CHAPTER 8

Writing to the Database

So far, you've dealt with only read-only data—pulling some data from a database without altering it. In this chapter, you'll discover how to create pages that allow users to add, modify, and delete the contents of a database. At the core of these three operations are three SQL queries: `INSERT`, `UPDATE`, and `DELETE`.

First, we'll look at modifying the data in separate pages that neatly wrap up the three different types of operations using a Command object. Then you'll see how to use a `DataSet` to make changes to tables.

This chapter covers the following topics:

- How to use single-value and list Web controls to build a query sent directly to the database with a call to `ExecuteNonQuery()` or `ExecuteScalar()`
- How to validate data entered through Web controls to make sure no invalid changes are made
- How to use a `DataSet` to hold several different tables and propagate any changes back to the database in one call to `Update()` through the DataAdapter

Making Changes to a Database

Those three basic steps you first heard about back in Chapter 1—creating the connection, sending the query, and handling the results—still hold true for making changes to a database. However, you have a lot more things to consider, and the changes must play by the database's rules.

The main difference is in sending the query, where you'll need to use the appropriate query for what you want to do. The results of the query will generally be a scalar value indicating the number of rows in the database that have changed as the result of the query. It's your choice whether you use this result, but it does provide quite a good indication of whether the query that you've executed has worked correctly.

In this chapter, you'll learn how to use the following queries:

- The SQL `INSERT` query to add new rows to a table in a database
- The SQL `UPDATE` query to change rows already in a database
- The SQL `DELETE` query to remove rows from a database

Unlike the SELECT query, which just retrieves data, these three queries must obey the rules you created when you built the database and created relationships between tables. What was the data type for this column? What was its maximum length? Was it a key? Can it be null? The onus is on you to make sure that the data you try to add to a table obeys its rules. As with dealing with data for display, the basics are straightforward, but you need to expend a little more effort to make the page user-friendly (and idiot-proof).

Inserting Data into the Database

You'll always have information to change and new data to collect, so providing a way to add new information to your databases is pretty crucial. Some sites may hide this functionality away in an administration section. How inserting data is handled depends on what the database models and who is logged on. For instance, Amazon hides the functionality to add new product information from you, the public, but it does let you add new feedback, and user information to its database, provided you're logged in. Similarly, eBay allows anyone to add a new auction to its database, but only the auctioneer can change those details. Security, then, is also a very important issue to consider.

The INSERT Query

At the heart of the code to add new information to a database is the SQL INSERT query. Although it may seem otherwise, sending an INSERT query to a database is the only way to do this. Compared to the complexities of the SELECT query, the INSERT query is quite simple.

```
INSERT [INTO] <table name>
[ (column list) ]
VALUES ( column value list )
```

The query doesn't need to be split over three lines, but that format makes it easier to see that it has six pieces:

- The keyword INSERT denotes the action to the database.
- The optional keyword INTO makes the query more readable.
- The *table name* identifies the table to which you're adding information.
- The (comma-separated) *column list* names the columns in the new row to which you're giving values. Although this isn't required, it is a good idea to specify it. It makes the query easier to follow and can reduce the risk of problems when you make changes to the database structure.
- The keyword VALUES separates the *column list* from the *column value list*.
- The (comma-separated) *column value list* contains a value for each of the columns in the *column list* for the new row. The number of the items in the *column list* should equal the number of items in the *column value list* and be ordered in the same way. Thus, the first column named in the *column list* will be filled with the first value in the *column value list*, the second with the second, and so on. Each value can be one of the following:

- A literal
- An expression saying how a value is to be determined from the values of other columns (firstname + surname, for example)
- The keyword `DEFAULT`, indicating that the column should take its default value as defined in the database
- `NULL`

With this in mind, it shouldn't be too difficult to construct a simple `INSERT` query for any of the four tables in the sample database. To insert a new Player, for instance, you could use the following query:

```
INSERT Player (PlayerName, PlayerManufacturerID, PlayerCost, PlayerStorage)
VALUES ('New Player', 1, '199.99', 'Solid State')
```

As you'll recall, the Player table actually has five columns, and you have not specified one of them. This isn't an error!

