CHAPTER 4

Database Access in Code

The previous three chapters covered the fundamentals required to start using databases. Chapter 1 showed the various types of data sources available, and you saw how you can use text files, XML files, spreadsheets, and the Active Directory store, not to mention *real* databases as data sources. Chapter 2 then moved on to talk about relational databases, and by the end of the chapter, you had fundamentally the same database in SQL Server 2005, MySQL 5.0, and Microsoft Access.

In Chapter 3, you took the first look at actually using the database in a page. You used a `GridView` to display the results returned from a `SqlDataSource` using three different databases: SQL Server 2005 using the native SQL Server provider, MySQL 5.0 using an ODBC driver, and Microsoft Access using an OLE DB provider. In all three cases, the returned results were displayed in a tabular format by the `GridView`.

But as you learned in Chapter 1, this isn't the end of the story. You have several objects that you can use to communicate with the database: Command, Connection, Parameter, DataReader, and DataAdapter. The approach you took in Chapter 3 completely hid these, and wrapped all the requests to the database inside the `SqlDataSource`. You created the correct `SqlDataSource` and `GridView` controls, and ASP.NET performed all the necessary data access automatically.

You do, however, need to be able to interact with the database in code, as there are certain things that you can't do using the `SqlDataSource`. In this chapter, we'll look at dealing directly with the database in code, rather than relying on ASP.NET to do all of this for you.

This chapter covers the following topics:

- The connection and command life cycle
- How to connect to databases using the correct Connection object (the `SqlConnection`, `OdbcConnection`, and `OleDbConnection` objects)
- Connection pooling to reuse database connections and improve performance
- How to use the Command object to modify the query that is being executed based on the user's actions
- How to use parameters to change the query being executed
- The SQL scalar functions that you can use to return information from a database
- Error handling in code to access a database

The Connection and Command Life Cycle

We can summarize the life cycle for connecting to a data source and executing queries against it as follows:

- Create the Connection object and specify the data source.
- Create the Command object.
- Tell the Command object which Connection to use.
- Specify the query to execute and pass to the Command object.
- Open the connection to the data source.
- Execute a query against the data source.
- If there are query results, you may need to do something with them.
- Close the connection to the data source.

In practice, the sequence in this list is what would ideally happen, but it isn't always the case. You can perform several of the tasks in the life cycle list in a slightly different order without causing any problems. As with most coding tasks, you can do these in several slightly different ways, and usually none of them is more correct than the others.

We're going to look at this life cycle in three stages. We'll first look at the Connection object and how to configure this to connect to the data source, and how the connection is opened and closed. Next, we'll deal with the Command object and how to configure it to use a specific Connection object and a given query. You'll see that there are several different ways that you can execute the query, depending on what the query does. The queries that we've looked at in Chapter 3 have always returned a set of results (a list of Manufacturers, for instance), and we'll spend some time looking at how to connect to the database and return a DataReader containing the set of results using the `ExecuteReader()` method.

■ **Note** As you saw in Chapter 1, you can also return data using a DataAdapter to populate a `DataSet`. The DataReader is the easier method of accessing the database, and we'll concentrate on it in this chapter. In Chapter 5, we'll look at the differences between the DataReader and the `DataSet`, and in Chapter 6, we'll start using the DataAdapter and `DataSet` objects to query the database.

As you'll see, not all queries to a database produce a set of results. You can write queries that only return single values, most commonly when returning the result from a scalar function, and we'll spend some time looking at scalar functions and how you can use the `ExecuteScalar()` method to retrieve these values from the database.

You can also write queries that don't return any results at all. Any `INSERT`, `UPDATE`, or `DELETE` queries that you execute against the database won't return any results. You use a third method, `ExecuteNonQuery()`, to execute those queries. You'll see how to use `ExecuteNonQuery()` in Chapter 8.

Connection Objects

As you saw in Chapter 1, the data provider for a particular data source contains implementations of several objects. Each of these objects handles a specific task, and a Connection object, unsurprisingly, handles a connection to a data source. The Connection object (as was the case with traditional ADO) is the basis for all interactions with the data source you want to use. You must open the connection before you access the data source, and you must close it when you're finished.

■**Note** Unlike with traditional ADO, you must always create a Connection object when talking to a data source. With ADO, you could pass an ADODB.Connection object or a connection string to an ADODB. Command object, but when using ADO.NET, you must create an instance of a Connection object and pass this to the Command object.

Again, as you saw in Chapter 1, the data provider architecture allows a data provider to be specifically designed for a data source. Here, we'll look at three implementations of the Connection object:

- The `SqlConnection` object to connect to a SQL Server database
- The `OdbcConnection` object to connect to a data source using an ODBC driver
- The `OleDbConnection` object to connect to a data source using an OLE DB provider

---

■**Note** If you're trying to connect to SQL Server version 6.5, you can't use the `SqlConnection` object because it works with only SQL Server version 7.0 and newer. With SQL Server 6.5, use the `OleDbConnection` object and the OLE DB provider for SQL Server.

Try It Out: Connecting to SQL Server 2005 Using SqlConnection

To connect to a SQL Server 2005 database, you use the `SqlConnection` object. As this is part of the `SqlClient` data provider, you can assume that it's the quickest way of accessing the database.

In this example, you'll use the `SqlConnection` object to connect to the database and the `SqlCommand` object to return all of the Players that are in the database.

1. Start Visual Web Developer and create a new Web site in the `C:\BAND\Chapter04` folder. Delete the auto-created `Default.aspx` file.
2. Add a new `Web.config` file by selecting Add New Item from the Web site's context menu. Click Web Configuration File, and then click the Add button.
3. Find the `<connectionStrings />` element in `Web.config` and replace it with the following:

```
<connectionStrings>
  <add name="SqlConnectionString"
    connectionString="Data Source=localhost\BAND;Initial Catalog=Players;
    User ID=band;Password=letmein" />
</connectionStrings>
```

4. Add a Web Form by selecting the Add New Item option from the Web site's context menu. Click Web Form and make sure that the Place Code in Separate File option is unselected and Visual C# is selected as the Language. Give the page a name of `Select.aspx` and click the Add button.

5. In the Source view of the page, change the `<TITLE>` element to **Displaying Data with SqlClient**. Then add the following to the top of the page after the `<% Page %>` tag:

```
<%@ Import Namespace="System.Data" %>
<%@ Import Namespace="System.Data.SqlClient" %>
```

6. Switch to the Design view of the page and add a `GridView` from the Data tab of the Toolbox onto the page. (If the Toolbox is not visible, select View ➤ Toolbox.)

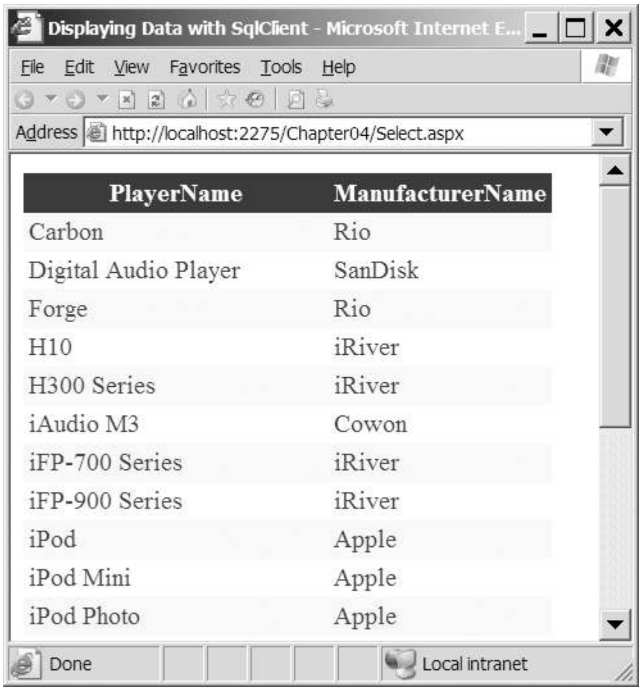
7. From the `GridView` Tasks menu, select AutoFormat and choose the Colorful scheme. (If you don't like that scheme, choose a different one.)

8. Double-click somewhere on the page that isn't the `GridView` to add a `Page_Load` event to the page. Add the following code to the event handler:

```
protected void Page_Load(object sender, EventArgs e)
{
  if (Page.IsPostBack == false)
  {
    // create the connection
    string strConnectionString = ConfigurationManager.
      ConnectionStrings["SqlConnectionString"].ConnectionString;
    SqlConnection myConnection =
      new SqlConnection(strConnectionString);

    // create the command
    string strCommandText = "SELECT Player.PlayerName, ➥
      Manufacturer.ManufacturerName FROM Player INNER JOIN ➥
      Manufacturer ON Player.PlayerManufacturerID = ➥
      Manufacturer.ManufacturerID ORDER BY Player.PlayerName";
    SqlCommand myCommand =
      new SqlCommand(strCommandText, myConnection);

    // open the database connection
    myConnection.Open();
```

```
        // show the data
        GridView1.DataSource = myCommand.ExecuteReader();
        GridView1.DataBind();

        // close the database connection
        myConnection.Close();
      }
    }
```

9. In the Solution Explorer, right-click `Select.aspx` and select Set As Start Page.
10. Run the Web site. If you're presented with the Debugging Not Enabled dialog box, click OK to modify `Web.config` and start debugging. This will load your browser and display the page as shown in Figure 4-1.


Displaying Data with SqlClient - Microsoft Internet E...
File Edit View Favorites Tools Help
Address http://localhost:2275/Chapter04/Select.aspx
Done
Local intranet

PlayerName	ManufacturerName
Carbon	Rio
Digital Audio Player	SanDisk
Forge	Rio
H10	iRiver
H300 Series	iRiver
iAudio M3	Cowon
iFP-700 Series	iRiver
iFP-900 Series	iRiver
iPod	Apple
iPod Mini	Apple
iPod Photo	Apple

Figure 4-1. *Results of the query using the SqlConnection object*

How It Works

That wasn't too hard was it? You needed only seven lines of code to connect to the database, retrieve the data, and then bind the results returned to the `GridView` for display. Let's start by looking at using a `SqlConnection`.

