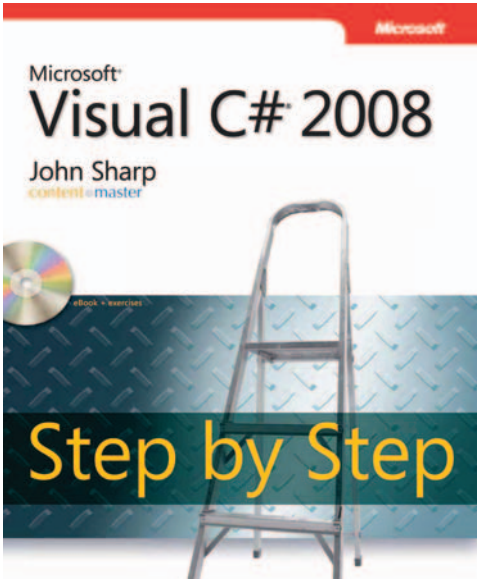


Microsoft® Visual C#® 2008 Step by Step

John Sharp (Content Master)

To learn more about this book, visit Microsoft Learning at
http://www.microsoft.com/MSPress/books/11298.aspx

9780735624306

Microsoft Press

Table of Contents

Chapter 25

Querying Information in a Database

After completing this chapter, you will be able to:

- Fetch and display data from a Microsoft SQL Server database by using Microsoft ADO.NET.
- Define entity classes for holding data retrieved from a database.
- Use DLINQ to query a database and populate instances of entity classes.
- Create a custom *DataContext* class for accessing a database in a typesafe manner.

In Part IV of this book, "Working with Windows Applications," you learned how to use Microsoft Visual C# to build user interfaces and present and validate information. In Part V, you will learn about managing data by using the data access functionality available in Microsoft Visual Studio 2008 and the Microsoft .NET Framework. The chapters in this part of the book describe ADO.NET, a library of objects specifically designed to make it easy to write applications that use databases. In this chapter, you will also learn how to query data by using DLINQ—extensions to LINQ based on ADO.NET that are designed for retrieving data from a database. In Chapter 26, "Displaying and Editing Data by Using Data Binding," you will learn more about using ADO.NET and DLINQ for updating data.

Important To perform the exercises in this chapter, you must have installed Microsoft SQL Server 2005 Express Edition, Service Pack 2. This software is available on the retail DVD with Microsoft Visual Studio 2008 and Visual C# 2008 Express Edition and is installed by default.

Important It is recommended that you use an account that has Administrator privileges to perform the exercises in this chapter and the remainder of this book.

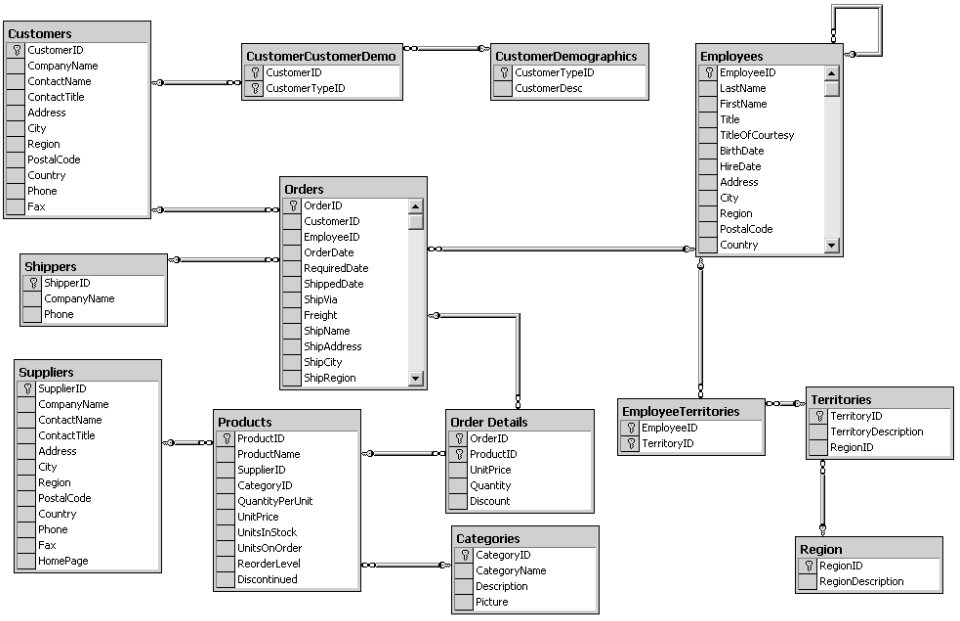
Querying a Database by Using ADO.NET

The ADO.NET class library contains a comprehensive framework for building applications that need to retrieve and update data held in a relational database. The model defined by ADO.NET is based on the notion of data providers. Each database management system (such as SQL Server, Oracle, IBM DB2, and so on) has its own data provider that implements an abstraction of the mechanisms for connecting to a database, issuing queries, and updating data. By using these abstractions, you can write portable code that is independent of the

underlying database management system. In this chapter, you will connect to a database managed by SQL Server 2005 Express Edition, but the techniques that you will learn are equally applicable when using a different database management system.

The Northwind Database

Northwind Traders is a fictitious company that sells edible goods with exotic names. The Northwind database contains several tables with information about the goods that Northwind Traders sells, the customers they sell to, orders placed by customers, suppliers from whom Northwind Traders obtains goods to resell, shippers that they use to send goods to customers, and employees who work for Northwind Traders. Figure 25-1 shows all the tables in the Northwind database and how they are related to one another. The tables that you will be using in this chapter are *Orders* and *Products*.



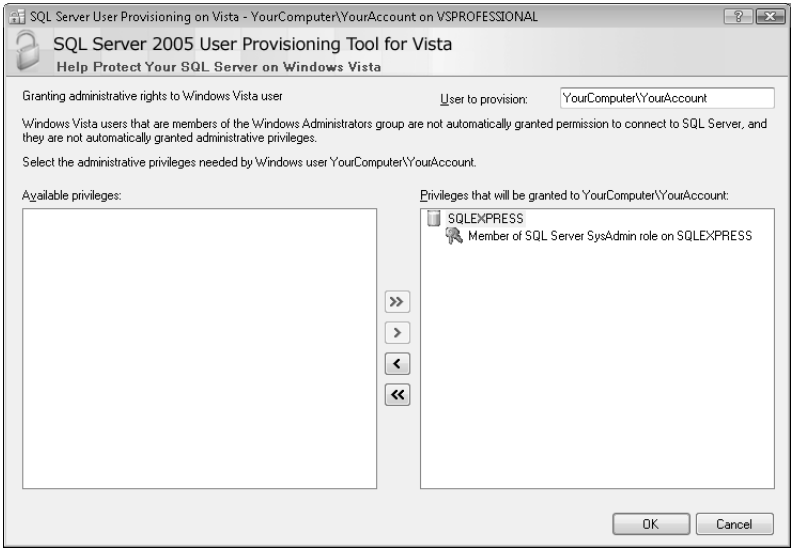
Creating the Database

Before proceeding further, you need to create the Northwind database.

