplication1.sln

This bit of code does not have many lines to it, but it does query the `Products` table within the Northwind database and pulls out the data to display. It is important to step through this code starting with the first line in the `Main()` method:

```
NorthwindDataContext dc = new NorthwindDataContext();
```

The `NorthwindDataContext` object is an object of type `DataContext`. Basically, you can view this as something that maps to a `Connection` type object. This object works with the connection string and connects to the database for any required operations.

The next line is quite interesting:

```
var query = dc.Products;
```

Here, you are using the new `var` keyword, which is an implicitly typed variable. If you are unsure of the output type, you can use `var` instead of defining a type and the type will be set into place at compile time. Actually, the code `dc.Products;` returns a `System.Data.Linq.Table<ConsoleApplication1.Product>` object and this is what `var` is set to when the application is compiled. Therefore, this means that you could have also just as easily written the statement as such:

```
Table<Product> query = dc.Products;
```

This approach is actually better because programmers coming to look at the code of the application will find it easier to understand what is happening. Using the `var` keyword has so much of a hidden aspect to it that programmers might find it problematic. To use `Table<Product>`, which is basically a generic list of `Product` objects, you should make a reference to the `System.Data.Linq` namespace.

The value assigned to the `query` object is the value of the `Products` property, which is of type `Table<Product>`. From there, the next bit of code iterates through the collection of `Product` objects found in `Table<Product>`:

```
foreach (Product item in query)
{
    Console.WriteLine("{0} | {1} | {2}",
        item.ProductID, item.ProductName, item.UnitsInStock);
}
```

The iteration, in this case, pulls out the `ProductID`, `ProductName`, and `UnitsInStock` properties from the `Product` object and writes them out to the program. Because you are using only a few of the items from the table, you also have the option from the O/R Designer to delete the columns that you are not interested in pulling from the database. The results coming out from the program are presented here:

```
1 | Chai | 39
2 | Chang | 17
3 | Aniseed Syrup | 13
4 | Chef Anton's Cajun Seasoning | 53
5 | Chef Anton's Gumbo Mix | 0

** Results removed for space reasons **

73 | Röd Kaviar | 101
74 | Longlife Tofu | 4
75 | Rhönbräu Klosterbier | 125
76 | Lakalikööri | 57
77 | Original Frankfurter grüne Soße | 32
```

From this example, you can see just how easy it is to query a SQL Server database using LINQ to SQL.

HOW OBJECTS MAP TO LINQ OBJECTS

The great thing about LINQ is that it gives you strongly typed objects to use in your code (with IntelliSense) and these objects map to existing database objects. Again, LINQ is nothing more than a thin façade over these pre-existing database objects. The following table shows the mappings that are between the database objects and the LINQ objects.

DATABASE OBJECT	LINQ OBJECT
Database	DataContext
Table	Class and Collection
View	Class and Collection
Column	Property
Relationship	Nested Collection
Stored Procedure	Method

On the left side, you are dealing with your database. The database is the entire entity — the tables, views, triggers, stored procedures — everything that makes up the database. On the LINQ side of this, you have an object called the `DataContext` object. A `DataContext` object is bound to the database. For the required interaction with the database, it contains a connection string; it will manage all the transactions that occur, it will take care of any logging, and it will manage the output of the data. The `DataContext` object completely manages the transactions with the database on your behalf.

Tables, as you saw in the console application example, are converted to classes. This means that if you have a `Products` table, you will have a `Product` class. You will notice that LINQ is name-friendly in that it changes plural tables to singular to give the proper name to the class that you are using in your code. In addition to database tables being treated as classes, you will find that database views are also treated as the same. Columns, on the other hand, are treated as properties. This gives you the ability to manage the attributes (names and type definitions) of the column directly.

Relationships are nested collections that map between these various objects. This gives you the ability to define relationships that are mapped to multiple items.

It is also important to understand the mapping of stored procedures. These actually map to methods within your code from the `DataContext` instance. The next section takes a closer look at the `DataContext` and the table objects within LINQ.

When dealing with the architecture of LINQ to SQL, you will notice that there are really three layers to this — your application, the LINQ to SQL layer, and the SQL Server database. As you saw from the previous examples, you can create a strongly typed query in your application's code:

```
dc.Products;
```

This in turn gets translated to a SQL query by the LINQ to SQL layer, which is then supplied to the database on your behalf:

```
SELECT [t0].[ProductID], [t0].[ProductName], [t0].[SupplierID],
[t0].[CategoryID], [t0].[QuantityPerUnit], [t0].[UnitPrice],
[t0].[UnitsInStock], [t0].[UnitsOnOrder], [t0].[ReorderLevel],
[t0].[Discontinued]
FROM [dbo].[Products] AS [t0]
```

In return, the LINQ to SQL layer takes the rows coming out of the database from this query and turns the returned data into a collection of strongly typed objects that you can easily work with.