First, you added a connection string for the SQL Server database that you'll use in the `connectionStrings` section of `Web.config`:

```
<add name="SqlConnectionString"
  connectionString="Data Source=localhost\BAND;Initial Catalog=Players;
  User ID=band;Password=letmein"/>
```

This is similar to the connection strings that you used in Chapter 3. The only difference is the absence of the providerName property here. The providerName property is used by the SqlDataSource to decide which Connection and Command objects to use, but as you're manually specifying the Connection and Command objects in this example, you don't need this property. To use the same connection string in both situations, you can simply add the correct providerName to the connection string.

Before you can use any of the database objects, you must reference the namespaces that contain those objects. You could use fully referenced names to refer to them, but this requires a lot of typing and some unwieldy lines of code. As you saw in Chapter 1, a general namespace exists for all the data objects and a namespace exists for the SQL Server-specific objects. You included a reference to both of these namespaces, which allows you to refer to the objects using much shorter names:

```
<%@ Import Namespace="System.Data" %
<%@ Import Namespace="System.Data.SqlClient" %>
```

Because the page needs to populate the GridView when the page is loaded, you need to use the page's Page_Load event. The Page_Load event handler executes every time the page is loaded (whether a first view or a postback responding to a request from the user). In data-driven Web sites, you'll want to perform some actions only once, such as populating the GridView, rather than every time the page is loaded. You can check for the type of page request using the IsPostBack property of the Page object, which returns true if there has been a postback because of a user request.

The GridView remembers the data that it's populated with, so you need to populate it only once. First, you check that the page is not responding to a postback from the client by seeing if the IsPostBack property is false. If it is, you're showing the page for the first time, and you execute the code to populate the GridView.

The first line of code retrieves the connection string, SqlConnectionString, that you're going to use from the Web site configuration and stores this in the strConnectionString local variable:

```
string strConnectionString = ConfigurationManager.
  ConnectionStrings["SqlConnectionString"].ConnectionString;
```

The ConfigurationManager.ConnectionStrings property accepts the name of the connection string that you want and returns a ConnectionStringSettings object that contains the details specified in Web.config for the connection string. There's a Name property for the name element, a ProviderName property for the providerName element, and—the one you're after—a ConnectionString property for the connectionString element.

Once the connection string has been populated, it's time to create the SqlConnection object and point it at the correct database. You do this by passing the connection string into the constructor, like so:

```
SqlConnection myConnection = new SqlConnection(strConnectionString);
```

■ **Note** Although you pass the connection string into the constructor to initialize it, you can create a connection without passing in a connection string. In this case, you must set the `ConnectionString` property of the Connection object to the correct connection string before you attempt to open the connection.

The next two lines of code are concerned with the query that you pass to the database and the `SqlCommand` object that you use to actually execute the query. We'll come back to this when we discuss the Command object later in the chapter; for now, just be assured that it works. However, you should recognize the query—you saw it in the previous chapter. It returns all of the Players in the database along with the name of the Manufacturer, in Player name order.

Although you've now created a `SqlConnection` object to connect to the database and the `SqlCommand` object to query the database, you still haven't made the connection to the database. Only a finite number of database connections are available, and you shouldn't open a connection if you're not actually doing anything with it. While you have the connection open, you're preventing everyone else from using that connection. The connection should be opened at the last possible moment by calling the `Open()` method on the `SqlConnection` object, like so:

```
myConnection.Open();
```

Once the connection is opened, you carry out the tasks on the data. In this case, you're doing a little data binding:

```
GridView1.DataSource = myCommand.ExecuteReader();
GridView1.DataBind();
```

Once you're finished with the connection to the database, you should close the connection to the database as soon as it isn't required anymore. Again, keeping it open once you're finished with it prevents anyone else from using that connection. To close the database connection, use the `Close()` method, like so:

```
myConnection.Close();
```

That's all there is to connecting to the database. You've created a connection, opened the connection, and then closed it. It doesn't get any more complex than that.

Try It Out: Connecting to MySQL 5.0 Using OdbcConnection

For some data sources, such as SQL Server, you'll have a native data provider to use. In a production environment, you should always use the specific data provider if one is available. If one isn't available, then you need to use the `OleDb` or `Odbc` data provider with the correct OLE DB provider or ODBC driver for your database.

Indeed, MySQL has its own data provider that you can download from the MySQL Web site at `http://dev.mysql.com/downloads/connector/net/1.0.html`. However, we're going to forgo the native data provider (until Chapter 10 anyway) and connect through the `Odbc` data provider using the MySQL ODBC driver.

Using the `OdbcConnection` object to connect to a MySQL database is no different from using the `SqlConnection` object to talk to a SQL Server database. The same is also true if you were to use the `MySqlClient` data provider (`MySqlConnection`) or `OleDb` data provider (`OleDbConnection`) to connect to using an OLE DB provider. Although you use different objects, the methodology remains the same.

In this example, you'll build the same page as in the previous example, but using the `OdbcConnection` object to talk to a MySQL database. Follow these steps:

1. Start Visual Web Developer and open the Chapter04 Web site from `C:\BAND\Chapter04`.

2. Open `Web.config` and add a new connection string to the connection string element:

```
<add name="OdbcConnectionString"
  connectionString="Driver={MySQL ODBC 3.51 Driver};
    server=localhost;database=players;uid=band;pwd=letmein;" />
```

3. Create a new folder called `odbc` and add a new Web Form called `Select.aspx` to the folder.

4. In the Source view of the page, change the `<TITLE>` element to **Displaying Data with Odbc**. Then add the following to the top of the page after the `<% Page %>` tag:

```
<%@ Import Namespace="System.Data" %>
<%@ Import Namespace="System.Data.Odbc" %>
```

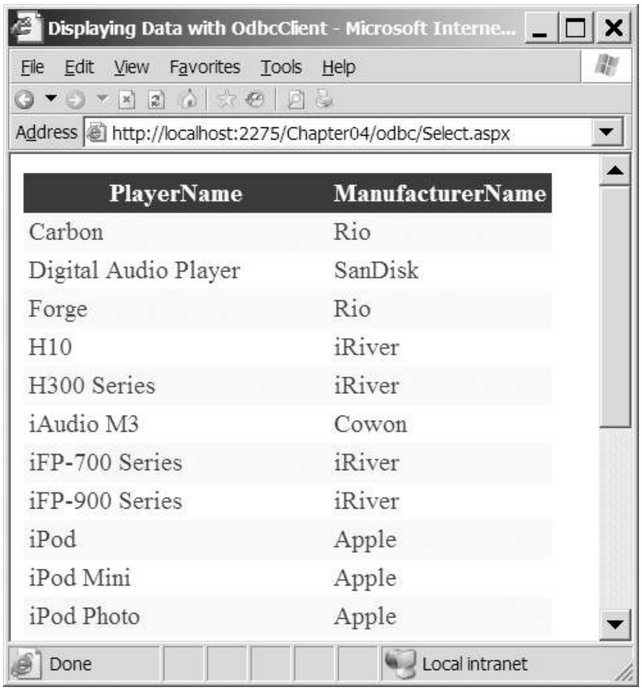
5. Switch to the Design view of the page and add a `GridView` to the page. Set its AutoFormat property to Colorful (or another scheme that takes your fancy).

6. Double-click somewhere on the page that isn't the `GridView` to add a `Load` event to the page. Add the following code to the event:

```
protected void Page_Load(object sender, EventArgs e)
{
  if (Page.IsPostBack == false)
  {
    // create the connection
    string strConnectionString = ConfigurationManager.
      ConnectionStrings["OdbcConnectionString"].ConnectionString;
    OdbcConnection myConnection =
      new OdbcConnection(strConnectionString);

    // create the command
    string strCommandText = "SELECT Player.PlayerName, ➥
      Manufacturer.ManufacturerName FROM Player INNER JOIN ➥
      Manufacturer ON Player.PlayerManufacturerID = ➥
      Manufacturer.ManufacturerID ORDER BY Player.PlayerName";
    OdbcCommand myCommand = new
      OdbcCommand(strCommandText, myConnection);

    // open the database connection
    myConnection.Open();
```

```
        // show the data
        GridView1.DataSource = myCommand.ExecuteReader();
        GridView1.DataBind();

        // close the database connection
        myConnection.Close();
      }
    }
```

7. Right-click `Select.aspx` in the `odbc` folder in the Solution Explorer and select View in Browser. This will launch your browser and display the results for the query, as shown in Figure 4-2.


Displaying Data with OdbcClient - Microsoft Interne...
File Edit View Favorites Tools Help
Address http://localhost:2275/Chapter04/odbc/Select.aspx
Done
Local intranet

PlayerName	ManufacturerName
Carbon	Rio
Digital Audio Player	SanDisk
Forge	Rio
H10	iRiver
H300 Series	iRiver
iAudio M3	Cowon
iFP-700 Series	iRiver
iFP-900 Series	iRiver
iPod	Apple
iPod Mini	Apple
iPod Photo	Apple

Figure 4-2. *Results of the query using the OdbcConnection object*

How It Works

If you compare the code for the previous example and this one, you'll see they're similar.

In order to connect to the MySQL database, you need to add a new connection string to `Web.config`. This time, it's simplicity itself:

```
<add name="OdbcConnectionString"
  connectionString="Driver={MySQL ODBC 3.51 Driver};
    server=localhost;database=players;uid=band;pwd=letmein;" />
```

You create a new connection string with the name OdbcConnectionString and tell it that you're using the MySQL ODBC driver. As you'll recall from Chapter 3, you can use this driver by specifying the server, database, uid, and pwd that you want to use to connect to the database.

First, you include the correct namespaces in the page. You again use System.Data to allow access to the base data objects, but instead of the System.Data.SqlClient namespace, you use the System.Data.Odbc namespace, like so:

```
<%@ Import Namespace="System.Data" %>
<%@ Import Namespace="System.Data.Odbc" %>
```

The connection string is retrieved from Web.config in the same way as the previous example, except that you're retrieving the OdbcConnectionString setting rather than the SqlConnectionString. You store this in the strConnectionString local variable so that you can use it in the constructor of the OdbcConnection object:

```
OdbcConnection myConnection = new OdbcConnection(strConnectionString);
```

You pass in the connection string you want to use as the only parameter to the OdbcConnection object constructor, and you have a properly configured Connection object that you can use. This is exactly the same process as with the SqlConnection object.