Granting Permissions for Creating a SQL Server 2005 Database

You must have administrative rights for SQL Server 2005 Express before you can create a database. By default, if you are using the Windows Vista operating system, the computer *Administrator* account and members of the *Administrators* group do not have these rights. You can easily grant these permissions by using the SQL Server 2005 User Provisioning Tool for Vista, as follows:

1. Log on to your computer as an account that has administrator access.
2. Run the sqlprov.exe program, located in the folder C:\Program Files\Microsoft SQL Server\90\Shared.
3. In the *User Account Control* dialog box, click *Continue*. A console window briefly appears, and then the *SQL Server User Provisioning on Vista* window is displayed.
4. In the *User to provision* text box, type the name of the account you are using to perform the exercises. (Replace *YourComputer\YourAccount* with the name of your computer and your account.)
5. In the *Available privileges* box, click *Member of SQL Server SysAdmin role on SQLEXPRESS*, and then click the **>>** button.



6. Click *OK*.

   The permission will be granted to the specified user, and the SQL Server 2005 User Provisioning Tool for Vista will close automatically.

Create the Northwind database

1. On the Windows *Start* menu, click *All Programs*, click *Accessories*, and then click *Command Prompt* to open a command prompt window. If you are using Windows Vista, in the command prompt window type the following command to go to the \Microsoft Press\Visual CSharp Step by Step\Chapter 25 folder under your Documents folder. Replace *Name* with your user name.

   ```
   cd "\Users\Name\Documents\Microsoft Press\Visual CSharp Step by Step\Chapter 25"
   ```

   If you are using Windows XP, type the following command to go to the \Microsoft Press\Visual CSharp Step by Step\Chapter 25 folder under your My Documents folder, replacing *Name* with your user name.

   ```
   cd "\Documents and Settings\Name\My Documents\Microsoft Press\Visual CSharp Step by
   Step\Chapter 25"
   ```

2. In the command prompt window, type the following command:

   ```
   sqlcmd -S YourComputer\SQLExpress -E -iinstnwnd.sql
   ```

   Replace *YourComputer* with the name of your computer.

   This command uses the sqlcmd utility to connect to your local instance of SQL Server 2005 Express and run the instnwnd.sql script. This script contains the SQL commands that create the Northwind Traders database and the tables in the database and fills them with some sample data.

Tip Ensure that SQL Server 2005 Express is running before you attempt to create the Northwind database. (It is set to start automatically by default. You will simply receive an error message if it is not started when you execute the *sqlcmd* command.) You can check the status of SQL Server 2005 Express, and start it running if necessary, by using the SQL Configuration Manager available in the Configuration Tools folder of the Microsoft SQL Server 2005 program group.

3. When the script finishes running, close the command prompt window.

Note You can run the command you executed in step 2 at any time if you need to reset the Northwind Traders database. The instnwnd.sql script automatically drops the database if it exists and then rebuilds it. See Chapter 26 for additional information.

Using ADO.NET to Query Order Information

In the next set of exercises, you will write code to access the Northwind database and display information in a simple console application. The aim of the exercise is to help you learn more about ADO.NET and understand the object model it implements. In later exercises, you will use DLINQ to query the database. In Chapter 26, you will see how to use the wizards included with Visual Studio 2008 to generate code that can retrieve and update data and display data graphically in a Windows Presentation Foundation (WPF) application.

The application you are going to create first will produce a simple report displaying information about customers' orders. The program will prompt the user for a customer ID and then display the orders for that customer.

Connect to the database

1. Start Visual Studio 2008 if it is not already running.
2. Create a new project called ReportOrders by using the Console Application template. Save it in the \Microsoft Press\Visual CSharp Step By Step\Chapter 25 folder under your Documents folder, and then click *OK*.

Note Remember, if you are using Visual C# 2008 Express Edition, you can specify the location for saving your project by setting the *Visual Studio projects location* in the *Projects and Solutions* section of the *Options* dialog box on the *Tools* menu.

3. In *Solution Explorer*, change the name of the file Program.cs to Report.cs. In the *Microsoft Visual Studio* message, click *Yes* to change all references of the *Program* class to *Report*.
4. In the *Code and Text Editor* window, add the following *using* statement to the list at the top of the file:

```
using System.Data.SqlClient;
```

   The *System.Data.SqlClient* namespace contains the SQL Server data provider classes for ADO.NET. These classes are specialized versions of the ADO.NET classes, optimized for working with SQL Server.

5. In the *Main* method of the *Report* class, add the following statement shown in bold type, which declares a *SqlConnection* object:

```
static void Main(string[] args)
{
    SqlConnection dataConnection = new SqlConnection();
}
```

SqlConnection is a subclass of an ADO.NET class called *Connection*. It is designed to handle connections to SQL Server databases.

6. After the variable declaration, add a *try/catch* block to the *Main* method. All the code that you will write for gaining access to the database goes inside the *try* part of this block. In the *catch* block, add a simple handler that catches *SqlException* exceptions. The new code is shown in bold type here:

```
static void Main(string[] args)
{
    ...
    try
    {
        // You will add your code here in a moment
    }
    catch(SqlException e)
    {
        Console.WriteLine("Error accessing the database: {0}", e.Message);
    }
}
```

A *SqlException* is thrown if an error occurs when accessing a SQL Server database.

7. Replace the comment in the *try* block with the code shown in bold type here:

```
try
{
    dataConnection.ConnectionString =
        "Integrated Security=true;Initial Catalog=Northwind;" +
        "Data Source=YourComputer\\SQLExpress";
    dataConnection.Open();
}
```

Important In the *ConnectionString* property, replace *YourComputer* with the name of your computer. Make sure that you type the string on a single line.

This code attempts to create a connection to the Northwind database. The contents of the *ConnectionString* property of the *SqlConnection* object contain elements that specify that the connection will use Windows Authentication to connect to the Northwind database on your local instance of SQL Server 2005 Express Edition. This is the preferred method of access because you do not have to prompt the user for any form of user name or password, and you are not tempted to hard-code user names and passwords into your application. Notice that a semicolon separates all the elements in the *ConnectionString*.

You can also encode many other elements in the connection string. See the documentation supplied with Visual Studio 2008 for details.

Using SQL Server Authentication

Windows Authentication is useful for authenticating users who are all members of a Windows domain. However, there might be occasions when the user accessing the database does not have a Windows account, for example, if you are building an application designed to be accessed by remote users over the Internet. In these cases, you can use the *User ID* and *Password* parameters instead, like this:

```
string userName = ...;
string password = ...;
// Prompt the user for their name and password, and fill these variables

string connString = String.Format(
    "User ID={0};Password={1};Initial Catalog=Northwind;"  +
    "Data Source=YourComputer\\SQLExpress", username, password);

myConnection.ConnectionString = connString;
```

At this point, I should offer a sentence of advice: never hard-code user names and passwords into your applications. Anyone who obtains a copy of the source code (or who reverse-engineers the compiled code) can see this information, and this renders the whole point of security meaningless.