The DataContext Object

Again, the `DataContext` object manages the transactions that occur with the database that you are working with when working with LINQ to SQL. There is actually a lot that you can do with the `DataContext` object.

In instantiating one of these objects, you will notice that it takes a couple of optional parameters. These options include:

- A string that represents the location of the SQL Server Express database file or the name of the SQL Server that is used
- A connection string
- Another DataContext object

The first two string options also have the option of including your own database mapping file. After you have instantiated this object, you are then able to programmatically use it for many types of operations.

Using the ExecuteQuery Method

One of the simpler things that you can accomplish with the DataContext object is to run quick commands that you write yourself using the ExecuteQuery<T>() method. For instance, if you are going to pull all the products from the Products table using the ExecuteQuery<T>() method, your code would be similar to the following:



```
using System;
using System.Collections.Generic;
using System.Data.Linq;

namespace ConsoleApplication1
{
    class Class1
    {
        static void Main(string[] args)
        {
            DataContext dc = new DataContext(@"Data Source=.\\SQLEXPRESS;
            AttachDbFilename=|DataDirectory|\\NORTHWND.MDF;
            Integrated Security=True;User Instance=True");

            IEnumerable<Product> myProducts =
                dc.ExecuteQuery<Product>("SELECT * FROM PRODUCTS", "");

            foreach (Product item in myProducts)
            {
                Console.WriteLine(item.ProductID + " | " + item.ProductName);
            }

            Console.ReadLine();
        }
    }
}
```

[code download ConsoleApplication1.sln](#)

In this case, the ExecuteQuery<T>() method is called passing in a query string and returning a collection of Product objects. The query utilized in the method call is a simple Select statement that doesn't require any additional parameters to be passed in. Because there are no parameters passed in with the query, you can either use the double quotes as the second required parameter to the method call or not even provide that parameter. If you optionally substitute any values in the query, you construct your ExecuteQuery<T>() call as such:

```
IEnumerable<Product> myProducts =
    dc.ExecuteQuery<Product>("SELECT * FROM PRODUCTS WHERE UnitsInStock > {0}",
    50);
```

In this case, the {0} is a placeholder for the substituted parameter value that you are going to pass in, and the second parameter of the ExecuteQuery<T>() method is the parameter that will be used in the substitution.

Using the Connection Property

The `Connection` property actually returns an instance of the `System.Data.SqlClient.SqlConnection` that is used by the `DataContext` object. This is ideal if you need to share this connection with other ADO.NET code that you might be using in your application, or if you need to get at any of the `SqlConnection` properties or methods that it exposes. For instance, getting at the connection string is a simple affair:

```
NorthwindDataContext dc = new NorthwindDataContext();

Console.WriteLine(dc.Connection.ConnectionString);
```

Using an ADO.NET Transaction

If you have an ADO.NET transaction that you can use, you are able to assign that transaction to the `DataContext` object instance using the `Transaction` property. You can also make use of transactions using the `TransactionScope` object that is from the .NET 2.0 Framework (you will need to make a reference to the `System.Transactions` assembly):



```
using System;
using System.Collections.Generic;
using System.Data.Linq;
using System.Transactions;

namespace ConsoleApplication1
{
    class Class1
    {
        static void Main(string[] args)
        {
            NorthwindDataContext dc = new NorthwindDataContext();

            using (TransactionScope myScope = new TransactionScope())
            {
                Product p1 = new Product() { ProductName = "Bill's Product" };
                dc.Products.InsertOnSubmit(p1);

                Product p2 = new Product() { ProductName = "Another Product" };
                dc.Products.InsertOnSubmit(p2);

                try
                {
                    dc.SubmitChanges();

                    Console.WriteLine(p1.ProductID);
                    Console.WriteLine(p2.ProductID);
                }
                catch (Exception ex)
                {
                    Console.WriteLine(ex.ToString());
                }

                myScope.Complete();
            }

            Console.ReadLine();
        }
    }
}
```

code download ConsoleApplication1.sln

In this case, the `TransactionScope` object is used and if one of the operations on the database fails, everything will be rolled back to the original state.