As with the first example, the next two lines of code don't really concern us at the moment, but you will notice that you're using the OdbcCommand object rather than the SqlCommand object.

You then open the connection to the database using the Open() method, do the necessary data binding using the OdbcCommand object you created and configured, and then close the database connection using the Close() method.

From this brief description, you can see that the process for creating and using the objects to communicate with MySQL using an ODBC driver is the same as that for communicating with SQL Server. Once you had the correct connection string defined, you could have simply copied the page from the previous example and replaced all references to Sql objects in the Page_Load event with their Odbc equivalents, and the code would have worked perfectly.

As the databases in SQL Server 2005 and MySQL 5.0 contain the exact same data, the results of executing this page are identical to those of the previous example, as you'll see if you compare Figure 4-2 with Figure 4-1.

Try It Out: Connecting to Microsoft Access Using OleDbConnection

The easiest way to connect to an Access database is by using the Microsoft Jet database engine, more commonly known as the Jet engine, which is an OLE DB provider. You can access this provider using the OleDbConnection object. The Jet engine allows you to connect to various other data sources, such as dBASE and Paradox databases, Excel spreadsheets, and text files.

■Note Since the release of MDAC 2.6, the Jet engine isn't installed as standard. Therefore, if you don't have Microsoft Access installed, you may not have it. You can download the latest version of the Jet engine, Service Pack 8, from `http://support.microsoft.com/?kbid=239114`.

In this example, you'll build the same page as in the previous two examples, but this time use the `OleDbConnection` object to talk to a Microsoft Access database. Follow these steps:

1. Start Visual Web Developer and open the Chapter04 Web site from `C:\BAND\Chapter04`.
2. Open `Web.config` and add a new connection string to the connection string element:

   ```
   <add name="OleDbConnectionString"
     connectionString="Provider=Microsoft.Jet.OLEDB.4.0;
     Data Source=C:\BAND\Players.mdb" />
   ```

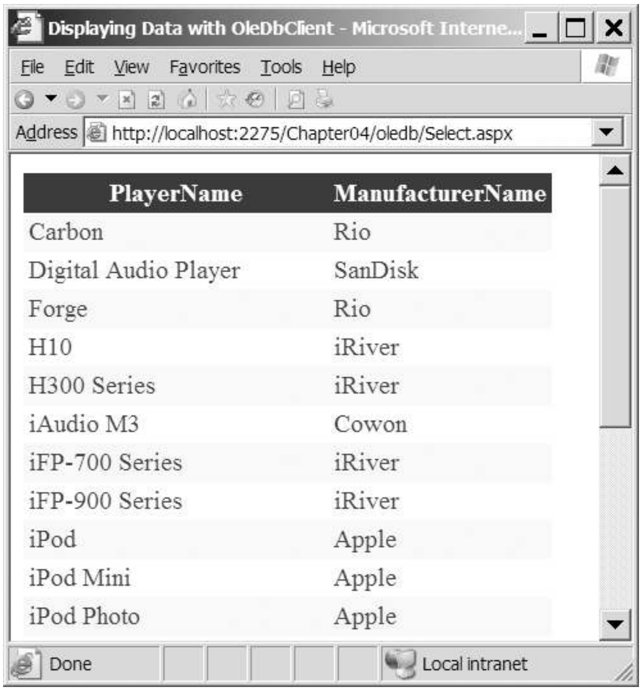
3. Create a new folder called `oledb` and copy the `Select.aspx` page from the `odbc` folder to this new folder.
4. Open the `Select.aspx` page in the `oledb` folder and change the `<TITLE>` element to **Displaying Data with OleDb**.
5. Change the `Import` for the `Odbc` namespace to its `OleDb` equivalent:

   ```
   <%@ Import Namespace="System.Data.OleDb" %>
   ```

6. In the `Page_Load` event, you need to change from using the `Odbc` objects to the `OleDb` equivalents. You also need to use the correct connection string. The changed lines are as follows:

   ```
   string strConnectionString = ConfigurationManager.
     ConnectionStrings["OleDbConnectionString"].ConnectionString;
   OleDbConnection myConnection = new OleDbConnection(strConnectionString);

   ...

   OleDbCommand myCommand = new OleDbCommand(strCommandText, myConnection);
   ```

7. Right-click `Select.aspx` in the `oledb` folder in the Solution Explorer and select View in Browser. This will launch your browser and display the results for the query, as shown in Figure 4-3.


Displaying Data with OleDbClient - Microsoft Interne...
File Edit View Favorites Tools Help
Address http://localhost:2275/Chapter04/oledb/Select.aspx
Done
Local intranet

PlayerName	ManufacturerName
Carbon	Rio
Digital Audio Player	SanDisk
Forge	Rio
H10	iRiver
H300 Series	iRiver
iAudio M3	Cowon
iFP-700 Series	iRiver
iFP-900 Series	iRiver
iPod	Apple
iPod Mini	Apple
iPod Photo	Apple

Figure 4-3. *Results of the query using the OleDbConnection object*

How It Works

This time, you really did just replace the `Odbc` objects with their `OleDb` equivalents to get the code to work with the different database objects—did you really want to build the same exact page again? All you've needed to do was use the correct connection string, and then change from using the `OdbcConnection` and `OdbcCommand` objects to the `OleDbConnection` and `OleDbCommand` objects.

As you can see if you compare the results of all three examples, the page returned is exactly the same, regardless of the data source and method of accessing the data source.

Connection Object Methods and Properties

You've already seen two of the methods of the Connection object, `Open()` and `Close()`. Now, you'll look at a couple of properties that come in quite handy: `ConnectionString` and `State`.

Other properties and methods are available on every implementation of the Connection object, including a common set of properties and methods (inherited from the `DbConnection` abstract class) that each Connection object implements. Additionally, each object can also implement its own properties and methods. Several are rarely used, and others will never be used except in very advanced situations. If you need further details, you can find more information on MSDN at the following locations:

- `SqlConnection`:
  `http://msdn.microsoft.com/en-us/library/system.data.sqlclient.sqlconnection.aspx`

- `OdbcConnection`:
  `http://msdn.microsoft.com/en-us/library/system.data.odbc.odbcconnection.aspx`
- `OleDbConnection`:
  `http://msdn.microsoft.com/en-us/library/system.data.oledb.oledbconnection.aspx`

The ConnectionString Property

In the examples you've looked at so far, you've always created the Connection object by passing the connection string to the constructor. The Connection object also exposes the `ConnectionString` property, which you can use instead to specify the connection string after you've created the Connection object. So, for the first example in this chapter, you could have used the alternative method of setting the connection string, like so:

```
SqlConnection myConnection = new SqlConnection();
myConnection.ConnectionString = strConnectionString;
```

The State Property

One of the more useful properties of the Connection object is the `State` property. This allows you to check the state of the connection to the data source. The `State` property is read-only and may take, in the current release of the .NET runtime, a value of `ConnectionState.Open` or `ConnectionState.Closed`.

Connection Pooling

When connecting to a database, several time-consuming tasks must be performed before the connection can be classed as open. A physical connection to the database is created, the login details parsed and authenticated against the database, and so on. If this occurred every time you connected to the database, the connection preparation time soon mounts up.

In your Web site, you will probably use a limited number of different databases, or maybe only one database. This is where *connection pooling* can help.

Connection pooling works under the covers and creates a pool of connections that are used when you call the `Open()` method on the Connection object. When you call the `Open()` method, the data provider will look to see if there are any connections in the pool. If there are, then a pooled connection will be reused. If not, the data provider will create a new connection to the database. On closing the connection with the `Close()` method, the connection will not actually be closed yet. It will be returned to the pool to be used again on another call to the `Open()` method.

So how do you enable connection pooling? The answer is you don't. If your data provider supports connection pooling, it will be enabled by default. But it does depend on which data provider you're using:

- The `SqlClient` data provider uses connection pooling for all connections to the database.
- The `OleDb` data provider will use connection pooling if the underlying OLE DB provider supports it.
- Connection pooling for ODBC drivers is controlled at the ODBC level and not within the `Odbc` data provider. If the ODBC driver has connection pooling enabled, it will be enabled.

The key to connection pooling is the connection string that you use for the Connection object. Each different connection string has its own connection pool, and if the connection strings you're using differ, even very slightly, connections will not be reused across the connection pools.

So if you have one `SqlConnection` object using this:

```
Data Source=localhost\BAND;Initial Catalog=Players;User ID=band;Password=letmein;
```

And you have another `SqlConnection` object using this:

```
Server=localhost\BAND;database=Players;uid=band;pwd=letmein;
```

The two `SqlConnection` objects will not be able to use the same connection pool, because the connection strings are different—even though they go to the exact same database on the same server with the same security credentials.

For now, you should just be aware that connection pooling can happen. Be sure to keep connections to the database open for the minimum amount of time, by opening the connection at the last opportunity before you need it and closing the connection at the first opportunity once you're finished with it.

Command Objects

In the examples so far, you've used a Connection object to connect to the database. You then have two lines of code that create the correct Command object. For the `SqlConnection` version of the code, the `SqlCommand` object was created as follows:

```
// create the command
string strCommandText = "SELECT Player.PlayerName, ➥
  Manufacturer.ManufacturerName FROM Player INNER JOIN ➥
  Manufacturer ON Player.PlayerManufacturerID = ➥
  Manufacturer.ManufacturerID ORDER BY Player.PlayerName";
SqlCommand myCommand = new SqlCommand(strCommandText, myConnection);
```

In this code, you've simply created a string, `strCommandText`, to hold the query that you want to execute, and passed both `strCommandText` and `myConnection` (the connection to the database that you created) to the constructor of the `SqlCommand` object.

In this section, the discussion will focus on the `SqlClient` data provider. However, everything that's discussed in relation to `SqlClient` is equally applicable to the `OleDb` and `Odbc` data providers. Where you prefix objects with `Sql`, you can, unless noted, replace these with an `OleDb` or `Odbc` version of the same object. I'll point out when the code required is slightly different depending on which database you're using.

■ **Note** In the code download for each of the chapters (available from the Downloads section of the Apress Web site at `http://www.apress.com`), you'll find `odbc` and `oledb` folders that contain the corresponding code for MySQL 5.0 and Microsoft Access.