The next step is to prompt the user for a customer ID and then query the database to find all of the orders for that customer.

Query the *Orders* table

1. Add the statements shown here in bold type to the *try* block after the *dataConnection.Open();* statement:

```
try
{
    ...
    Console.Write("Please enter a customer ID (5 characters): ");
    string customerId = Console.ReadLine();
}
```

   These statements prompt the user for a customer ID and read the user's response in the string variable *customerId*.

2. Type the following statements shown in bold type after the code you just entered:

```
try
{
    ...
    SqlCommand dataCommand = new SqlCommand();
    dataCommand.Connection = dataConnection;
```

```
    dataCommand.CommandText =
        "SELECT OrderID, OrderDate, ShippedDate, ShipName, ShipAddress, " +
        "ShipCity, ShipCountry " +
        "FROM Orders WHERE CustomerID='" + customerId + "'";
    Console.WriteLine("About to execute: {0}\n\n", dataCommand.CommandText);
}
```

The first statement creates a *SqlCommand* object. Like *SqlConnection*, this is a specialized version of an ADO.NET class, *Command*, that has been designed for performing queries against a SQL Server database. An ADO.NET *Command* object is used to execute a command against a data source. In the case of a relational database, the text of the command is a SQL statement.

The second line of code sets the *Connection* property of the *SqlCommand* object to the database connection you opened in the preceding exercise. The next two statements populate the *CommandText* property with a SQL SELECT statement that retrieves information from the *Orders* table for all orders that have a *CustomerID* that matches the value in the *customerId* variable. The *Console.WriteLine* statement just repeats the command about to be executed to the screen.

Important If you are an experienced database developer, you will probably be about to e-mail me telling me that using string concatenation to build SQL queries is bad practice. This approach renders your application vulnerable to SQL injection attacks. However, the purpose of this code is to quickly show you how to execute queries against a SQL Server database by using ADO.NET, so I have deliberately kept it simple. Do not write code such as this in your production applications.

For a description of what a SQL injection attack is, how dangerous it can be, and how you should write code to avoid such attacks, see the SQL Injection topic in SQL Server Books Online, available at *http://msdn2.microsoft.com/en-us/library/ms161953.aspx*.

3. Add the following statement shown in bold type after the code you just entered:

```
try
{
    ...
    SqlDataReader dataReader = dataCommand.ExecuteReader();
}
```

The *ExecuteReader* method of a *SqlCommand* object constructs a *SqlDataReader* object that you can use to fetch the rows identified by the SQL statement. The *SqlDataReader* class provides the fastest mechanism available (as fast as your network allows) for retrieving data from a SQL Server.

The next task is to iterate through all the orders (if there are any) and display them.

Fetch data and display orders

1. Add the *while* loop shown here in bold type after the statement that creates the *SqlDataReader* object:

```
try
{
    ...
    while (dataReader.Read())
    {
        // Code to display the current row
    }
}
```

The *Read* method of the *SqlDataReader* class fetches the next row from the database. It returns *true* if another row was retrieved successfully; otherwise, it returns *false*, usually because there are no more rows. The *while* loop you have just entered keeps reading rows from the *dataReader* variable and finishes when there are no more rows.

2. Add the statements shown here in bold type to the body of the *while* loop you created in the preceding step:

```
while (dataReader.Read())
{
    int orderId = dataReader.GetInt32(0);
    DateTime orderDate = dataReader.GetDateTime(1);
    DateTime shipDate = dataReader.GetDateTime(2);
    string shipName = dataReader.GetString(3);
    string shipAddress = dataReader.GetString(4);
    string shipCity = dataReader.GetString(5);
    string shipCountry = dataReader.GetString(6);
    Console.WriteLine(
        "Order: {0}\nPlaced: {1}\nShipped: {2}\n" +
        "To Address: {3}\n{4}\n{5}\n{6}\n\n", orderId, orderDate,
        shipDate, shipName, shipAddress, shipCity, shipCountry);
}
```

This block of code shows how you read the data from the database by using a *SqlDataReader* object. A *SqlDataReader* object contains the most recent row retrieved from the database. You can use the *GetXXX* methods to extract the information from each column in the row—there is a *GetXXX* method for each common type of data. For example, to read an *int* value, you use the *GetInt32* method; to read a string, you use the *GetString* method; and you can probably guess how to read a *DateTime* value. The *GetXXX* methods take a parameter indicating which column to read: 0 is the first column, 1 is the second column, and so on. The preceding code reads the various columns from the current *Orders* row, stores the values in a set of variables, and then prints out the values of these variables.

Firehose Cursors

One of the major drawbacks in a multiuser database application is locked data. Unfortunately, it is common to see applications retrieve rows from a database and keep those rows locked to prevent another user from changing the data while the application is using them. In some extreme circumstances, an application can even prevent other users from reading data that it has locked. If the application retrieves a large number of rows, it locks a large proportion of the table. If there are many users running the same application at the same time, they can end up waiting for one another to release locks and it all leads to a slow-running and frustrating mess.

The *SqlDataReader* class has been designed to remove this drawback. It fetches rows one at a time and does not retain any locks on a row after it has been retrieved. It is wonderful for improving concurrency in your applications. The *SqlDataReader* class is sometimes referred to as a "firehose cursor." (The term *cursor* is an acronym that stands for "current set of rows.")

When you have finished using a database, it's good practice to close your connection and release any resources you have been using.

Disconnect from the database, and test the application

1. Add the statement shown here in bold type after the *while* loop in the *try* block:

```
try
{
    ...
    while(dataReader.Read())
    {
        ...
    }
    dataReader.Close();
}
```

This statement closes the *SqlDataReader* object. You should always close a *SqlDataReader* object when you have finished with it because you will not able to use the current *SqlConnection* object to run any more commands until you do. It is also considered good practice to do it even if all you are going to do next is close the *SqlConnection*.

Note If you activate multiple active result sets (MARS) with SQL Server 2005, you can open more than one *SqlDataReader* object against the same *SqlConnection* object and process multiple sets of data. MARS is disabled by default. To learn more about MARS and how you can activate and use it, consult SQL Server 2005 Books Online.

2. After the *catch* block, add the following *finally* block:

```
catch(SqlException e)
{
    ...
}
finally
{
    dataConnection.Close();
}
```

Database connections are scarce resources. You need to ensure that they are closed when you have finished with them. Putting this statement in a *finally* block guarantees that the *SqlConnection* will be closed, even if an exception occurs; remember that the code in the *finally* block will be executed after the *catch* handler has finished.