Other Methods and Properties of the DataContext Object

In addition to the items just described, a number of other methods and properties are available from the `DataContext` object. The following table shows some of the available methods from `DataContext`.

METHOD	DESCRIPTION
<code>CreateDatabase</code>	Allows you to create a database on the server.
<code>DatabaseExists</code>	Allows you to determine whether a database exists and can be opened.
<code>DeleteDatabase</code>	Deletes the associated database.
<code>ExecuteCommand</code>	Allows you to pass in a command to the database to be executed.
<code>ExecuteQuery</code>	Allows you to pass queries directly to the database.
<code>GetChangeSet</code>	The <code>DataContext</code> object keeps track of changes occurring in the database on your behalf and this method allows you access to these changes.
<code>GetCommand</code>	Gives you access to the commands that are performed.
<code>GetTable</code>	Provides access to a collection of tables from the database.
<code>Refresh</code>	Allows you to refresh your objects from the data that is stored within the database.
<code>SubmitChanges</code>	Executes your CRUD commands in the database that have been established in your code.
<code>Translate</code>	Converts an <code>IDataReader</code> to objects.

In addition to these methods, the `DataContext` object exposes some of the properties shown in the following table.

PROPERTY	DESCRIPTION
<code>ChangeConflicts</code>	Provides a collection of objects that cause concurrency conflicts when the <code>SubmitChanges()</code> method is called.
<code>CommandTimeout</code>	Allows you to set the timeout period in which a command against the database is allowed to run. You should set this to a higher value if your query needs more time to execute.
<code>Connection</code>	Allows you to work with the <code>System.Data.SqlClient.SqlConnection</code> object used by the client.
<code>DeferredLoadingEnabled</code>	Allows you to specify whether or not to delay the loading of one-to-many or one-to-one relationships.
<code>LoadOptions</code>	Allows you to specify or retrieve the value of the <code>DataLoadOptions</code> object.
<code>Log</code>	Allows you to specify the location of the output of the command that was used in the query.
<code>Mapping</code>	Provides the <code>MetaModel</code> on which the mapping is based.
<code>ObjectTrackingEnabled</code>	Specifies whether or not to track changes to the objects within the database for transactional purposes. If you are dealing with a read-only database, you should set this property to <code>false</code> .
<code>Transaction</code>	Allows you to specify the local transaction used with the database.

The Table<TEntity> Object

The Table<TEntity> object is a representation of the tables that you are working with from the database. For instance, you saw the use of the Product class, which is a Table<Product> instance. As you see throughout this chapter, a number of methods are available from the Table<TEntity> object. Some of these methods are defined in the following table.

METHOD	DESCRIPTION
Attach	Allows you to attach an entity to the DataContext instance.
AttachAll	Allows you to attach a collection of entities to the DataContext instance.
DeleteAllOnSubmit<TSubEntity>	Allows you to put all the pending actions into a state of being ready for deletion. Everything here is enacted when the SubmitChanges() method is called from the DataContext object.
DeleteOnSubmit	Allows you to put a pending action into a state of being ready for deletion. Everything here is enacted when the SubmitChanges() method is called from the DataContext object.
GetModifiedMembers	Provides an array of modified objects. You will be able to access their current and changed values.
GetNewBindingList	Provides a new list for binding to the data store.
GetOriginalEntityState	Provides you an instance of the object as it appeared in its original state.
InsertAllOnSubmit<TSubEntity>	Allows you to put all the pending actions into a state of being ready for insertion. Everything here is enacted with the SubmitChanges() method called off of the DataContext object.
InsertOnSubmit	Allows you to put a pending action into a state of being ready for insertion. Everything here is enacted when the SubmitChanges() method is called from the DataContext object.

WORKING WITHOUT THE O/R DESIGNER

Although the new O/R Designer in Visual Studio 2010 makes the creation of everything you need for LINQ to SQL quite easy, it is important to note that the underlying framework upon which this all rests allows you to do everything from the ground up yourself. This provides the most control over what is actually happening.