Creating a Command Object

Creating a Command object is straightforward. In the code that you've already used in this chapter, you saw one way of doing this: by passing the query and connection to the Command object constructor.

The `SqlCommand` object has four constructors:

- `SqlCommand()`: This constructor creates a Command object that has nothing configured, and you must, at a minimum, specify a connection to use and the query you want to execute. You can specify these by using the `Connection` and `CommandText` properties.
- `SqlCommand(string)`: This allows you to specify the query you want to execute, although you'll still need to provide a connection, using the `Connection` property.
- `SqlCommand(string, SqlConnection)`: This specifies both the query you want to execute and the connection you want to use.
- `SqlCommand(string, SqlConnection, SqlTransaction)`: This allows you to specify, along with the connection and query, the transaction in which you want to participate. You'll look at transactions in more detail in Chapter 12.

You can use whichever version of the constructor you prefer, or as Microsoft likes to say, "You have a lifestyle choice."

In the previous examples, you've used the two-parameter version, like so:

```
SqlCommand myCommand = new SqlCommand(strCommandText, myConnection);
```

This is equivalent to the following:

```
SqlCommand myCommand = new SqlCommand(strCommandText);
myCommand.Connection = myConnection;
```

This is also equivalent to the following:

```
SqlCommand myCommand = new SqlCommand();
myCommand.CommandText = strCommandText;
myCommand.Connection = myConnection;
```

There is also a fifth way of creating a `SqlCommand` object: get the `SqlConnection` object do it for you. Using the `CreateCommand()` method of the `SqlConnection` object creates a `SqlCommand` object with its `Connection` property already set to the correct connection, like so:

```
SqlCommand myCommand = myConnection.CreateCommand();
myCommand.CommandText = strCommandText;
```

All the different methods available for creating the `SqlCommand` object may seem confusing at first. Just pick one that you're comfortable with and stick to it. You won't have any problems as long as you remember to set all the necessary properties before you open the connection and attempt to execute the query against the database.

When you're executing queries directly against the database, the two-parameter version is the one that requires the least number of lines of code, so that's the form you'll continue to use for the rest of the chapter.

Returning the Results

Once the SqlCommand object has been created correctly, you can use the ExecuteReader() method to return the results that you want to display as a SqlDataReader object. You can use this object directly as a data source for the GridView, so you pass it directly into the DataSource property:

```
// show the data
GridView1.DataSource = myCommand.ExecuteReader();
GridView1.DataBind();
```

We'll take a much more detailed look at the SqlDataReader object in Chapter 5. For now, it's enough to know that it returns a read-only, forward-only view of the results of the query.

Once the DataSource for the GridView is set, you call the DataBind() method to actually populate the GridView with the results. Without this call, no results will be shown, as the automatic data binding that you saw when using the SqlDataSource doesn't apply when you set the DataSource manually. We'll look at data binding in great detail in Chapters 6 and 7. For now, you can just rely on the fact that it works.

Filtering the Results

So far in this chapter, you've looked at the basics of connecting to a data source, executing a query, and returning the results to the page. However, you've hard-coded the query that you want to execute, so it's not very dynamic. You can show the Players for all of the Manufacturers, but you don't have any way to filter those results to show only the Players for a single Manufacturer.

You can filter the results of a query by using a WHERE clause to constrain the records that are returned. You can do this in two ways:

- By modifying the query that you're executing at runtime and specifying the variables within the WHERE clause directly
- By placing parameters within the WHERE clause of the query at design time and changing the values of these parameters at runtime

You'll look at each of these methods in turn.

Try It Out: Modifying the Query

In this example, you'll build on the previous example and allow the user to select the Manufacturer of interest. Once a selection has been made, only the Players for that Manufacturer will be returned. Follow these steps:

1. If you've closed Select.aspx from the root of the Chapter04 Web site, reopen it.
2. Switch to the Design view of the page and add a DropDownList to the top of the page, above the GridView that is already there.
3. From the DropDownList Tasks menu, check the Enable AutoPostPack option, as shown in Figure 4-4.

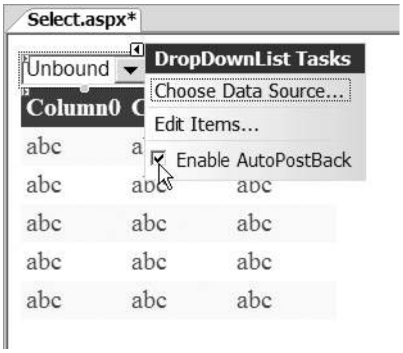


Figure 4-4. *Enabling the DropDownList to post back automatically*

4. Double-click the `DropDownList` to add the `SelectedIndexChanged` event. Add the following code (this is pretty much what is in the `Page_Load` event so you could copy it and modify the bits that are changed to avoid any extra typing):

```
protected void DropDownList1_SelectedIndexChanged(object sender, EventArgs e)
{
  // create the connection
  string strConnectionString = ConfigurationManager.
    ConnectionStrings["SqlConnectionString"].ConnectionString;
  SqlConnection myConnection = new SqlConnection(strConnectionString);

  // build the basic query
  string strCommandText = "SELECT Player.PlayerName, ➥
    Manufacturer.ManufacturerName FROM Player INNER JOIN ➥
    Manufacturer ON Player.PlayerManufacturerID = ➥
    Manufacturer.ManufacturerID";

  // add the filter
  string filterValue = DropDownList1.SelectedValue;
  if (filterValue != "0")
  {
    strCommandText += " WHERE Player.PlayerManufacturerID = " + filterValue;
  }

  // add the ordering
  strCommandText += " ORDER BY Player.PlayerName";

  // create the command
  SqlCommand myCommand = new SqlCommand(strCommandText, myConnection);

  // open the database connection
  myConnection.Open();
```

```
    // show the data
    GridView1.DataSource = myCommand.ExecuteReader();
    GridView1.DataBind();

    // close the database connection
    myConnection.Close();
  }
```

5. Replace the code within the `Page_Load` event with the following (again, you could just amend what is already there to avoid any unnecessary typing):

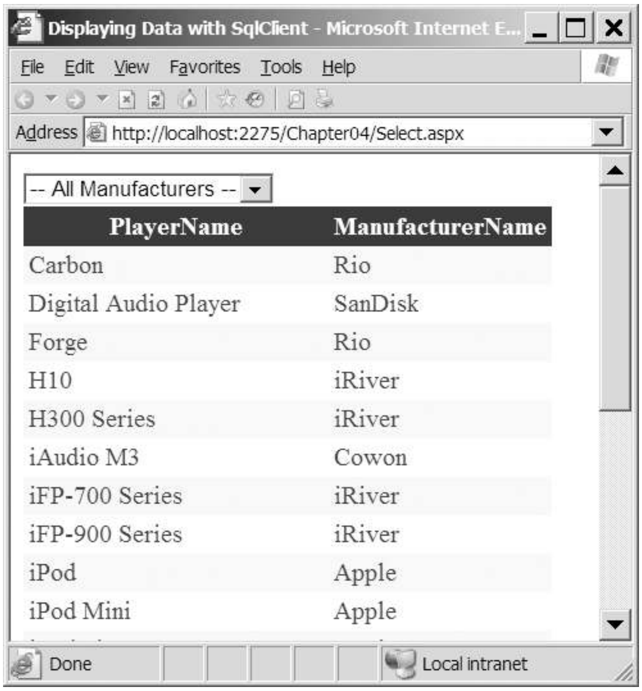
```
protected void Page_Load(object sender, EventArgs e)
{
  if (Page.IsPostBack == false)
  {
    // create the connection
    string strConnectionString = ConfigurationManager.
      ConnectionStrings["SqlConnectionString"].ConnectionString;
    SqlConnection myConnection = new SqlConnection(strConnectionString);

    // create the command
    string strCommandText = "SELECT ManufacturerID, ManufacturerName ➥
      FROM Manufacturer ORDER BY ManufacturerName";
    SqlCommand myCommand = new SqlCommand(strCommandText, myConnection);

    // open the database connection
    myConnection.Open();

    // show the data
    DropDownList1.DataSource = myCommand.ExecuteReader();
    DropDownList1.DataTextField = "ManufacturerName";
    DropDownList1.DataValueField = "ManufacturerID";
    DropDownList1.DataBind();

    // close the database connection
    myConnection.Close();

    // force the first data bind
    DropDownList1_SelectedIndexChanged(null,null);
  }
}
```

6. Switch back to the Design view of the page. Select the `DropDownList` and from the Properties window, add a `DataBound` event by double-clicking the DataBound entry.

7. Add the following code to the `DataBound` event:

```
protected void DropDownList1_DataBound(object sender, EventArgs e)
{
  DropDownList1.Items.Insert(0, new ListItem("-- All Manufacturers --", "0"));
}
```

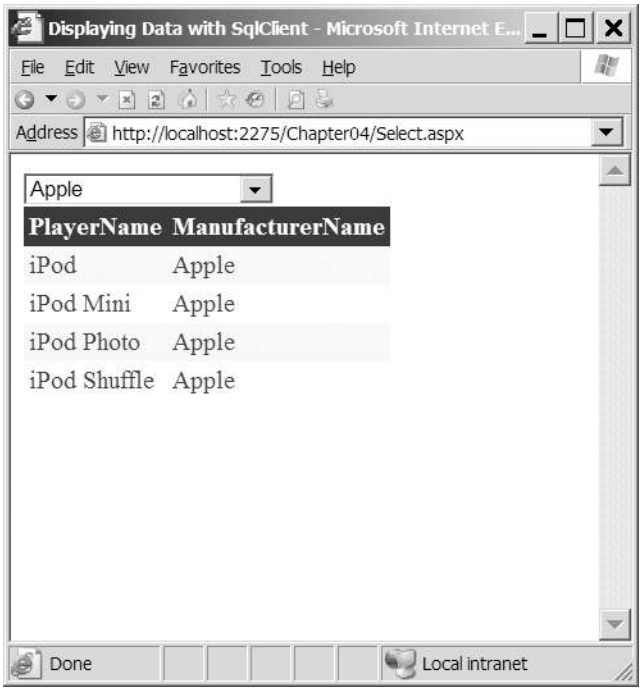
8. Execute the page. On the first load of the page, you'll see that the `GridView` displays a list of all of the Players for all the Manufacturers, as shown in Figure 4-5.