If a column is an identity column or has a default value, then you don't need to specify it when you're adding a new row; the database takes care of populating the column. So, even though you haven't specified the PlayerID column, the value is entered automatically by the database.

It's also possible to insert data into a database using the `INSERT` query without specifying the columns you want to insert the data into, as long as you specify the data for all the columns (bar the identity columns) in the order they appear in the database. Even columns that have default values must be specified.

So, you could change the previous `INSERT` query to the following without any problems:

```
INSERT Player
VALUES ('New Player', 1, '199.99', 'Solid State')
```

Although inserting data without specifying a list of columns is perfectly valid, it makes more sense to name the columns. As with the `SELECT` query, specifying the columns makes the query slightly quicker and shows which columns you're trying to affect. With the `INSERT` query, it also avoids putting data in the wrong column if columns have been added to or removed from the table.

■ **Note** One point to remember about `INSERT` is that it works with only a single table at a time. If you're working with complex data that would be sourced from two or more tables in a database, you'll need to write an `INSERT` query for each table to be updated. For example, to add details for a new Player to the sample database, you would have to write an `INSERT` query for both the Player table and the WhatPlaysWhatFormat table at the least. If the new Player were manufactured by a Manufacturer not in the database, you would need to create an `INSERT` query for the Manufacturer table as well.

The database-generation scripts in the code download for this book illustrate this point. The scripts contain `INSERT` queries for each row in each table, with each table populated in the correct order so that no data entry breaks any of the database constraints.

Working to the Database's Rules

Unlike playground rules, database rules aren't made to be broken, and you need to keep the following in mind when you're inserting new data into a table using `INSERT`:

Primary keys: You must provide a unique value for the column(s) in a table's primary key. If you don't, the database will return an error. Thus, you need to ensure that when you insert a new row into a table using `INSERT`, it contains a valid and unique value for the primary key. Things are a bit simpler if the primary key you use in the table is an identity column, such as the PlayerID column in the Player table, the ManufacturerID column in the Manufacturer table, or the FormatID column in the Format table. By establishing such a primary key, you can omit this column from the `INSERT` query's *`column list`*, because the database will automatically generate the value for you as you add the new row.

Foreign keys: If one of the columns in a table is a foreign key, you must ensure that any value you try to add to that column already exists as a value for the primary key in the corresponding table. When adding a Player, for example, the Manufacturer must exist before you can use it for the Player.

Mandatory columns: If a table doesn't allow a column to be null, you must give it a value when you add a new row. Either the user provides a value or you give it a default value when the user doesn't.

Column data types: Each column must be given a value of the appropriate type.

Each of these rules complicates things. Can you ensure that values are unique? What Web controls best suit data entry for each column? How do you enter a default value and make sure a column is given a certain value? You'll learn the answers to these questions as you work through this chapter's examples.

Try It Out: Inserting a New Player with INSERT

In this example, you'll build a page that allows you to add details of a new Player to the sample database. However, you won't include the selection of the supported Formats for the Player, which is handled in a later example.

1. In Visual Web Developer, create a new Web site at `C:\BAND\Chapter08` and delete the auto-generated `Default.aspx` file.

2. Add a new `Web.config` file to the Web site and add a new setting to the `<connectionStrings />` element:

```
<add name="SqlConnectionString"
  connectionString="Data Source=localhost\BAND;Initial Catalog=Players;
    Persist Security Info=True;User ID=band;Password=letmein"
  providerName="System.Data.SqlClient" />
```

3. Add a new Web Form to the Web site called `Players.aspx`. Make sure that the Place Code in Separate File check box is unchecked.

4. In the Source view, find the `<title>` tag within the HTML at the bottom of the page and change the page title to **Players**.

5. Switch to the Design view and add a `SqlDataSource` to the page. Choose to configure the data source and use `SqlConnectionString` to connect to the database. Select the PlayerID, PlayerName, and PlayerCost columns from the Player table to configure the `SELECT` query.