Creating Your Own Custom Object

To accomplish the same task as was accomplished earlier with the Customer table, you need to expose the Customer table yourself via a class. The first step is to create a new class in your project called Customer.cs. The code for this class is presented here:



Available for
download on
Wrox.com

```
using System.Data.Linq.Mapping;

namespace ConsoleApplication1
{
    [Table(Name = "Customers")]
    public class Customer
    {
        [Column(IsPrimaryKey = true)]
        public string CustomerID { get; set; }
        [Column]
        public string CompanyName { get; set; }
    }
}
```

```

        [Column]
        public string ContactName { get; set; }
        [Column]
        public string ContactTitle { get; set; }
        [Column]
        public string Address { get; set; }
        [Column]
        public string City { get; set; }
        [Column]
        public string Region { get; set; }
        [Column]
        public string PostalCode { get; set; }
        [Column]
        public string Country { get; set; }
        [Column]
        public string Phone { get; set; }
        [Column]
        public string Fax { get; set; }
    }
}

```

code snippet Customer.cs

Here, the `Customer.cs` file defines the `Customer` object that you want to use with LINQ to SQL. The class has the `Table` attribute assigned to it to signify the table class. The `Table` class attribute includes a property called `Name`, which defines the name of the table to use within the database that is referenced with the connection string. Using the `Table` attribute also means that you need to make a reference to the `System.Data.Linq.Mapping` namespace in your code.

In addition to the `Table` attribute, each of the defined properties in the class makes use of the `Column` attribute. As stated earlier, columns from the SQL Server database will map to properties in your code.

Querying with Your Custom Object and LINQ

With only the `Customer` class in place, you are then able to query the Northwind database for the `Customers` table. The code to accomplish this task is illustrated in the following example found in the same console application:

```

using System;
using System.Data.Linq;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main()
        {
            DataContext dc = new DataContext(@"Data Source=.\SQLEXPRESS;
            AttachDbFilename=|DataDirectory|\NORTHWND.MDF;
            Integrated Security=True;User Instance=True");

            dc.Log = Console.Out; // Used for outputting the SQL used

            Table<Customer> myCustomers = dc.GetTable<Customer>();

            foreach (Customer item in myCustomers)
            {
                Console.WriteLine("{0} | {1}",
                    item.CompanyName, item.Country);
            }
        }
    }
}

```

```

        Console.ReadLine();
    }
}

```

In this case, the default `DataContext` object is used and the connection string to the Northwind SQL Server Express database is passed in as a parameter. A `Table` class of type `Customer` is then populated using the `GetTable<TEntity>()` method. For this example, the `GetTable<TEntity>()` operation uses your custom-defined `Customer` class:

```
dc.GetTable<Customer>();
```

What happens is that LINQ to SQL uses the `DataContext` object to make the query to the SQL Server database on your behalf and gets the returned rows as strongly typed `Customer` objects. This allows you to then iterate through each of the `Customer` objects in the `Table` object's collection and get at the information that you need, as is done with the `Console.WriteLine()` statements here:

```

foreach (Customer item in myCustomers)
{
    Console.WriteLine("{0} | {1}",
        item.CompanyName, item.Country);
}

```

Running this code produces the following results in your console application:

```

SELECT [t0].[CustomerID], [t0].[CompanyName], [t0].[ContactName],
[t0].[ContactTitle], [t0].[Address], [t0].[City], [t0].[Region],
[t0].[PostalCode], [t0].[Country], [t0].[Phone], [t0].[Fax]
FROM [Customers] AS [t0]
--Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build: 4.0.21006.1

```

```

Alfreds Futterkiste | Germany
Ana Trujillo Emparedados y helados | Mexico
Antonio Moreno Taquería | Mexico
Around the Horn | UK
Berglunds snabbköp | Sweden

```

```
// Output removed for clarity
```

```

Wartian Herkku | Finland
Wellington Importadora | Brazil
White Clover Markets | USA
Wilman Kala | Finland
Wolski Zajazd | Poland

```

Limiting the Columns Called with the Query

You will notice that the query retrieved every single column that was specified in your `Customer` class file. If you remove the columns that you are not going to need, you can then have a new `Customer` class file as shown here:



Available for
download on
Wrox.com

```

using System.Data.Linq.Mapping;

namespace ConsoleApplication1
{
    [Table(Name = "Customers")]
    public class Customer
    {
        [Column(IsPrimaryKey = true)]
        public string CustomerID { get; set; }
        [Column]
        public string CompanyName { get; set; }
        [Column]
    }
}

```

```

        public string Country { get; set; }
    }
}

```

code snippet Customer.cs

In this case, I removed all the columns that are not utilized by the application. Now if you run the console application and look at the SQL query that is produced, you will see the following results:

```

SELECT [t0].[CustomerID], [t0].[CompanyName], [t0].[Country]
FROM [Customers] AS [t0]

```

You can see that only the three columns that are defined within the `Customer` class are utilized in the query to the `Customers` table.