Displaying Data with SqlClient - Microsoft Internet E...
File Edit View Favorites Tools Help
Address http://localhost:2275/Chapter04/Select.aspx
-- All Manufacturers --
Done
Local intranet

PlayerName	ManufacturerName
Carbon	Rio
Digital Audio Player	SanDisk
Forge	Rio
H10	iRiver
H300 Series	iRiver
iAudio M3	Cowon
iFP-700 Series	iRiver
iFP-900 Series	iRiver
iPod	Apple
iPod Mini	Apple

Figure 4-5. *You can show Players for all the Manufacturers.*

9. If you expand the drop-down list, you'll see that it contains all of the Manufacturers in the database. Select Apple from the drop-down list. This will post the page back to the server and execute the `DropDownList1_SelectedIndexChanged` event handler, populating the `GridView` with the Players manufactured by Apple, as shown in Figure 4-6.


Displaying Data with SqlClient - Microsoft Internet E...
File Edit View Favorites Tools Help
Address http://localhost:2275/Chapter04/Select.aspx
Apple
Done
Local intranet

PlayerName	ManufacturerName
iPod	Apple
iPod Mini	Apple
iPod Photo	Apple
iPod Shuffle	Apple

Figure 4-6. *You can filter the Players by Manufacturer.*

10. Select any of the other Manufacturers to see that the list of Players is indeed modified to display only the correct list. Select -- All Manufacturers -- to see that the page shows all the Players in the database when that option is selected.

How It Works

Here, you've built the same page as you did in the examples in Chapter 3, but using code to query the database instead of the `SqlDataSource`. The code that you've built is executed in three different event handlers: `Page_Load`, `DataBound`, and `SelectedIndexChanged`. We'll look at each of these in turn.

The Page_Load Event

The `Page_Load` event handler executes every time the page is loaded, and you want to populate the list of Manufacturers when this occurs. However, you don't want to repopulate the Manufacturers every time the page is loaded; the drop-down list remembers what data it contains automatically, so there is no need to reconnect to the database again to retrieve the list of Manufacturers. You check that the page is not a postback and populate the list of Manufacturers only if this is the first load of the page.

The data-access code is remarkably similar to all of the other code that you've seen so far. You create a connection to the database, specify the query you want to execute, and then set the `DataSource` for `DropDownList1` to the query results by using the `ExecuteReader()` method.

The query that you're executing to retrieve the results is a simple one that you've already looked at in Chapter 3:

```
SELECT ManufacturerID, ManufacturerName
FROM Manufacturer
ORDER BY ManufacturerName
```

You retrieve the ManufacturerID and ManufacturerName from the Manufacturer table, ordering the results by ManufacturerName. Looking at the query should give you a clue as to what the two new lines of code introduced here are for:

```
DropDownList1.DataTextField = "ManufacturerName";
DropDownList1.DataValueField = "ManufacturerID";
```

We'll look at data binding in a lot more detail in Chapters 6 and 7, but you need to understand what the `DataTextField` and `DataValueField` properties do. All list Web controls, of which the `DropDownList` is one, show a text description for each row returned and have a "hidden" value that is returned whenever you want to know what the selected value is. The `DataTextField` and `DataValueField` properties are used to set what is displayed and what is returned.

You've set the `DataTextField` value to be ManufacturerName, as this is what will make sense to the user; showing a numeral such as 2 as a ManufacturerName is meaningless. Similarly, the numeral that you don't want to show the user makes sense as the value of the Web control. The ManufacturerID is a value you can easily search for within the database, as it's the primary key for the Manufacturer table and is used as a foreign key within the Player table.

Once you've set these values, you can call the `DataBind()` method on the drop-down list to populate the list.

Once the data binding is complete, you can then close the connection to the database and call the `DropDownList1_SelectedIndexChanged` event handler to force the `GridView` to be populated.

The DataBound Event

As you'll recall from the previous chapter, when the data binding has completed, the `DataBound` event is fired for the Web control. This occurs during the execution of the `DataBind()` method in `Page_Load`; execution will move from `Page_Load` to the `DataBound` event handler and then back once the event handler is complete.

As you're retrieving the data from the database, you won't get an -- All Manufacturers -- entry automatically. Therefore, you need to manually add one at the start of the list. The Manufacturers in the database have an ID value of 1 and upwards, so you add the -- All Manufacturers -- entry with a value of 0, which won't match a real Manufacturer in the database.

The SelectedIndexChanged Event

Unlike the data binding you saw in Chapter 3, Web controls are not automatically data-bound when using code to query the database. You must manually tell the Web controls to bind to their data source using the `DataBind()` method. As you'll soon see, all of the code to populate the `GridView` is contained within the `DropDownList1_SelectedIndexChanged` event handler, so you call it manually, passing in `null` as both parameters. Although you're calling the event handler, this doesn't cause the page to postback to execute the handler. An event handler is just a normal method that is called by ASP.NET to respond to a particular event. You're free to call this from your own code if you desire.

The `DropDownList1_SelectedIndexChanged` event handler is used to populate the `GridView` with the list of Players based on the Manufacturer the user selected and is the part of the code that is of most interest now.

Unlike in the previous examples, here the query is built dynamically based on the user's selection. The first part of the query is as follows:

```
SELECT Player.PlayerName, Manufacturer.ManufacturerName
FROM Player INNER JOIN Manufacturer
  ON Player.PlayerManufacturerID = Manufacturer.ManufacturerID
```

This is a perfectly valid query in its own right, and it will return all the Players in the database, regardless of the Manufacturer. In fact, it's the same query as you used in the previous example with the `ORDER BY` clause removed. If you recall the `SELECT` syntax from Chapter 3, you'll remember that the `WHERE` clause to constrain the query must come before the `ORDER BY` clause—you must filter the query before you can order it—and you need to remove the `ORDER BY` clause so that you can add the `WHERE` clause.

To constrain the query, you first retrieve the value of the user's selection from the drop-down list using the `SelectedValue` property. You store this in a local variable, `filterValue`, because you'll use it in several places, like so:

```
string filterValue = DropDownList1.SelectedValue;
```

You've stored this value because you can't simply use it to constrain the query—if you've requested all the Manufacturers, you don't want to add a constraint. You'll recall that you added an -- All Manufacturers -- entry to the drop-down list, and if you've selected this, you want to return all the Players in the database as opposed to a list of Players for a particular Manufacturer. It's when a Manufacturer has been selected that you want to modify the query. You can check whether a specific Manufacturer has been selected by checking for a value that's nonzero. If it's nonzero, you want to add a `WHERE` clause:

```
strCommandText += " WHERE Player.PlayerManufacturerID = " + filterValue;
```

The effect of this `WHERE` clause is to tell the database you want only the records that have a PlayerManufacturerID that's equal to the value you've specified.

■ **Note** Although you specify that you want to constrain the query on the Player.PlayerManufacturerID column, nothing is stopping you from using the Manufacturer.ManufacturerID column instead. As they're the columns that make the join, you can use either of them and still return the same results.

Regardless of whether a `WHERE` clause has been added to the query, you add an `ORDER BY` clause so that the Players are ordered alphabetically, like so:

```
ORDER BY Player.PlayerName
```

Now that you know how the query is built, you can look at what's actually executed against the database. If you've selected the -- All Manufacturers -- option, you're not adding a `WHERE` clause, and the query that's executed is the query you had in the previous example:

```
SELECT Player.PlayerName, Manufacturer.ManufacturerName
FROM Player INNER JOIN Manufacturer
  ON Player.PlayerManufacturerID = Manufacturer.ManufacturerID
ORDER BY Player.PlayerName
```

This query returns all the Players in the database because you're not constraining the query. However, if you select the Apple option, you want to add a `WHERE` clause, and the query you execute is as follows:

```
SELECT Player.PlayerName, Manufacturer.ManufacturerName
FROM Player INNER JOIN Manufacturer
  ON Player.PlayerManufacturerID = Manufacturer.ManufacturerID
WHERE Player.PlayerManufacturerID = 1
ORDER BY Player.PlayerName
```

You constrain the query to return only the results that have a PlayerManufacturerID equal to 1, which is the ManufacturerID value for Apple.

Open to SQL Injection Attacks

Before we move on, it's worth taking a quick detour into why you shouldn't construct queries at runtime using string concatenation. When constructing queries using string concatenation, it is far too easy to leave the database wide open to attack—SQL injection attacks in particular.

Consider the case where you want to execute the following query to return the user's profile from the database:

```
SELECT * FROM tblUser WHERE UserName = '<<USERNAME>>';
```

Now suppose the user enters her username on a page. The majority of users will enter the correct username, and the query will run as expected. But what if the user entered the following:

```
' OR '1' = '1
```

The string concatenation would merge the entered username with the query that you've defined and actually execute the following against the database:

```
SELECT * FROM tblUser WHERE UserName = '' OR '1' = '1';
```

Oh! Do you really want to show all of the users in tblUser? Even worse, what if the user entered the following:

```
'; DELETE FROM tblUser; --
```

Granted, the user needs a little understanding of your table structure, but you've potentially lost all of the data in tblUser. The database would actually be executing two queries:

```
SELECT * FROM tblUser WHERE UserName = '';
DELETE FROM tblUser;
```

To prevent SQL injection attacks, you might perform checking on the string that the user has entered to make sure that it doesn't do anything it shouldn't. If you go down this route, you'll be fighting a losing battle, but thankfully there is a solution. You can use parameters to fixed queries, rather than constructing the query using string concatenation at runtime.