6. Switch back to the Source view and add the following markup after the definition of the `SqlDataSource`:

```
<asp:HyperLink ID="HyperLink1" runat="server"
  NavigateUrl="./Player_Insert.aspx">Add player</asp:HyperLink>
<br /><br />
<asp:GridView ID="GridView1" runat="server" AutoGenerateColumns="False"
  DataSourceID="SqlDataSource1">
  <Columns>
    <asp:BoundField DataField="PlayerID" HeaderText="PlayerID" />
    <asp:BoundField DataField="PlayerName" HeaderText="Name" />
    <asp:BoundField DataField="PlayerCost" DataFormatString="{0:n}"
      HeaderText="Cost" />
  </Columns>
</asp:GridView>
```

7. Add a new Web Form to the Web site called `Player_Insert.aspx`. Make sure that the Place Code in Separate File check box is unchecked.

8. In the Source view, find the `<title>` tag within the HTML at the bottom of the page and change the page title to **INSERT Player**. Add the required `Import` statement to the top of the page:

```
<%@ Page Language="C#" %>
<%@ Import Namespace="System.Data.SqlClient" %>
```

9. Add some Web controls to allow the addition of the Player to the database: a `Button` to insert the Player, a `Button` to return to the list of Players, a `TextBox` for the user to add the Player's name, a `DropDownList` for the Manufacturer, a `TextBox` for the Player's cost, and a final `TextBox` to specify the storage type for the Player. Call these `SubmitButton`, `ReturnButton`, `PlayerName`, `ManufacturerList`, `PlayerCost`, and `PlayerStorage`, respectively. Also add a `Label`, called `QueryResult`, to show the results from the query that was actually executed. You can see how the Web controls are laid out in Figure 8-1.

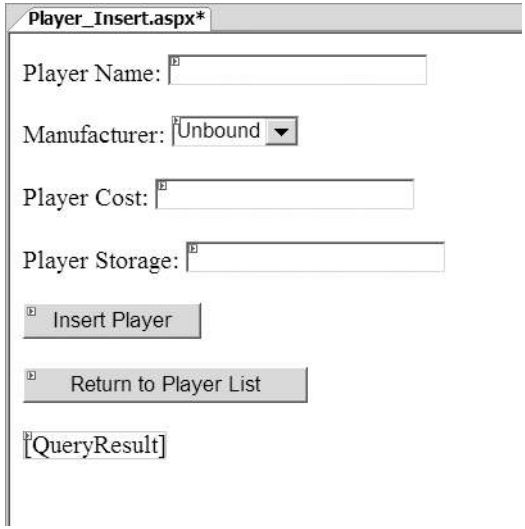

Figure 8-1. *The Web control layout for Player_Insert.aspx*

10. Add a `Page_Load` event handler to the page, as follows:

```
protected void Page_Load(object sender, EventArgs e)
{
  if (Page.IsPostBack == false)
  {
    // populate the list of manufacturers
    PopulateManufacturers();
  }
}
```

11. Add the `PopulateManufacturers()` method:

```
private void PopulateManufacturers()
{
  // create the connection
  string strConnectionString = ConfigurationManager.
    ConnectionStrings["SqlConnectionString"].ConnectionString;
  SqlConnection myConnection = new SqlConnection(strConnectionString);

  try
  {
    // query to execute
    string strQuery = "SELECT ManufacturerID, ManufacturerName ➥
      FROM Manufacturer ORDER BY ManufacturerName";

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

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

    // run query
    SqlDataReader myReader = myCommand.ExecuteReader();

    // set the data source and bind
    ManufacturerList.DataSource = myReader;
    ManufacturerList.DataTextField = "ManufacturerName";
    ManufacturerList.DataValueField = "ManufacturerID";
    ManufacturerList.DataBind();

    // close the reader
    myReader.Close();
  }
  finally
  {
    // always close the database connection
    myConnection.Close();
  }
}
```

12. Switch to the Design view of the page and add a `DataBound` event handler for the `ManufacturerList` control. Add the following code to the event handler:

```
protected void ManufacturerList_DataBound(object sender, EventArgs e)
{
  ListItem myListItem = new ListItem();
  myListItem.Text = "please select...";
  myListItem.Value = "0";
  ManufacturerList.Items.Insert(0, myListItem);
}
```