The property `CustomerID` is interesting in that you are able to signify that this column is a primary key for the table through the use of the `IsPrimaryKey` setting in the `Column` attribute. This setting takes a `Boolean` value and in this case, it is set to `true`.

Working with Column Names

The other important point of the columns is that the name of the property you define in the `Customer` class needs to be the same name as is used in the database. For instance, if you change the name of the `CustomerID` property to `MyCustomerID`, you get the following exception when you try to run your Console application:

```

System.Data.SqlClient.SqlException was unhandled
  Message="Invalid column name 'MyCustomerID'."
  Source=".Net SqlClient Data Provider"
  ErrorCode=-2146232060
  Class=16
  LineNumber=1
  Number=207
  Procedure=""
  Server="\\\\.\\pipe\\F5E22E37-1AF9-44\\tsql\\query"

```

To get around this, you need to define the name of the column in the custom `Customer` class that you have created. You can do this by using the `Column` attribute as illustrated here:

```

[Column(IsPrimaryKey = true, Name = "CustomerID")]
public string MyCustomerID { get; set; }

```

Like the `Table` attribute, the `Column` attribute includes a `Name` property that allows you to specify the name of the column as it appears in the `Customers` table.

Doing this will generate a query as shown here:

```

SELECT [t0].[CustomerID] AS [MyCustomerID], [t0].[CompanyName], [t0].[Country]
FROM [Customers] AS [t0]

```

This also means that you need to now reference the column using the new name of `MyCustomerID` (for example, `item.MyCustomerID`).

Creating Your Own DataContext Object

Now it is probably not the best approach to use the plain-vanilla `DataContext` object, but instead, you will find that you have more control by creating your own `DataContext` class. To accomplish this task, create a new class called `MyNorthwindDataContext.cs` and have the class inherit from `DataContext`. Your class in its simplest form is illustrated here:



Available for
download on
Wrox.com

```

using System.Data.Linq;

namespace ConsoleApplication1
{

```

```

public class MyNorthwindDataContext: DataContext
{
    public Table<Customer> Customers;

    public MyNorthwindDataContext()
    : base(@"Data Source=. \SQLEXPRESS;
          AttachDbFilename=|DataDirectory|\NORTHWND.MDF;
          Integrated Security=True;User Instance=True")
    {
    }
}

```

code snippet MyNorthwindDataContext.cs

Here, the class `MyNorthwindDataContext` inherits from `DataContext` and provides an instance of the `Table<Customer>` object from the `Customer` class that you created earlier. The constructor is the other requirement of this class. This constructor uses a base to initialize a new instance of the object referencing a file (in this case a connection to a SQL database file).

Using your own `DataContext` object now allows you to change the code in your application to the following:



```

using System;
using System.Data.Linq;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main()
        {
            MyNorthwindDataContext dc = new MyNorthwindDataContext();
            Table<Customer> myCustomers = dc.Customers;

            foreach (Customer item in myCustomers)
            {
                Console.WriteLine("{0} | {1}",
                                   item.CompanyName, item.Country);
            }

            Console.ReadLine();
        }
    }
}

```

code download ConsoleApplication1.sln

By creating an instance of the `MyNorthwindDataContext` object, you are now allowing the class to manage the connection to the database. You will also notice that now you have direct access to the `Customer` class through the `dc.Customers` statement.



Note that the examples provided in this chapter are considered bare-bones examples in that they don't include all the error-handling and logging that would generally go into building your applications. This is done to illustrate the points being discussed in the chapter and nothing more.

CUSTOM OBJECTS AND THE O/R DESIGNER

In addition to building your custom object in your own .cs file and then tying that class to the DataContext that you have built, you can also use the O/R Designer in Visual Studio 2010 to build your class files. When you use Visual Studio in this manner, it will create the appropriate .cs file on your behalf, but by using the O/R Designer, you will also have a visual representation of the class file and any possible relationships that are established.

When viewing the Designer view of your .dbml file, you will notice that there are three items present in the toolbox. These items are Class, Association, and Inheritance.

For an example of this, take the Class object from the toolbox and drop it onto the design surface. You will be presented with an image of the generic class as shown in Figure 56-7.



FIGURE 56-7

From here, you can now click the Class1 name and rename this class to Customer. Right-clicking next to the name enables you to add properties to the class file by selecting Add Property from the provided menu. For this example, give the Customer class three properties — CustomerID, CompanyName, and Country. If you highlight the CustomerID property, you will be able to configure the property from the Properties dialog in Visual Studio and change the Primary Key setting from False to True. You also want to highlight the entire class and go to the Properties dialog and change the Source property to Customers because this is the name of the table from which this Customer object needs to work. After this is all done, you will have a visual representation of the class as shown in Figure 56-8.