Try It Out: Using Parameters in Queries

Rather than constructing a SQL query at runtime and passing it to the database to be executed, you can instead use parameters to modify a fixed SQL query. This example will use that methodology to replicate the functionality you saw in the previous example. Follow these steps:

1. Open `Select.aspx` from the root of the Chapter04 Web site and switch to the Source view of the page.
2. In the `DropDownList1_SelectedIndexChanged` event, replace the code that creates the SQL query to be executed with the following:

```
string strCommandText = "SELECT Player.PlayerName, ➥
  Manufacturer.ManufacturerName FROM Player INNER JOIN Manufacturer ➥
  ON Player.PlayerManufacturerID = Manufacturer.ManufacturerID ➥
  WHERE @ManufacturerID = 0 OR Player.PlayerManufacturerID = ➥
  @ManufacturerID ORDER BY Player.PlayerName";
```

3. Remove the existing lines of code that add the filter and the ordering to `strCommandText`.
4. Add the following code before the call to open the database:

```
// add the parameter
SqlParameter myParameter = new SqlParameter();
myParameter.ParameterName = "@ManufacturerID";
myParameter.SqlDbType = SqlDbType.Int;
myParameter.Value = DropDownList1.SelectedValue;
myCommand.Parameters.Add(myParameter);
```

5. Execute the page. You'll see that the page performs exactly as you would expect, with the list of Players filtered correctly according to the Manufacturer that is selected.

How It Works

All that you've changed is the query used to select the Players from the database. Rather than having a query constructed at runtime, you have a complete query that you modify using a parameter.

As you saw in Chapter 3, parameters are the means whereby you can pass information into a query at runtime without making any changes to the query itself.

The query that you're using is the same query that you had for the filtering example in Chapter 3. That query used a parameter called `@ManufacturerID` to allow the Manufacturer to be specified by the user's selection:

```
SELECT Player.PlayerName, Manufacturer.ManufacturerName
FROM Player INNER JOIN Manufacturer
  ON Player.PlayerManufacturerID = Manufacturer.ManufacturerID
WHERE @ManufacturerID = 0 OR
  Player.PlayerManufacturerID = @ManufacturerID
ORDER BY Player.PlayerName
```

You therefore need to provide a parameter to the query based on the user's selection. When you used a SqlDataSource, you added a ControlParameter to the SelectParameters collection, and passing the selected value was handled automatically; the query executed with the correct value for the selected Manufacturer. However, you need to provide the parameter and its value, and this is what the extra code that you've added to the example shows.

To add the value of @ManufacturerID to the query, you use a SqlParameter object. You must first create the necessary SqlParameter object:

```
SqlParameter myParameter = new SqlParameter();
```

Before you can use this parameter, you must set various properties on it. At a bare minimum, you must set the name, the data type, and the value of the parameter, like so:

```
myParameter.ParameterName = "@ManufacturerID";
myParameter.SqlDbType = SqlDbType.Int;
myParameter.Value = DropDownList1.SelectedValue;
```

The ParameterName value must match the name in the query, and Value is simply the value that you want to assign to the parameter—in this case, the SelectedValue of the drop-down list.

You specify the data type using the SqlDbType property and passing in a member of the SqlDbType enumeration. There are values in the enumeration corresponding to all the data types that are available in SQL, and you can use any of them in parameters. In this case, you want a simple integer, so you specify SqlDbType.Int. You'll find a list of the values available in the SqlDbType enumeration in Appendix B, along with the equivalent .NET and SQL types.

Once you've created the parameter, you must add it to the SqlCommand object before the query executes:

```
myCommand.Parameters.Add(myParameter);
```

Other Methods for Adding Parameters

Although you've created a SqlParameter object and added this to the Parameters collection, this isn't the only way you can add parameters to the SqlCommand object. The Parameters property returns a SqlParametersCollection that has several different methods for adding parameters:

- Add(SqlParameter): Adds a SqlParameter that you've already created to the collection.
- Add(string, SqlDbType): Adds a new SqlParameter to the collection with the specified name and type. The parameter will not have a value.

- `Add(string, SqlDbType, int)`: Adds a new `SqlParameter` to the collection with the specified name, type, and size. This is useful when you're using a text type, as you can set the length of the string. The parameter will not have a value.
- `AddWithValue(string, Object)`: Adds a new `SqlParameter` with the specified name and value. The type will be inferred from the object that is passed to the method.

Another way of creating a `SqlParameter` object is to have the `SqlCommand` object do it for you. Using the `CreateParameter()` method of the `SqlCommand` object creates a new `SqlParameter` object that has already been added to the `Parameters` collection, like so:

```
SqlParameter myParameter = myCommand.CreateParameter();
myParameter.Name = "@ManufactuerID";
myParameter.SqlDbType = SqlDbType.Int;
myParameter.Value = DropDownList1.SelectedValue;
```

You can use any of these methods to add parameters. I chose to use the `Add()` overload that requires a `SqlParameter` object because it introduces the concept of using names, types, and values for parameters.

Protected from SQL Injection Attacks

As I explained after the example of modifying the query, creating queries using string concatenation isn't the correct way to do things. One big problem is that it leaves your database wide open for SQL injection attacks. As I said, parameters are the solution.

You're executing this query to return the user's profile from the database:

```
SELECT * FROM tblUser WHERE UserName = '<<USERNAME>>';
```

Even if the user enters one of the strings that would break the previous example, such as:

```
' OR '1'='1
```

this won't actually cause a problem, other than that the user won't be found in the database.

The value entered as the parameter is treated as the whole parameter, so the `WHERE` clause is actually checking whether the UserName value in the database is equal to the entire string that the user entered: `'  OR '1' = '`.

I don't think any user will have picked that as a username (although with users, you never know!), so the query will not return any results—exactly what you would hope to happen.

Parameters and Queries

Using parameters is one of the few instances where the query to be executed and the corresponding code changes depend on the Command object you're using. Each of the Command objects uses parameters in slightly different ways and requires different queries and changes to the way parameters are added.

Recall from the previous example that the parameters were added with the `ParameterName` of the `SqlParameter` matching the name of the parameter in the query passed to the `SqlCommand`. The query had a parameter called `@ManufacturerID`, and a `SqlParameter` with its `ParameterName` set to `@ManufacturerID` was added to the `SqlCommand`. With named parameters, the `SqlConnection` object can determine which `SqlParameter` is required, and even though the parameter was used twice in the query, only one `SqlParameter` is required.

Only when using the SqlClient data provider to connect to SQL Server 2005 can you use named parameters. Neither the Odbc data provider when connecting to MySQL 5.0 nor the OleDb data provider when connecting to any database (such as Microsoft Access) support named parameters. Both the OdbcCommand and OleDbCommand objects rely on the order that the Parameter objects are added to the Command object to determine how they're inserted into the query.

As named parameters are no longer used, OdbcCommand and OleDbCommand require a slightly different query than SqlCommand. You can no longer use the @ syntax to refer to a parameter; you use a question mark (?) instead, like so:

```
SELECT Player.PlayerName, Manufacturer.ManufacturerName
FROM Player INNER JOIN Manufacturer
  ON Player.PlayerManufacturerID = Manufacturer.ManufacturerID
WHERE ? = 0 OR Player.PlayerManufacturerID = ?
```

As you have two parameters (indicated by the two question marks), you must have two Parameter objects added to the Command object, even if, as in this case, the two parameters take the same value. As these aren't named parameters, you can add them by specifying only their type and value; you don't need to give the parameter a name.

For the OdbcCommand object, you need to create two OdbcParameter objects and add them to the OdbcCommand object, like so:

```
// add the first parameter
OdbcParameter myParameter1 = new OdbcParameter();
myParameter1.OdbcType = OdbcType.Int;
myParameter1.Value = DropDownList1.SelectedValue;
myCommand.Parameters.Add(myParameter1);

// add the second parameter
OdbcParameter myParameter2 = new OdbcParameter();
myParameter2.OdbcType = OdbcType.Int;
myParameter2.Value = DropDownList1.SelectedValue;
myCommand.Parameters.Add(myParameter2);
```

For the OleDbCommand object, the process is the same. You create the following two OleDbParameter objects and add these to the OleDbCommand object:

```
// add the first parameter
OleDbParameter myParameter1 = new OleDbParameter();
myParameter1.OleDbType = OleDbType.Integer;
myParameter1.Value = DropDownList1.SelectedValue;
myCommand.Parameters.Add(myParameter1);

// add the second parameter
OleDbParameter myParameter2 = new OleDbParameter();
myParameter2.OleDbType = OleDbType.Integer;
myParameter2.Value = DropDownList1.SelectedValue;
myCommand.Parameters.Add(myParameter2);
```

■ **Note** Although you don't need to give the parameter a name, it's a good idea to do so. Both OdbcCommand and OleDbCommand will ignore the name, but the name can still be used to access the individual parameters in the Parameters collection. If you ever need to change the value of the parameter, it's much easier to access the parameter using a name rather than the parameter's position in the Parameters collection.

Command Object Methods and Properties

Although you've looked at querying data sources using three different Command objects, there are a few properties and methods that, while not essential for querying the data sources, allow you greater control over the query. Here, we'll look at some of the most useful properties and methods. If you would like more details about the others that are available, you can find more information on MSDN at the following locations:

- SqlCommand:
  http://msdn.microsoft.com/en-us/library/system.data.sqlclient.sqlcommand.aspx
- OdbcCommand:
  http://msdn.microsoft.com/en-us/library/system.data.odbc.odbccommand.aspx
- OleDbCommand:
  http://msdn.microsoft.com/en-us/library/system.data.oledb.oledbcommand.aspx

The CommandText Property

In the examples you've looked at so far, you've always created the Command object by passing the query to execute directly to the constructor. The Command object also exposes the CommandText property, which you can use instead to specify the query after you've created the Command object. So, you can create a SqlCommand object and specify the query to execute as follows:

```
SqlCommand myCommand = new SqlCommand();
myCommand.CommandText = "SELECT ManufacturerID, ManufacturerName ➥
  FROM Manufacturer ORDER BY ManufacturerName";
```

The CommandType Property

So far, we've looked at passing SQL queries directly to the database. However, this isn't the only way that you can query the database. For instance, as you'll see in Chapter 10, you can also use stored procedures.

A SQL query is the default query type, and this is represented by a value of Text from the CommandType enumeration. Table 4-1 shows all the different values for the CommandType enumeration.

You've already implicitly used the CommandType.Text value. as you've been executing queries against the database without specifying a value for the CommandType property. In this case, the default value was used automatically.

Table 4-1. *Values of the CommandType Enumeration*

Value	Description
`StoredProcedure`	Indicates that the value passed as the `CommandText` is the name of a stored procedure to execute (discussed in Chapter 10).
`TableDirect`	Specifies that the `CommandText` is the name of a table within the data source and all the data within the table should be returned. This value is not supported by either the `SqlClient` or `Odbc` data providers.
`Text`	Indicates that the `CommandText` property contains a SQL query to execute. This is the default value.