13. With the Web control layout sorted and populated as required, you need to implement the code to insert the Player into the database. Switch back to the Design view of the page and double-click the `SubmitButton` control to add a `Click` event handler. Add the following code to the event handler:

```
protected void SubmitButton_Click(object sender, EventArgs e)
{
  // save the player to the database
  int intPlayerID = SavePlayer();

  // did an error occur?
  if (intPlayerID == -1)
  {
    QueryResult.Text = "An error has occurred!";
  }
```

```
    else
    {
      // show the result
      QueryResult.Text = "Save of player '" + intPlayerID.ToString()
        + "' was successful";

      // disable the submit button
      SubmitButton.Enabled = false;
    }
  }
```

14. To insert the Player into the database, you call a function named `SavePlayer()`. This function returns the PlayerID for the new entry, or it returns `-1` if an error occurs:

```
  private int SavePlayer()
  {
    int intPlayerID = 0;

    // create the connection
    string strConnectionString = ConfigurationManager.
      ConnectionStrings["SqlConnectionString"].ConnectionString;
    SqlConnection myConnection = new SqlConnection(strConnectionString);

    try
    {
      // query to execute
      string strQuery = "INSERT Player (PlayerName, PlayerManufacturerID, ➥
        PlayerCost, PlayerStorage) VALUES (@Name, @ManufacturerID, ➥
        @Cost, @Storage); SELECT SCOPE_IDENTITY();";

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

      // add the parameters
      myCommand.Parameters.AddWithValue("@Name", PlayerName.Text);
      myCommand.Parameters.AddWithValue("@ManufacturerID",
        ManufacturerList.SelectedValue);
      myCommand.Parameters.AddWithValue("@Cost", PlayerCost.Text);
      myCommand.Parameters.AddWithValue("@Storage", PlayerStorage.Text);

      // open the connection
      myConnection.Open();

      // execute the query
      intPlayerID = Convert.ToInt32(myCommand.ExecuteScalar());
    }
    catch
    {
```

```
    // return -1 to indicate error
    intPlayerID = -1;
  }
  finally
  {
    // close the connection
    myConnection.Close();
  }

  // return the ID
  return(intPlayerID);
}
```

15. Finally, you need to provide a means for the user to return to the list of Players. Switch to the Design view and double-click the `ReturnButton`. Add the following code to the `Click` event handler:

```
protected void ReturnButton_Click(object sender, EventArgs e)
{
  Response.Redirect("./Players.aspx");
}
```

16. Save the page, and then open the Web site in your browser. In the list of Players, click the Add Player link. On the following page, add the details for a new Player. Then click the Insert Player button to execute the `INSERT` query and return the ID of the Player added, as shown in Figure 8-2.

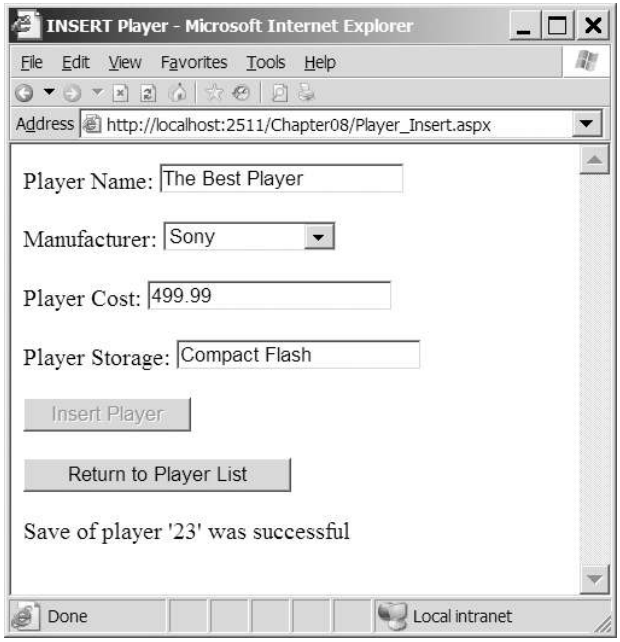

Figure 8-2. *Adding a new Player to the database*

17. Click the Return to Player List button. You'll see that the new Player has been added to the end of the list of Players, as shown in Figure 8-3.