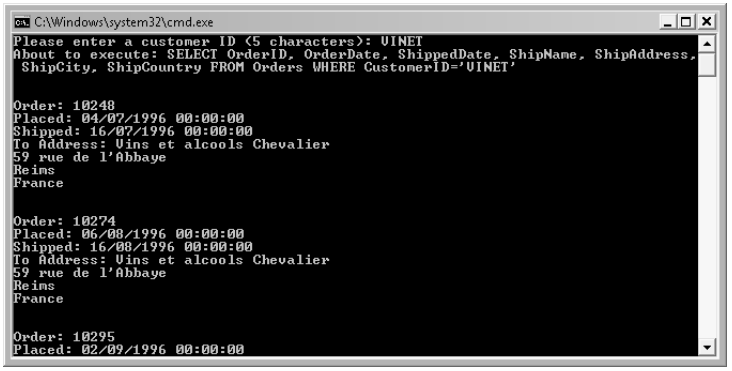
Tip An alternative approach to using a *finally* block is to wrap the code that creates the *SqlDataConnection* object in a *using* statement, as shown in the following code. At the end of the block defined by the *using* statement, the *SqlConnection* object is closed automatically, even if an exception occurs:

```
using (SqlConnection dataConnection = new SqlConnection())
{
    try
    {
        dataConnection.ConnectionString = "...";
        ...
    }
    catch (SqlException e)
    {
        Console.WriteLine("Error accessing the database: {0}", e.Message);
    }
}
```

3. On the *Debug* menu, click *Start Without Debugging* to build and run the application.

4. At the customer ID prompt, type the customer ID **VINET**, and press Enter.

The SQL SELECT statement appears, followed by the orders for this customer, as shown in the following image:



You can scroll back through the console window to view all the data. Press the Enter key to close the console window when you have finished.

5. Run the application again, and then type **BONAP** when prompted for the customer ID.

   Some rows appear, but then an error occurs. If you are using Windows Vista, a message box appears with the message "ReportOrders has stopped working." Click *Close program* (or *Close the program* if you are using Visual C# Express). If you are using Windows XP, a message box appears with the message "ReportOrders has encountered a problem and needs to close. We are sorry for the inconvenience." Click *Don't Send*.

   An error message containing the text "Data is Null. This method or property cannot be called on Null values" appears in the console window.

   The problem is that relational databases allow some columns to contain null values. A null value is a bit like a null variable in C#: It doesn't have a value, but if you try to read it, you get an error. In the *Orders* table, the *ShippedDate* column can contain a null value if the order has not yet been shipped. You should also note that this is a *SqlNullValueException* and consequently is not caught by the *SqlException* handler.

6. Press Enter to close the console window and return to Visual Studio 2008.

Closing Connections

In many older applications, you might notice a tendency to open a connection when the application starts and not close the connection until the application terminates. The rationale behind this strategy was that opening and closing database connections were expensive and time-consuming operations. This strategy had an impact on the scalability of applications because each user running the application had a connection to the database open while the application was running, even if the user went to lunch for a few hours. Most databases limit the number of concurrent connections that they allow. (Sometimes this is because of licensing, but usually it's because each connection consumes resources on the database server that are not infinite.) Eventually, the database would hit a limit on the number of users that could operate concurrently.

Most .NET Framework data providers (including the SQL Server provider) implement *connection pooling*. Database connections are created and held in a pool. When an application requires a connection, the data access provider extracts the next available connection from the pool. When the application closes the connection, it is returned to the pool and made available for the next application that wants a connection. This means that opening and closing database connections are no longer expensive operations. Closing a connection does not disconnect from the database; it just returns the connection to the pool. Opening a connection is simply a matter of obtaining an already-open connection from the pool. Therefore, you should not hold on to connections longer than you need to—open a connection when you need it, and close it as soon as you have finished with it.

You should note that the *ExecuteReader* method of the *SqlCommand* class, which creates a *SqlDataReader*, is overloaded. You can specify a *System.Data.CommandBehavior* parameter that automatically closes the connection used by the *SqlDataReader* when the *SqlDataReader* is closed, like this:

```
SqlDataReader dataReader =
    dataCommand.ExecuteReader(System.Data.CommandBehavior.CloseConnection);
```

When you read the data from the *SqlDataReader* object, you should check that the data you are reading is not null. You'll see how to do this next.

Handle null database values

1. In the *Main* method, change the code in the body of the *while* loop to contain an *if* ... *else* block, as shown here in bold type:

```
while (dataReader.Read())
{
    int orderId = dataReader.GetInt32(0);
    if (dataReader.IsDBNull(2))
    {
        Console.WriteLine("Order {0} not yet shipped\n\n", orderId);
    }
    else
    {
        DateTime orderDate = dataReader.GetDateTime(1);
        DateTime shipDate = dataReader.GetDateTime(2);
        string shipName = dataReader.GetString(3);
        string shipAddress = dataReader.GetString(4);
        string shipCity = dataReader.GetString(5);
        string shipCountry = dataReader.GetString(6);
        Console.WriteLine(
            "Order {0}\nPlaced {1}\nShipped{2}\n" +
            "To Address {3}\n{4}\n{5}\n{6}\n\n", orderId, orderDate,
            shipDate, shipName, shipAddress, shipCity, shipCountry);
    }
}
```

   The *if* statement uses the *IsDBNull* method to determine whether the *ShippedDate* column (column 2 in the table) is null. If it is null, no attempt is made to fetch it (or any of the other columns, which should also be null if there is no *ShippedDate* value); otherwise, the columns are read and printed as before.

2. Build and run the application again.

3. Type **BONAP** for the customer ID when prompted.

   This time you do not get any errors, but you receive a list of orders that have not yet been shipped.

4. When the application finishes, press Enter and return to Visual Studio 2008.

Querying a Database by Using DLINQ

In Chapter 20, "Querying In-Memory Data by Using Query Expressions," you saw how to use LINQ to examine the contents of enumerable collections held in memory. LINQ provides query expressions, which use SQL-like syntax for performing queries and generating a result set that you can then step through. It should come as no surprise that you can use an extended form of LINQ, called DLINQ, for querying and manipulating the contents of a database. DLINQ is built on top of ADO.NET. DLINQ provides a high level of abstraction, removing the need for you to worry about the details of constructing an ADO.NET *Command* object, iterating through a result set returned by a *DataReader* object, or fetching data column by column by using the various *GetXXX* methods.

Defining an Entity Class

You saw in Chapter 20 that using LINQ requires the objects that you are querying be enumerable; they must be collections that implement the *IEnumerable* interface. DLINQ can create its own enumerable collections of objects based on classes you define and that map directly to tables in a database. These classes are called *entity classes*. When you connect to a database and perform a query, DLINQ can retrieve the data identified by your query and create an instance of an entity class for each row fetched.

The best way to explain DLINQ is to see an example. The *Products* table in the Northwind database contains columns that contain information about the different aspects of the various products that Northwind Traders sells. The part of the instnwnd.sql script that you ran in the first exercise in this chapter contains a *CREATE TABLE* statement that looks similar to this (some of the columns, constraints, and other details have been omitted):

```
CREATE TABLE "Products" (
    "ProductID" "int" NOT NULL ,
    "ProductName" nvarchar (40) NOT NULL ,
    "SupplierID" "int" NULL ,
    "UnitPrice" "money" NULL,
    CONSTRAINT "PK_Products" PRIMARY KEY CLUSTERED ("ProductID"),
    CONSTRAINT "FK_Products_Suppliers" FOREIGN KEY ("SupplierID")
        REFERENCES "dbo"."Suppliers" ("SupplierID")
)
```

You can define an entity class that corresponds to the *Products* table like this:

```
[Table(Name = "Products")]
public class Product
{
    [Column(IsPrimaryKey = true, CanBeNull = false)]
    public int ProductID { get; set; }

    [Column(CanBeNull = false)]
    public string ProductName { get; set; }
```

```
    [Column]
    public int? SupplierID { get; set; }

    [Column(DbType = "money")]
    public decimal? UnitPrice { get; set; }
}
```

The *Product* class contains a property for each of the columns in which you are interested in the *Products* table. You don't have to specify every column from the underlying table, but any columns that you omit will not be retrieved when you execute a query based on this entity class. The important points to note are the *Table* and *Column* attributes.