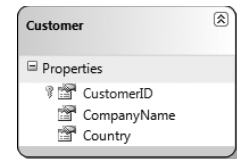


FIGURE 56-8

As you can see from this image, the CustomerID property is properly represented with a primary key icon next to the name. With this in place, you can expand the plus sign next to the Northwind.dbml file and you will find three files here — Northwind.dbml.layout, Northwind.designer.cs, and Northwind.dbml.diagram. The Northwind.dbml.layout file is an XML file that helps Visual Studio with the visual representation shown in the O/R Designer. The most important file is Northwind.designer.cs. This is the Customer class file that was created on your behalf. When you open this file, you are able to see what Visual Studio created for you.

First, you will find the Customer class file within the code of the page:

```
[Table(Name="Customers")]
public partial class Customer: INotifyPropertyChanging,
    INotifyPropertyChanged
{
    // Code removed for clarity
}
```

The Customer class is the name of the class according to what you provided in the designer. The class comes with the Table attribute and provides a name value of Customers because this is the name of the database that this object needs to work with when connecting to the Northwind database.

Within the Customer class, you will find the three properties that you defined. Presented here is just one of the properties — CustomerID:



Available for
download on
Wrox.com

```
[Column(Storage="_CustomerID", CanBeNull=false, IsPrimaryKey=true)]
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();
        }
    }
}

```

code snippet Customer.cs

Similar to when you built a class for yourself from the earlier example, the properties defined use the `Column` attribute and some of the properties available to this attribute. You can see that the primary key setting is set using the `IsPrimaryKey` item.

In addition to the `Customer` class, you will find that a class inheriting from the `DataContext` object is also within the created file:

```

[System.Data.Linq.Mapping.DatabaseAttribute(Name="NORTHWND")]
public partial class NorthwindDataContext: System.Data.Linq.DataContext
{
    // Code removed for clarity
}

```

The `DataContext` object, `NorthwindDataContext`, allows you to connect to the Northwind database and the `Customers` table, as was accomplished in the previous examples.

You will find that using the O/R Designer is a process that can make the creation of your database object class files simple and straightforward. However, at the same time, if you want complete control, you can code up everything yourself and get the results you are after.

QUERYING THE DATABASE

As you've seen, there are a number of ways in which you can query the database from the code of your application. In some of the simplest forms, your queries looked like the following:

```
Table<Product> query = dc.Products;
```

This command was pulling down the entire `Products` table to your query object instance.

Using Query Expressions

In addition to pulling a table straight out of the database using `dc.Products`, you also can use a query expression directly in your code that is strongly typed. An example of this is shown in the following code:



```

using System;
using System.Linq;

namespace ConsoleApplication1
{
    class Class1
    {
        static void Main(string[] args)
        {
            NorthwindDataContext dc = new NorthwindDataContext();

            var query = from p in dc.Products
                        select p;
        }
    }
}

```

```
foreach (Product item in query)
{
    Console.WriteLine(item.ProductID + " | " + item.ProductName);
}

Console.ReadLine();
}
```

[code download ConsoleApplication1.sln](#)

In this case, a query object (again, a `Table<Product>` object) is populated with the query value of `from p in dc.Products select p;`. This command, though shown on two lines for readability purposes, can also be presented on a single line if you wish.

Query Expressions in Detail

You will find that there are a number of query expressions that you can use from your code. The previous example is a simple select statement that returns the entire table. The following list of items are some of the other query expressions that you have at your disposal.

SEGMENTATION	SYNTAX
Project	<code>select <expression></code>
Filter	<code>where <expression>, distinct</code>
Test	<code>any(<expression>), all(<expression>)</code>
Join	<code><expression> join <expression> on <expression> equals <expression></code>
Group	<code>group by <expression>, into <expression>, <expression> group join <decision> on <expression> equals <expression> into <expression></code>
Aggregate	<code>count([<expression>]), sum(<expression>), min(<expression>), max(<expression>), avg(<expression>)</code>
Partition	<code>skip [while] <expression>, take [while] <expression></code>
Set	<code>union, intersect, except</code>
Order	<code>order by <expression>, <expression> [ascending descending]</code>

Filtering Using Expressions

In addition to straight queries for the entire table, you can filter items using the `where` and `distinct` options. The following provides an example of querying the `Products` table for a specific type of record:

```
var query = from p in dc.Products
            where p.ProductName.StartsWith("L")
            select p;
```