The Execute Methods

The `Execute` methods of the Command object, listed in Table 4-2, allow you to execute queries against the database. We've already looked at one of these: the `ExecuteReader()` method.

Table 4-2. *The Execute Methods*

Value	Description
`ExecuteNonQuery()`	Executes the specified query against the database and doesn't return any results from the query, even if the query had results to return. Instead, the query returns the number of rows affected by the query. Use this method when executing `INSERT`, `UPDATE`, and `DELETE` queries (see Chapter 8).
`ExecuteReader()`	Returns a read-only, forward-only view of the query results.
`ExecuteScalar()`	Returns a single value, rather than one or more rows of data. You can use the `ExecuteScalar()` method to return information from the database without the overhead of using a DataReader object.

Next, you'll see how to use the `ExecuteScalar()` method to return data from a database.

Scalar Commands

Although you've looked at the most common method for returning data from a database (the `ExecuteReader()` method), sometimes you can avoid the overhead that goes with returning the results as a DataReader object. If the query you're executing returns only a single value from the database, you can use the `ExecuteScalar()` method. Yes, you could perform the same task using the `ExecuteReader()` method and manipulating the DataReader object that's returned. However, that requires a lot more code and is slower than using the `ExecuteScalar()` method.

A common reason to return only one value from a query is when you're using scalar functions to query a table within the database.

Scalar Functions

Scalar functions, or *aggregate functions* as Microsoft likes to call them, are mathematical functions defined within SQL that return a single value. Table 4-3 describes some of the more common scalar functions.

Table 4-3. *Common Scalar Functions*

Scalar Function	Description
`AVG(column)`	Returns the average value of the specified column
`COUNT(DISTINCT column)`	Counts the number of distinct values in the specified column
`COUNT(*)`	Gives the number of rows in the specified table
`MAX(column)`	Returns the maximum value in the specified column
`MIN(column)`	Returns the minimum value in the specified column
`SUM(column)`	Returns the total of all the values in the specified column

You can use scalar functions in several places in SQL, but by far, the most common usage is returning them as columns from `SELECT` queries.

■ **Note** You can also use scalar functions as constraints in `SELECT` queries, but only if you've grouped the columns in the query using the `GROUP BY` clause. In this case, you would use the `HAVING` clause in place of the `WHERE` clause to apply the constraint. For more information about using the `GROUP BY` and `HAVING` clauses, see *SQL Queries for Mere Mortals* by Michael J. Hernandez and John L. Viescas (0-20143-336-2; Addison-Wesley, 2000).

Try It Out: Using the ExecuteScalar() Method

In this example, you'll build on one of the previous examples. You'll use the `COUNT(*)` scalar function and return the number of records that your query has matched. Follow these steps:

1. Open the Chapter04 Web site in Visual Web Developer.
2. Copy `Select.aspx`, and rename the copied version to `Scalar.aspx`.
3. On the Design view of the page, on a new line after the `DropDownList`, enter **Players for this Manufacturer:**. Then add a `Label` from the Toolbox. Change the ID of the label to `lblCount`, and remove its default text. You should have a page that looks similar to the one shown in Figure 4-7.

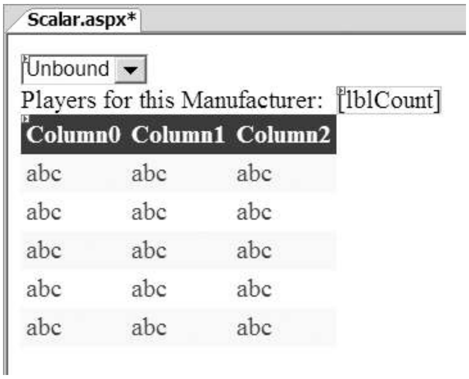
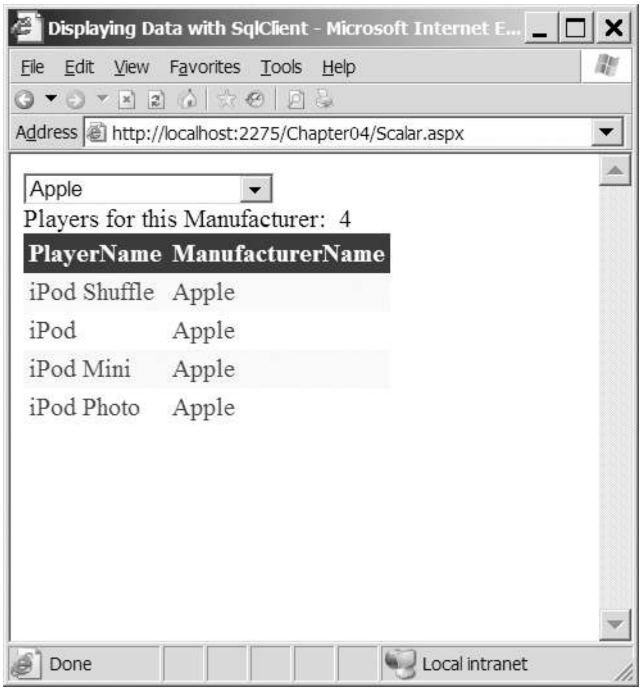


Figure 4-7. *Adding a count of the number of Players*

4. Switch to the Code view of the page. Add the following code immediately before opening the database connection in the `DropDownList1_SelectedIndexChanged` event handler (the additional code is shown in bold):

```
// create the count query
string strCommandTextCount = "SELECT COUNT(*) FROM Player WHERE ➥
  @ManufacturerID = 0 OR Player.PlayerManufacturerID = @ManufacturerID";
SqlCommand myCommandCount =
  new SqlCommand(strCommandTextCount, myConnection);
SqlParameter myParameterCount = new SqlParameter();
myParameterCount.ParameterName = "@ManufacturerID";
myParameterCount.SqlDbType = SqlDbType.Int;
myParameterCount.Value = DropDownList1.SelectedValue;
myCommandCount.Parameters.Add(myParameterCount);

// open the database connection
myConnection.Open();

// count the players for the manufacturer
lblCount.Text = Convert.ToString(myCommandCount.ExecuteScalar());
```

5. Execute the page. As well as seeing the selected Manufacturer's Players, the number of Players available will be displayed. Initially, the count will be 20, as you're not filtering the results. Selecting a Manufacturer, such as Apple, will filter the results, and the count of the Players will be adjusted accordingly, as shown in Figure 4-8.


Displaying Data with SqlClient - Microsoft Internet E...
File Edit View Favorites Tools Help
Address http://localhost:2275/Chapter04/Scalar.aspx
Apple
Done
Local intranet

Players for this Manufacturer: 4

PlayerName	ManufacturerName
iPod Shuffle	Apple
iPod	Apple
iPod Mini	Apple
iPod Photo	Apple

Figure 4-8. *The count of the number of Players is returned.*

How It Works

You've used an earlier example as the basis for this example, and you've simply added a label to the page that you populate with the count of the number of Players for the selected Manufacturer. The count is returned by using the following `COUNT(*)` scalar function and returning this as the result from a `SELECT` query:

```
SELECT COUNT(*)
FROM Player
WHERE @ManufacturerID = 0 OR Player.PlayerManufacturerID = @ManufacturerID
```

You're filtering the results using a parameter, and by specifying `COUNT(*)` as the only column, the query will return a single row containing a single column. This is how you use the `ExecuteScalar()` method.

The `ExecuteScalar()` method returns an object representing the value that has been returned from the query. In this case, you're returning an integer, and you need to convert this to a string before you can assign it to the `Text` property of the label, like so:

```
lblCount.Text = Convert.ToString(myCommandCount.ExecuteScalar());
```

Error Handling

As any programmer will tell you, you're never going to write code that doesn't fall over at some point—whether it's caused by an error within the code or something outside the scope of the code (such as someone unplugging the database server).

Unless you have some way of handling any errors that occur, any problems you encounter can have the side effect of leaving connections to the database open. As you've already learned, database connections are a finite resource, so leaving connections open is definitely not a good idea.

If you're using `SqlDataSource` objects to connect to the database, as in Chapter 3, you don't need to worry about error handling to close the database connections. The `SqlDataSource` handles all the connections to the database internally, so you can be sure that any open database connections are handled before any error is thrown. However, when you're interacting with the database in code, you do need to catch and handle errors.

Try It Out: Catching and Handling Errors

You should already be familiar with the `try..catch..finally` syntax for handling errors, so you'll now see how to use this syntax to handle any errors that may occur within a page. If an error occurs, you'll write the error to a log file and close the open database connection.