The *Table* attribute identifies this class as an entity class. The *Name* parameter specifies the name of the corresponding table in the database. If you omit the *Name* parameter, DLINQ assumes that the entity class name is the same as the name of the corresponding table in the database.

The *Column* attribute describes how a column in the *Products* table maps to a property in the *Product* class. The *Column* attribute can take a number of parameters. The ones shown in this example and described in the following list are the most common:

- The *IsPrimaryKey* parameter specifies that the property makes up part of the primary key. (If the table has a composite primary key spanning multiple columns, you should specify the *IsPrimaryKey* parameter for each corresponding property in the entity class.)
- The *DbType* parameter specifies the type of the underlying column in the database. In many cases, DLINQ can detect and convert data in a column in the database to the type of the corresponding property in the entity class, but in some situations you need to specify the data type mapping yourself. For example, the *UnitPrice* column in the *Products* table uses the SQL Server *money* type. The entity class specifies the corresponding property as a *decimal* value.

Note The default mapping of *money* data in SQL Server is to the *decimal* type in an entity class, so the *DbType* parameter shown here is actually redundant. However, I wanted to show you the syntax.

- The *CanBeNull* parameter indicates whether the column in the database can contain a null value. The default value for the *CanBeNull* parameter is *true*. Notice that the two properties in the *Product* table that correspond to columns that permit null values in the database (*SupplierID* and *UnitPrice*) are defined as nullable types in the entity class.

Note You can also use DLINQ to create new databases and tables based on the definitions of your entity classes by using the *CreateDatabase* method of the *DataContext* object. In the current version of DLINQ, the part of the library that creates tables uses the definition of the *DbType* parameter to specify whether a column should allow null values. If you are using DLINQ to create a new database, you should specify the nullability of each column in each table in the *DbType* parameter, like this:

```
[Column(DbType = "NVarChar(40) NOT NULL", CanBeNull = false)]
public string ProductName { get; set; }
...
[Column(DbType = "Int NULL", CanBeNull = true)]
public int? SupplierID { get; set; }
```

Like the *Table* attribute, the *Column* attribute provides a *Name* parameter that you can use to specify the name of the underlying column in the database. If you omit this parameter, DLINQ assumes that the name of the column is the same as the name of the property in the entity class.

Creating and Running a DLINQ Query

Having defined an entity class, you can use it to fetch and display data from the *Products* table. The following code shows the basic steps for doing this:

```
DataContext db = new DataContext("Integrated Security=true;" +
    "Initial Catalog=Northwind;Data Source=YourComputer\\SQLExpress");

Table<Product> products = db.GetTable<Product>();
var productsQuery = from p in products
                    select p;

foreach (var product in productsQuery)
{
    Console.WriteLine("ID: {0}, Name: {1}, Supplier: {2}, Price: {3:C}",
                      product.ProductID, product.ProductName,
                      product.SupplierID, product.UnitPrice);
}
```

Note Remember that the keywords *from*, *in*, and *select* in this context are C# identifiers. You must type them in lowercase.

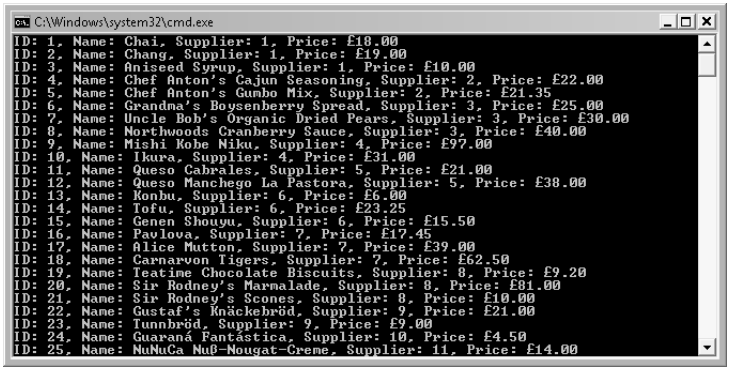
The *DataContext* class is responsible for managing the relationship between your entity classes and the tables in the database. You use it to establish a connection to the database and create collections of the entity classes. The *DataContext* constructor expects a connection string as a parameter, specifying the database that you want to use. This connection string is exactly the same as the connection string that you would use when connecting

through an ADO.NET *Connection* object. (The *DataContext* class actually creates an ADO.NET connection behind the scenes.)

The generic *GetTable<TEntity>* method of the *DataContext* class expects an entity class as its *TEntity* type parameter. This method constructs an enumerable collection based on this type and returns the collection as a *Table<TEntity>* type. You can perform DLINQ queries over this collection. The query shown in this example simply retrieves every object from the *Products* table.

Note If you need to recap your knowledge of LINQ query expressions, turn back to Chapter 20.

The *foreach* statement iterates through the results of this query and displays the details of each product. The following image shows the results of running this code. (The prices shown are per case, not per individual item.)



The *DataContext* object controls the database connection automatically; it opens the connection immediately prior to fetching the first row of data in the *foreach* statement and then closes the connection after the last row has been retrieved.

The DLINQ query shown in the preceding example retrieves every column for every row in the *Products* table. In this case, you can actually iterate through the *products* collection directly, like this:

```
Table<Product> products = db.GetTable<Product>();

foreach (Product product in products)
{
    ...
}
```

When the *foreach* statement runs, the *DataContext* object constructs a SQL SELECT statement that simply retrieves all the data from the *Products* table. If you want to retrieve a single row in the *Products* table, you can call the *Single* method of the *Products* entity class.

Single is an extension method that itself takes a method that identifies the row you want to find and returns this row as an instance of the entity class (as opposed to a collection of rows in a *Table* collection). You can specify the method parameter as a lambda expression. If the lambda expression does not identify exactly one row, the *Single* method returns an *InvalidOperationException*. The following code example queries the Northwind database for the product with the *ProductID* value of 27. The value returned is an instance of the *Product* class, and the *Console.WriteLine* statement prints the name of the product. As before, the database connection is opened and closed automatically by the *DataContext* object.

```
Product singleProduct = products.Single(p => p.ProductID == 27);
Console.WriteLine("Name: {0}", singleProduct.ProductName);
```

Deferred and Immediate Fetching

An important point to emphasize is that by default, DLINQ retrieves the data from the database only when you request it and not when you define a DLINQ query or create a *Table* collection. This is known as deferred fetching. In the example shown earlier that displays all of the products from the *Products* table, the *productsQuery* collection is populated only when the *foreach* loop runs. This mode of operation matches that of LINQ when querying in-memory objects; you will always see the most up-to-date version of the data, even if the data changes after you have run the statement that creates the *productsQuery* enumerable collection.