In this case, this query is selecting all the records from the `Products` table that start with the letter `L`. This is done via the `where p.ProductName.StartsWith("L")` expression. You will find a large selection of methods available from the `ProductName` property that allows you to fine-tune the filtering you need. This operation produces the following results:

```
65 | Louisiana Fiery Hot Pepper Sauce
66 | Louisiana Hot Spiced Okra
67 | Laughing Lumberjack Lager
74 | Longlife Tofu
76 | Lakkalikööri
```

You can also add as many of these expressions to the list as you need. For instance, here is an example of adding two where statements to your query:

```
var query = from p in dc.Products
            where p.ProductName.StartsWith("L")
            where p.ProductName.EndsWith("i")
            select p;
```

In this case, there is a filter expression that looks for items with a product name starting with the letter L and then a second expression is done to make sure the second criteria is also applied, which states that the items must also end with the letter i. This would give you the following results:

76 | Lakkalikööri

Performing Joins

In addition to working with one table, you can work with multiple tables and perform joins with your queries. If you drag and drop both the Customers table and the Orders table onto the Northwind.dbml design surface, you will get the result presented in Figure 56-9.

From this figure, you can see that after you drag and drop both of these elements onto the design surface, Visual Studio will know that there is a relationship between these items and will create this relationship for you in the code and represent it with the black arrow.

From here, you can use a join statement in your query to work with both of the tables as presented in the following example:

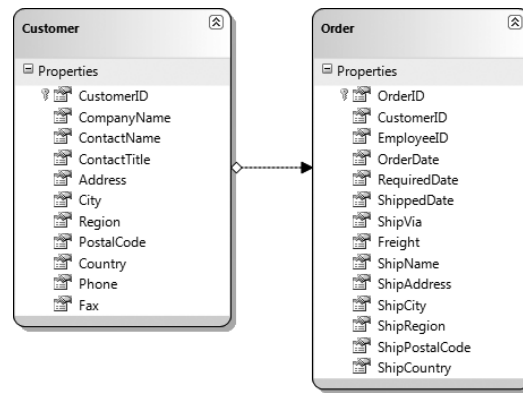


FIGURE 56-9



Available for
download on
Wrox.com

```
using System;
using System.Linq;

namespace ConsoleApplication1
{
    class Class1
    {
        static void Main(string[] args)
        {
            NorthwindDataContext dc = new NorthwindDataContext();
            dc.Log = Console.Out;

            var query = from c in dc.Customers
                        join o in dc.Orders on c.CustomerID equals o.CustomerID
                        orderby c.CustomerID
                        select new { c.CustomerID, c.CompanyName,
                                    c.Country, o.OrderID, o.OrderDate };

            foreach (var item in query)
            {
                Console.WriteLine(item.CustomerID + " | " + item.CompanyName
                                   + " | " + item.Country + " | " + item.OrderID
                                   + " | " + item.OrderDate);
            }
        }
    }
}
```

```

    }
    Console.ReadLine();
}
}
}

```

code download ConsoleApplication1.sln

This example is pulling from the `Customers` table and joining on the `Orders` table where the `CustomerID` columns match. This is done through the `join` statement:

```
join o in dc.Orders on c.CustomerID equals o.CustomerID
```

From here, a new object is created with the `select new` statement and this new object comprises of the `CustomerID`, `CompanyName`, and `Country` columns from the `Customer` table as well as the `OrderID` and `OrderDate` columns from the `Orders` table.

When it comes to iterating through the collection of this new object, the interesting part is that the `foreach` statement also uses the `var` keyword because the type is not known at this point in time:

```

foreach (var item in query)
{
    Console.WriteLine(item.CustomerID + " | " + item.CompanyName
        + " | " + item.Country + " | " + item.OrderID
        + " | " + item.OrderDate);
}

```

Regardless, the `item` object here has access to all the properties that you specified. When you run this example, you will get results similar to what is presented in this partial result:

WILMK	Wilman Kala	Finland	10695	10/7/1997 12:00:00 AM
WILMK	Wilman Kala	Finland	10615	7/30/1997 12:00:00 AM
WILMK	Wilman Kala	Finland	10673	9/18/1997 12:00:00 AM
WILMK	Wilman Kala	Finland	11005	4/7/1998 12:00:00 AM
WILMK	Wilman Kala	Finland	10879	2/10/1998 12:00:00 AM
WILMK	Wilman Kala	Finland	10873	2/6/1998 12:00:00 AM
WILMK	Wilman Kala	Finland	10910	2/26/1998 12:00:00 AM