1. Open `Select.aspx` from the root of the Chapter04 Web site and switch to the Source view of the page.

2. Add the following `Import` statement to the top of the page after the existing `Import` statements:

```
<%@ Import Namespace="System.IO" %>
```

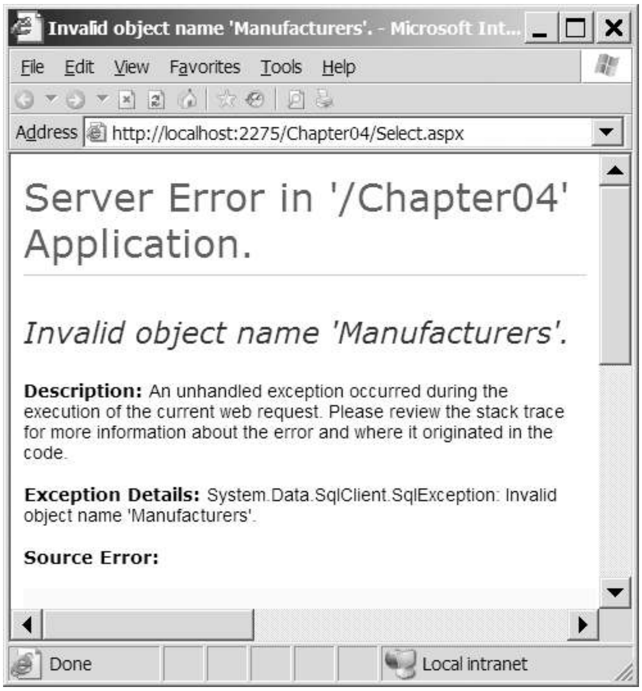
3. Change the `Page_Load` event as follows (the changed lines of code are shown in bold, and note that the name of the table is deliberately incorrect, with an *s* at the end):

```
protected void Page_Load(object sender, EventArgs e)
{
  if (Page.IsPostBack == false)
  {
    // create the connection
    SqlConnection myConnection = new SqlConnection();

    try
    {
      // configure the connection
      string strConnectionString = ConfigurationManager.
        ConnectionStrings["SqlConnectionString"].ConnectionString;
      myConnection.ConnectionString = strConnectionString;
```

```
        // create the command
        string strCommandText = "SELECT ManufacturerID, ManufacturerName ➥
          FROM Manufacturers ORDER BY ManufacturerName";
        SqlCommand myCommand = new SqlCommand(strCommandText, myConnection);

        // open the database connection
        myConnection.Open();

        // show the data
        DropDownList1.DataSource = myCommand.ExecuteReader();
        DropDownList1.DataTextField = "ManufacturerName";
        DropDownList1.DataValueField = "ManufacturerID";
        DropDownList1.DataBind();

        // force the first data bind
        DropDownList1_SelectedIndexChanged(null, null);
      }
      catch (Exception ex)
      {
        // write the error to file
        StreamWriter sw = File.AppendText(Server.MapPath("~/error.log"));
        sw.WriteLine(ex.Message);
        sw.Close();

        // now rethrow the error
        throw (ex);
      }
      finally
      {
        // close the database connection
        myConnection.Close();
      }
    }
  }
```

4. Execute the page. You should immediately be presented with the error shown in Figure 4-9.

5. In Windows Explorer, navigate to the `C:\BAND\Chapter04` folder, and you'll see that a file called `error.log` has been created. Open that file. You'll see the error has been logged, as shown in Figure 4-10.


Invalid object name 'Manufacturers'. - Microsoft Int...
File Edit View Favorites Tools Help
Address http://localhost:2275/Chapter04/Select.aspx
Done
Local intranet

Server Error in '/Chapter04' Application.

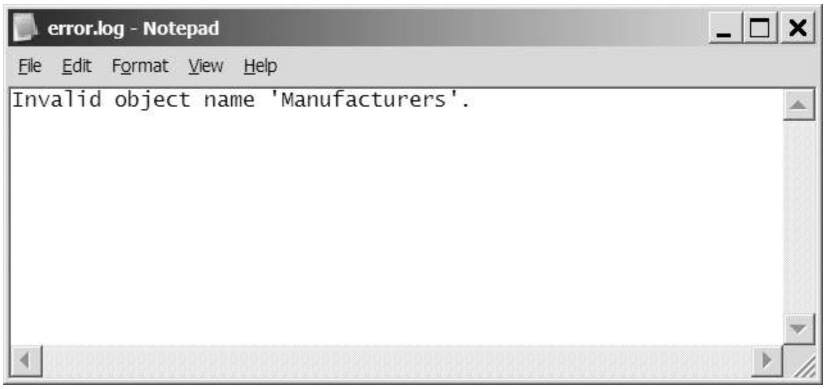
Invalid object name 'Manufacturers'.

Description: An unhandled exception occurred during the execution of the current web request. Please review the stack trace for more information about the error and where it originated in the code.

Exception Details: System.Data.SqlClient.SqlException: Invalid object name 'Manufacturers'.

Source Error:

Figure 4-9. *The error presented to the user*



```
Invalid object name 'Manufacturers'.
```

Figure 4-10. *The error has been logged in the error.log file.*

6. Switch back to the Source view of the page and fix the intentional error by changing the query to be executed within `Page_Load` as follows:

```
// create the command
string strCommandText = "SELECT ManufacturerID, ManufacturerName
  FROM Manufacturer ORDER BY ManufacturerName";
```

How It Works

You should already be familiar with error handling in .NET. Here, you've simply moved some of the code for accessing the database around to fit within the `try..catch..finally` syntax.

First, within the `Page_Load` event handler, before you get to any error-handling code, you need to create the Connection object. You must do this here, because the Connection object needs to be global to the entire event handler. If you created it in the `try` block, it wouldn't be available in the `catch` or the `finally` block. You create the Connection object without specifying the connection string, like so:

```
SqlConnection myConnection = new SqlConnection();
```

You then move into the `try` block. The code here is the same as you've seen in the previous example; the only difference is that instead of creating the Connection object with the correct connection string, you set the `ConnectionString` property of the existing Connection object, like so:

```
// configure the connection
string strConnectionString = ConfigurationManager.
  ConnectionStrings["SqlConnectionString"].ConnectionString;
myConnection.ConnectionString = strConnectionString;
```

The rest of the code is the same as you had previously (barring the intentional naming of the Manufacturer table incorrectly), except you've removed the code to close the database connection:

```
// create the command
string strCommandText = "SELECT ManufacturerID, ManufacturerName
  FROM Manufacturers ORDER BY ManufacturerName";
SqlCommand myCommand = new SqlCommand(strCommandText, myConnection);

// open the database connection
myConnection.Open();

// show the data
DropDownList1.DataSource = myCommand.ExecuteReader();
DropDownList1.DataTextField = "ManufacturerName";
DropDownList1.DataValueField = "ManufacturerID";
DropDownList1.DataBind();

// force the first data bind
DropDownList1_SelectedIndexChanged(null, null);
```

Whether or not you have an error, you always need to close the database connection. You've moved the call to `Close()` to the `finally` block.

If any of the code in the try block generates an error, then execution is automatically passed to the catch block, and it's in here that you log the error to the log file:

```
// write the error to file
StreamWriter sw = File.AppendText(Server.MapPath("~/error.log"));
sw.WriteLine(ex.Message);
sw.Close();
```

To write entries to the file, you create a StreamWriter using the System.IO.File.AppendText() static method. This method accepts a filename and opens the file for writing. If the file doesn't exist, it is created automatically.

You then use the WriteLine() method to write the error message to file, and then Close() the open StreamWriter.

■ **Note** When using IIS to host your Web site, you must ensure that the user running the ASP.NET process (ASPNET under IIS5 or NETWORK SERVICE under IIS6) has the required permissions to write to the folder where you want to store the log file. In this example, the page has been running under the account that you're logged on to the machine as, and you'll have write access to the C:\BAND\Chapter04 folder, as you created it in an earlier example.

You then reraise the error that you've handled, like so:

```
// now rethrow the error
throw(ex);
```

If you don't rethrow the error, ASP.NET will, since you've caught the error, assume that it has been handled and that any problems have been rectified. As you're only logging the error and not doing anything to fix it, you rethrow the error so that ASP.NET is aware that a problem occurred. If you don't rethrow the error, the user would be presented with a page that's equally as unhelpful as an ASP.NET error message, as shown in Figure 4-11.

Whether or not an error occurred, the finally block then executes. All you want to do here is close the connection to the database, like so:

```
// close the database connection
myConnection.Close();
```

Although it's possible to check the state of the connection using the State property and close the connection only if it's open, this isn't necessary. If the connection is already closed, then calling the Close() method won't have any unwanted side effects.

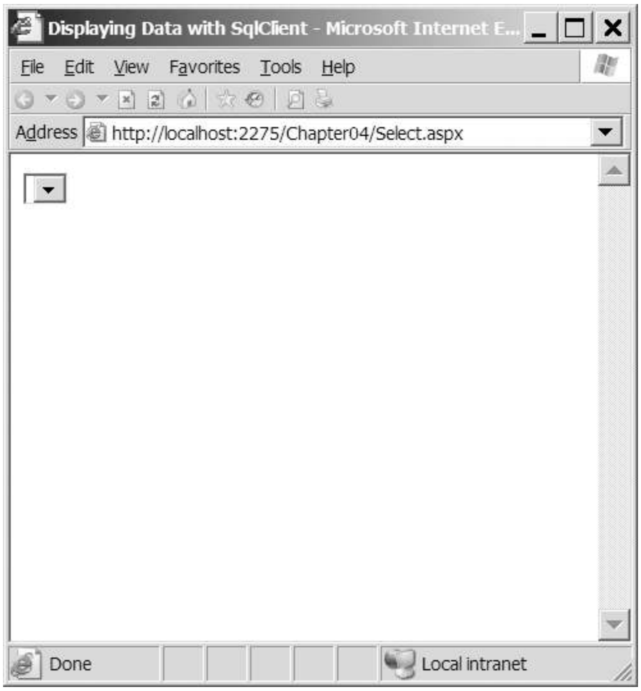

Displaying Data with SqlClient - Microsoft Internet E...
File Edit View Favorites Tools Help
Address http://localhost:2275/Chapter04/Select.aspx
Done
Local intranet

Figure 4-11. *You shouldn't hide errors from ASP.NET.*

Summary

As you saw in Chapter 1, before you can do anything with a data source, you must make a connection to it. You've spent some time in this chapter looking at connecting to several different data sources using the following Connection and Command objects:

- SQL Server 2005 using the `SqlConnection` and `SqlCommand` objects
- MySQL 5.0 using the `OdbcConnection` and `OdbcCommand` objects
- Microsoft Access using the Jet engine through the `OleDbConnection` and `OleDbCommand` objects

Although you've looked at all three different data sources and three different sets of objects, the beauty of the data provider architecture in ASP.NET is that the paradigm for all of the Connection and Command objects is exactly the same.

You also briefly looked at some of the other properties and methods of the Connection and Command objects.

Next, you learned about passing parameters into queries, and got your first look at how the different objects behave slightly differently. The `SqlCommand` object can handle named parameters, whereas the `OdbcCommand` and `OleDbCommand` objects, in our scenario, require parameters presented in the order in which they're to be used.

You then took a brief look at scalar functions and the `ExecuteScalar()` method as an alternative for returning information from the database when you want to return only one value.

At the end of the chapter, you looked at basic error handling and wrote an error handler that ensures you never leave any open connections.

The examples that you've seen so far have been simple pages that performed one task to give you a foundation in using databases and the different queries you can perform. In the next four chapters, you'll build on the techniques you've developed and learn how to build interactive pages. You'll also start to see the real power that's available to data-driven Web sites.