When the *foreach* loop starts, DLINQ creates and runs a SQL SELECT statement derived from the DLINQ query to create an ADO.NET *DataReader* object. Each iteration of the *foreach* loop performs the necessary *GetXXX* methods to fetch the data for that row. After the final row has been fetched and processed by the *foreach* loop, DLINQ closes the database connection.

Deferred fetching ensures that only the data an application actually uses is retrieved from the database. However, if you are accessing a database running on a remote instance of SQL Server, fetching data row by row does not make the best use of network bandwidth. In this scenario, you can fetch and cache all the data in a single network request by forcing immediate evaluation of the DLINQ query. You can do this by calling the *ToList* or *ToArray* extension methods, which fetch the data into a list or array when you define the DLINQ query, like this:

```
var productsQuery = from p in products.ToList()
                    select p;
```

In this code example, *productsQuery* is now an enumerable list, populated with information from the *Products* table. When you iterate over the data, DLINQ retrieves it from this list rather than sending fetch requests to the database.

Joining Tables and Creating Relationships

DLINQ supports the *join* query operator for combining and retrieving related data held in multiple tables. For example, the *Products* table in the Northwind database holds the ID of the supplier for each product. If you want to know the name of each supplier, you have to query the *Suppliers* table. The *Suppliers* table contains the *CompanyName* column, which specifies the name of the supplier company, and the *ContactName* column, which contains the name of the person in the supplier company that handles orders from Northwind Traders. You can define an entity class containing the relevant supplier information like this (the *SupplierName* column in the database is mandatory, but the *ContactName* allows null values):

```
[Table(Name = "Suppliers")]
public class Supplier
{
    [Column(IsPrimaryKey = true, CanBeNull = false)]
    public int SupplierID { get; set; }

    [Column(CanBeNull = false)]
    public string CompanyName { get; set; }

    [Column]
    public string ContactName { get; set; }
}
```

You can then instantiate *Table*<*Product*> and *Table*<*Supplier*> collections and define a DLINQ query to join these tables together, like this:

```
DataContext db = new DataContext(...);
Table<Product> products = db.GetTable<Product>();
Table<Supplier> suppliers = db.GetTable<Supplier>();
var productsAndSuppliers = from p in products
                           join s in suppliers
                           on p.SupplierID equals s.SupplierID
                           select new { p.ProductName, s.CompanyName, s.ContactName };
```

When you iterate through the *productsAndSuppliers* collection, DLINQ will execute a SQL SELECT statement that joins the *Products* and *Suppliers* tables in the database over the *SupplierID* column in both tables and fetches the data.

However, with DLINQ you can specify the relationships between tables as part of the definition of the entity classes. DLINQ can then fetch the supplier information for each product automatically without requiring that you code a potentially complex and error-prone *join* statement. Returning to the products and suppliers example, these tables have a many-to-one relationship in the Northwind database; each product is supplied by a single supplier, but a single supplier can supply several products. Phrasing this relationship slightly differently, a row in the *Product* table can reference a single row in the *Suppliers* table through the *SupplierID* columns in both tables, but a row in the *Suppliers* table can reference a whole set

of rows in the *Products* table. DLINQ provides the *EntityRef<TEntity>* and *EntitySet<TEntity>* generic types to model this type of relationship. Taking the *Product* entity class first, you can define the "one" side of the relationship with the *Supplier* entity class by using the *EntityRef<Supplier>* type, as shown here in bold type:

```
[Table(Name = "Products")]
public class Product
{
    [Column(IsPrimaryKey = true, CanBeNull = false)]
    public int ProductID { get; set; }
    ...
    [Column]
    public int? SupplierID { get; set; }
    ...
    private EntityRef<Supplier> supplier;
    [Association(Storage = "supplier", ThisKey = "SupplierID", OtherKey = "SupplierID")]
    public Supplier Supplier
    {
        get { return this.supplier.Entity; }
        set { this.supplier.Entity = value; }
    }
}
```

The private *supplier* field is a reference to an instance of the *Supplier* entity class. The public *Supplier* property provides access to this reference. The *Association* attribute specifies how DLINQ locates and populates the data for this property. The *Storage* parameter identifies the *private* field used to store the reference to the *Supplier* object. The *ThisKey* parameter indicates which property in the *Product* entity class DLINQ should use to locate the *Supplier* to reference for this product, and the *OtherKey* parameter specifies which property in the *Supplier* table DLINQ should match against the value for the *ThisKey* parameter. In this example, The *Product* and *Supplier* tables are joined across the *SupplierID* property in both entities.

Note The *Storage* parameter is actually optional. If you specify it, DLINQ accesses the corresponding data member directly when populating it rather than going through the *set* accessor. The *set* accessor is required for applications that manually fill or change the entity object referenced by the *EntityRef<TEntity>* property. Although the *Storage* parameter is actually redundant in this example, it is recommended practice to include it.

The *get* accessor in the *Supplier* property returns a reference to the *Supplier* entity by using the *Entity* property of the *EntityRef<Supplier>* type. The *set* accessor populates this property with a reference to a *Supplier* entity.

You can define the "many" side of the relationship in the *Supplier* class with the *EntitySet<Product>* type, like this:

```
[Table(Name = "Suppliers")]
public class Supplier
{
    [Column(IsPrimaryKey = true, CanBeNull = false)]
    public int SupplierID { get; set; }
    ...
    private EntitySet<Product> products = null;
    [Association(Storage = "products", OtherKey = "SupplierID", ThisKey = "SupplierID")]
    public EntitySet<Product> Products
    {
        get { return this.products; }
        set { this.products.Assign(value); }
    }
}
```

Tip It is conventional to use a singular noun for the name of an entity class and its properties. The exception to this rule is that *EntitySet<TEntity>* properties typically take the plural form because they represent a collection rather than a single entity.

This time, notice that the *Storage* parameter of the *Association* attribute specifies the private *EntitySet<Product>* field. An *EntitySet<TEntity>* object holds a collection of references to entities. The *get* accessor of the public *Products* property returns this collection. The *set* accessor uses the *Assign* method of the *EntitySet<Product>* class to populate this collection.