Grouping Items

You are also easily able to group items with your queries. In the `Northwind.dbml` example that you are working with, drag and drop the `Categories` table onto the design surface and you will see that there is a relation with this table and the `Products` table from earlier. The following example shows you how to group products by categories:



Available for
download on
Vrox.com

```

using System;
using System.Linq;

namespace ConsoleApplication1
{
    class Class1
    {
        static void Main(string[] args)
        {
            NorthwindDataContext dc = new NorthwindDataContext();

            var query = from p in dc.Products
                        orderby p.Category.CategoryName ascending
                        group p by p.Category.CategoryName into g
                        select new { Category = g.Key, Products = g };

            foreach (var item in query)
            {

```

```

        Console.WriteLine(item.Category);

        foreach (var innerItem in item.Products)
        {
            Console.WriteLine("      " + innerItem.ProductName);
        }

        Console.WriteLine();
    }

    Console.ReadLine();
}
}
}

```

[*code download ConsoleApplication1.sln*](#)

This example creates a new object, which is a group of categories, and packages the entire `Product` table into this new table called `g`. Before that, the categories are ordered by name using the `orderby` statement because the order provided is in ascending order (the other option being descending). The output is the `Category` (passed in through the `Key` property) and the `Product` instance. The iteration with the `foreach` statements is done once for the categories and once for each of the products that are found in the category.

A partial output of this program is presented here:

```

Beverages
    Chai
    Chang
    Guaraná Fantástica
    Sasquatch Ale
    Steeleye Stout
    Côte de Blaye
    Chartreuse verte
    Ipoh Coffee
    Laughing Lumberjack Lager
    Outback Lager
    Rhönbräu Klosterbier
    Lakkalikööri

Condiments
    Aniseed Syrup
    Chef Anton's Cajun Seasoning
    Chef Anton's Gumbo Mix
    Grandma's Boysenberry Spread
    Northwoods Cranberry Sauce
    Genen Shouyu
    Gula Malacca
    Sirop d'érable
    Vegie-spread
    Louisiana Fiery Hot Pepper Sauce
    Louisiana Hot Spiced Okra
    Original Frankfurter grüne Soße

```

You will find that there a lot more commands and expressions available to you beyond what are presented in this short chapter.

STORED PROCEDURES

So far, you have been querying the tables directly and leaving it up to LINQ to create the appropriate SQL statement for the operation. When working with pre-existing databases that make heavy use of stored procedures and for those that want to follow the best practice of using stored procedures within a database, you will find that LINQ is still a viable option.

LINQ to SQL treats working with stored procedures as a method call. As you saw in Figure 56-4, there is a design surface called the O/R Designer that allows you to drag and drop tables onto it so that you can then programmatically work with the table. On the right side of the O/R Designer, you will find a spot where you are able to drag and drop stored procedures.

Any stored procedures that you drag and drop onto this part of the O/R Designer will now become available methods to you from the `DataContext` object. For this example, drag and drop the `TenMostExpensiveProducts` stored procedure onto this part of the O/R Designer.

The following example shows how you would call this stored procedure within the Northwind database:



```
using System;
using System.Collections.Generic;
using System.Data.Linq;
using System.Linq;

namespace ConsoleApplication1
{
    class Class1
    {
        static void Main(string[] args)
        {
            NorthwindDataContext dc = new NorthwindDataContext();

            ISingleResult<Ten_Most_Expensive_ProductsResult> result =
                dc.Ten_Most_Expensive_Products();

            foreach (Ten_Most_Expensive_ProductsResult item in result)
            {
                Console.WriteLine(item.TenMostExpensiveProducts + " | " +
                    item.UnitPrice);
            }

            Console.ReadLine();
        }
    }
}
```

[code download ConsoleApplication1.sln](#)

From this example, you can see that the rows coming out of the stored procedure are collected into an `ISingleResult<Ten_Most_Expensive_ProductsResult>` object. From here, iteration through this object is as simple as all the rest.

As you can see from this example, calling your stored procedures is a simple process.

SUMMARY

One of the more exciting features of the .NET Framework 4 release is the LINQ capabilities that the platform provides. This chapter focused on using LINQ to SQL and some of the options available to you in querying your SQL Server databases.

Using LINQ to SQL enables you to have a strongly typed set of operations for performing CRUD operations against your database. With that said, though, you are still able to use pre-existing access capabilities whether that is interacting with ADO.NET or working with your stored procedures.