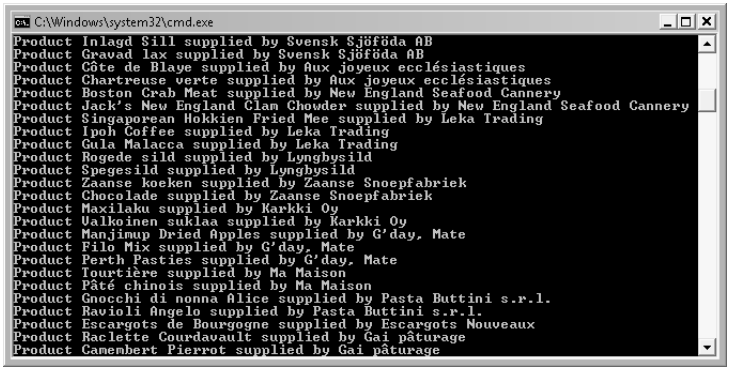
So, by using the *EntityRef<TEntity>* and *EntitySet<TEntity>* types you can define properties that can model a one-to-many relationship, but how do you actually fill these properties with data? The answer is that DLINQ fills them for you when it fetches the data. The following code creates an instance of the *Table<Product>* class and issues a DLINQ query to fetch the details of all products. This code is similar to the first DLINQ example you saw earlier. The difference is in the *foreach* loop that displays the data.

```
DataContext db = new DataContext(...);
Table<Product> products = db.GetTable<Product>();

var productsAndSuppliers = from p in products
                           select p;

foreach (var product in productsAndSuppliers)
{
    Console.WriteLine("Product {0} supplied by {1}",
        product.ProductName, product.Supplier.CompanyName);
}
```

The *Console.WriteLine* statement reads the value in the *ProductName* property of the product entity as before, but it also accesses the *Supplier* entity and displays the *CompanyName* property from this entity. If you run this code, the output looks like this:



As the code fetches each *Product* entity, DLINQ executes a second, deferred, query to retrieve the details of the supplier for that product so that it can populate the *Supplier* property, based on the relationship specified by the *Association* attribute of this property in the *Product* entity class.

When you have defined the *Product* and *Supplier* entities as having a one-to-many relationship, similar logic applies if you execute a DLINQ query over the *Table<Supplier>* collection, like this:

```
DataContext db = new DataContext(...);
Table<Supplier> suppliers = db.GetTable<Supplier>();
var suppliersAndProducts = from s in suppliers
                           select s;

foreach (var supplier in suppliersAndProducts)
{
    Console.WriteLine("Supplier name: {0}", supplier.CompanyName);
    Console.WriteLine("Products supplied");
    foreach (var product in supplier.Products)
    {
        Console.WriteLine("\t{0}", product.ProductName);
    }
    Console.WriteLine();
}
```

In this case, when the *foreach* loop fetches a supplier, it runs a second query (again deferred) to retrieve all the products for that supplier and populate the *Products* property. This time, however, the property is a collection (an *EntitySet<Product>*), so you can code a nested

foreach statement to iterate through the set, displaying the name of each product. The output of this code looks like this:

Deferred and Immediate Fetching Revisited

Earlier in this chapter, I mentioned that DLINQ defers fetching data until the data is actually requested but that you could apply the *ToList* or *ToArray* extension method to retrieve data immediately. This technique does not apply to data referenced as *EntitySet<TEntity>* or *EntityRef<TEntity>* properties; even if you use *ToList* or *ToArray*, the data will still be fetched only when accessed. If you want to force DLINQ to query and fetch referenced data immediately, you can set the *LoadOptions* property of the *DataContext* object as follows:

```
DataContext db = new DataContext(...);
Table<Supplier> suppliers = db.GetTable<Supplier>();
DataLoadOptions loadOptions = new DataLoadOptions();
loadOptions.LoadWith<Supplier>(s => s.Products);
db.LoadOptions = loadOptions;
var suppliersAndProducts = from s in suppliers
                           select s;
```

The *DataLoadOptions* class provides the generic *LoadWith* method. By using this method, you can specify whether an *EntitySet<TEntity>* property in an instance should be loaded when the instance is populated. The parameter to the *LoadWith* method is another method, which you can supply as a lambda expression. The example shown here causes the *Products* property of each *Supplier* entity to be populated as soon as the data for each *Product* entity is fetched rather than being deferred. If you specify the *LoadOptions* property of the *DataContext* object together with the *ToList* or *ToArray* extension method of a *Table* collection, DLINQ will load the entire collection as well as the data for the referenced properties for the entities in that collection into memory as soon as the DLINQ query is evaluated.

Tip If you have several *EntitySet<TEntity>* properties, you can call the *LoadWith* method of the same *LoadOptions* object several times, each time specifying the *EntitySet<TEntity>* to load.

Defining a Custom *DataContext* Class

The *DataContext* class provides functionality for managing databases and database connections, creating entity classes, and executing commands to retrieve and update data in a database. Although you can use the raw *DataContext* class provided with the .NET Framework, it is better practice to use inheritance and define your own specialized version that declares the various *Table<TEntity>* collections as public members. For example, here is a specialized *DataContext* class that exposes the *Products* and *Suppliers Table* collections as public members:

```
public class Northwind : DataContext
{
    public Table<Product> Products;
    public Table<Supplier> Suppliers;

    public Northwind(string connectionInfo) : base(connectionInfo)
    {
    }
}
```

Notice that the *Northwind* class also provides a constructor that takes a connection string as a parameter. You can create a new instance of the *Northwind* class and then define and run DLINQ queries over the *Table* collection classes it exposes like this:

```
Northwind nwindDB = new Northwind(...);

var suppliersQuery = from s in nwindDB.Suppliers
                     select s;

foreach (var supplier in suppliersQuery)
{
    ...
}
```

This practice makes your code easier to maintain, especially if you are retrieving data from multiple databases. Using an ordinary *DataContext* object, you can instantiate any entity class by using the *GetTable* method, regardless of the database to which the *DataContext* object connects. You find out that you have used the wrong *DataContext* object and have connected to the wrong database only at run time, when you try to retrieve data. With a custom *DataContext* class, you reference the *Table* collections through the *DataContext* object. (The *base DataContext* constructor uses a mechanism called *reflection* to examine its members, and it automatically instantiates any members that are *Table* collections—the details of how

reflection works are outside the scope of this book.) It is obvious to which database you need to connect to retrieve data for a specific table; if IntelliSense does not display your table when you define the DLINQ query, you have picked the wrong *DataContext* class, and your code will not compile.

Using DLINQ to Query Order Information

In the following exercise, you will write a version of the console application that you developed in the preceding exercise that prompts the user for a customer ID and displays the details of any orders placed by that customer. You will use DLINQ to retrieve the data. You will then be able to compare DLINQ with the equivalent code written by using ADO.NET.

Define the *Order* entity class

1. Using Visual Studio 2008, create a new project called DLINQOrders by using the Console Application template. Save it in the \Microsoft Press\Visual CSharp Step By Step\Chapter 25 folder under your Documents folder, and then click *OK*.
2. In *Solution Explorer*, change the name of the file Program.cs to DLINQReport.cs. In the *Microsoft Visual Studio* message, click *Yes* to change all references of the *Program* class to *DLINQReport*.
3. On the *Project* menu, click *Add Reference*. In the *Add Reference* dialog box, click the *.NET* tab, select the *System.Data.Linq* assembly, and then click *OK*.

   This assembly holds the DLINQ types and attributes.
4. In the *Code and Text Editor* window, add the following *using* statements to the list at the top of the file:

   ```
   using System.Data.Linq;
   using System.Data.Linq.Mapping;
   using System.Data.SqlClient;
   ```
5. Add the *Order* entity class to the DLINQReport.cs file after the *DLINQReport* class, as follows:

   ```
   [Table(Name = "Orders")]
   public class Order
   {
   }
   ```

   The table is called *Orders* in the Northwind database. Remember that it is common practice to use the singular noun for the name of an entity class because an entity object represents one row from the database.

6. Add the property shown here in bold type to the *Order* class:

```
[Table(Name = "Orders")]
public class Order
{
    [Column(IsPrimaryKey = true, CanBeNull = false)]
    public int OrderID { get; set; }
}
```

The *OrderID* column is the primary key for this table in the Northwind database.

7. Add the following properties shown in bold type to the *Order* class:

```
[Table(Name = "Orders")]
public class Order
{
    ...
    [Column]
    public string CustomerID { get; set; }

    [Column]
    public DateTime? OrderDate { get; set; }

    [Column]
    public DateTime? ShippedDate { get; set; }

    [Column]
    public string ShipName { get; set; }

    [Column]
    public string ShipAddress { get; set; }

    [Column]
    public string ShipCity { get; set; }

    [Column]
    public string ShipCountry { get; set; }
}
```

These properties hold the customer ID, order date, and shipping information for an order. In the database, all of these columns allow null values, so it is important to use the nullable version of the *DateTime* type for the *OrderDate* and *ShippedDate* properties (*string* is a reference type that automatically allows null values). Notice that DLINQ automatically maps the SQL Server *NVarChar* type to the .NET Framework *string* type and the SQL Server *DateTime* type to the .NET Framework *DateTime* type.

8. Add the following *Northwind* class to the DLINQReport.cs file after the *Order* entity class:

```
public class Northwind : DataContext
{
    public Table<Order> Orders;
```

```
    public Northwind(string connectionInfo) : base (connectionInfo)
    {
    }
}
```

The *Northwind* class is a *DataContext* class that exposes a *Table* property based on the *Order* entity class. In the next exercise, you will use this specialized version of the *DataContext* class to access the *Orders* table in the database.

Retrieve order information by using a DLINQ query

1. In the *Main* method of the *DLINQReport* class, add the statement shown here in bold type, which creates a *Northwind* object. Be sure to replace *YourComputer* with the name of your computer:

```
static void Main(string[] args)
{
    Northwind northwindDB = new Northwind("Integrated Security=true;" +
        "Initial Catalog=Northwind;Data Source=YourComputer\\SQLExpress");
}
```

The connection string specified here is exactly the same as in the earlier exercise. The *northwindDB* object uses this string to connect to the Northwind database.

2. After the variable declaration, add a *try/catch* block to the *Main* method:

```
static void Main(string[] args)
{
    ...
    try
    {
        // You will add your code here in a moment
    }
    catch(SqlException e)
    {
        Console.WriteLine("Error accessing the database: {0}", e.Message);
    }
}
```

As when using ordinary ADO.NET code, DLINQ raises a *SqlException* if an error occurs when accessing a SQL Server database.

3. Replace the comment in the *try* block with the following code shown in bold type:

```
try
{
    Console.Write("Please enter a customer ID (5 characters): ");
    string customerId = Console.ReadLine();
}
```

These statements prompt the user for a customer ID and save the user's response in the string variable *customerId*.

4. Type the statement shown here in bold type after the code you just entered:

```
try
{
    ...
    var ordersQuery = from o in northwindDB.Orders
                      where String.Equals(o.CustomerID, customerId)
                      select o;
}
```

This statement defines the DLINQ query that will retrieve the orders for the specified customer.

5. Add the *foreach* statement and *if...else* block shown here in bold type after the code you added in the preceding step:

```
try
{
    ...
    foreach (var order in ordersQuery)
    {
        if (order.ShippedDate == null)
        {
            Console.WriteLine("Order {0} not yet shipped\n\n", order.OrderID);
         }
         else
         {
             // Display the order details
         }
    }
}
```

The *foreach* statement iterates through the orders for the customer. If the value in the *ShippedDate* column in the database is *null*, the corresponding property in the *Order* entity object is also *null*, and then the *if* statement outputs a suitable message.

6. Replace the comment in the *else* part of the *if* statement you added in the preceding step with the code shown here in bold type:

```
if (order.ShippedDate == null)
{
    ...
}
else
{
    Console.WriteLine("Order: {0}\nPlaced: {1}\nShipped: {2}\n" +
                      "To Address: {3}\n{4}\n{5}\n{6}\n\n", order.OrderID,
                       order.OrderDate, order.ShippedDate, order.ShipName,
                       order.ShipAddress, order.ShipCity,
                       order.ShipCountry);
}
```

7. On the *Debug* menu, click *Start Without Debugging* to build and run the application.
8. In the console window displaying the message "Please enter a customer ID (5 characters):", type **VINET**.

   The application should display a list of orders for this customer. When the application has finished, press Enter to return to Visual Studio 2008.
9. Run the application again. This time type **BONAP** when prompted for a customer ID.

   The final order for this customer has not yet shipped and contains a null value for the *ShippedDate* column. Verify that the application detects and handles this null value. When the application has finished, press Enter to return to Visual Studio 2008.

You have now seen the basic elements that DLINQ provides for querying information from a database. DLINQ has many more features that you can employ in your applications, including the ability to modify data and update a database. You will look briefly at some of these aspects of DLINQ in the next chapter.

- If you want to continue to the next chapter

  Keep Visual Studio 2008 running, and turn to Chapter 26.
- If you want to exit Visual Studio 2008 now

  On the *File* menu, click *Exit*. If you see a *Save* dialog box, click *Yes* (if you are using Visual Studio 2008) or *Save* (if you are using Visual C# 2008 Express Edition) and save the project.

Chapter 25 Quick Reference

To	Do this
Connect to a SQL Server database by using ADO.NET	Create a *SqlConnection* object, set its *ConnectionString* property with details specifying the database to use, and call the *Open* method.
Create and execute a database query by using ADO.NET	Create a *SqlCommand* object. Set its *Connection* property to a valid *SqlConnection* object. Set its *CommandText* property to a valid SQL SELECT statement. Call the *ExecuteReader* method to run the query and create a *SqlDataReader* object.
Fetch data by using an ADO.NET *SqlDataReader* object	Ensure that the data is not null by using the *IsDBNull* method. If the data is not null, use the appropriate *GetXXX* method (such as *GetString* or *GetInt32*) to retrieve the data.

Define an entity class	Define a class with public properties for each column. Prefix the class definition with the *Table* attribute, specifying the name of the table in the underlying database. Prefix each property with the *Column* attribute, and specify parameters indicating the name, type, and nullability of the corresponding column in the database.
Create and execute a query by using DLINQ	Create a *DataContext* variable, and specify a connection string for the database. Create a *Table* collection variable based on the entity class corresponding to the table you want to query. Define a DLINQ query that identifies the data to be retrieved from the database and returns an enumerable collection of entities. Iterate through the enumerable collection to retrieve the data for each row and process the results.