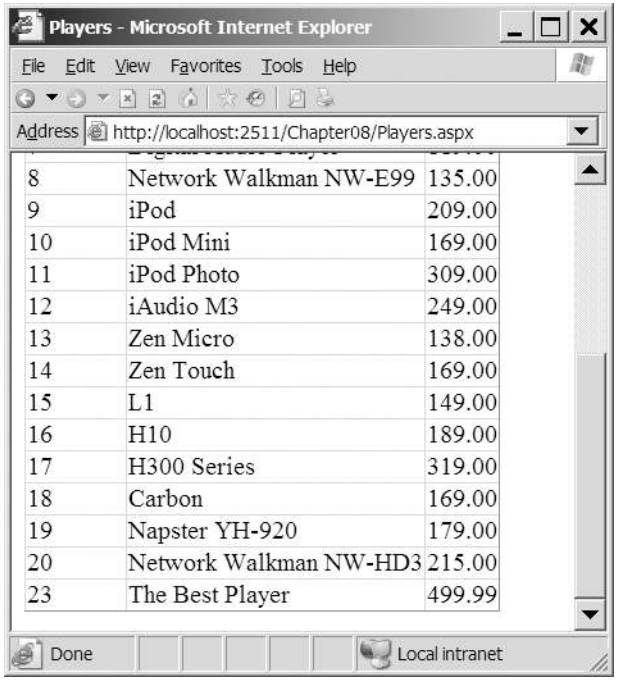

Figure 8-3. *The new Player added to the database*

How It Works

This example has provided you with the means to add a Player to the database. The first 12 steps of the example should be quite familiar to you by now. You built a page that lists basic details for all of the Players in the database using a `SqlDataSource` and a `GridView`. We looked at table binding a `GridView` in Chapter 7.

The second page is the one that lets you insert data into the database.

Web Control Selection

The first stage of this page needs to take the rules of the sample database into consideration. You're adding a new row to the Player table, so the first task is to figure out which Web control is most suitable for adding the value for each column, as follows:

PlayerID: This is the primary key for the Player table, but it's also an identity column, so you don't need to insert a value for this column. It will be added for you automatically.

PlayerName: A Player's name is just text, so a `TextBox` is appropriate.

PlayerManufacturerID: This is a foreign key from the Manufacturer table, so it can hold only values already in the Manufacturer table. It makes sense to give the user a choice of Manufacturers from a list, so you use a `DropDownList` and bind the ManufacturerName to `DataTextField` and the ManufacturerID to `DataValueField`. You could use any data-aware list Web control, but `DropDownList` works fine.

PlayerCost: The cost of the Player is a decimal, and the best way for entering this value is using a `TextBox`.

PlayerStorage: At this point, it becomes obvious that the sample database design is (deliberately) flawed and that the Storage Type entries should really be in their own table. This would mean you could bind the available Storage Types to a list Web control and keep control of the Storage Types for the Players. But since the types are in the Player table, we're allowing users to enter any Storage Type that they want. This is a good example of one of the repercussions of bad database design.

The list of Manufacturers is populated using a simple query to return just the ManufacturerID and ManufacturerName columns from the Manufacturer table. You saw how to do this in Chapter 6. You could also have used a `SqlDataSource` to populate the `DropDownList`.

Once the Web controls are set up as required, the user can enter the details of the new Player and click the Insert Player button to save the Player to the database. Here, you see the first problem with the page.

Error Handling

Rerun the Web site and enter a new Player without a name, cost, or storage type. Now save the Player. Instead of the Player being saved, an error has been trapped and an error message displayed, as shown in Figure 8-4.

Entering invalid data and trying to save it causes a `SqlException` to be raised and handled by the `catch` clause of your data-access code. If you add a breakpoint to the code within the `catch` clause, you'll see that the exception is thrown because you're trying to convert an empty string (an `nvarchar`) to a numeric value, and it's not a valid cast. Figure 8-5 shows this information.

A multitude of different errors can arise if you don't validate entries made by the user when inserting and updating data to the database. We'll look at validating the user's input in the "Validating Data" section later in this chapter, and you'll update this example so that invalid data can't make its way to the database. For now, you must enter values in all of the columns.

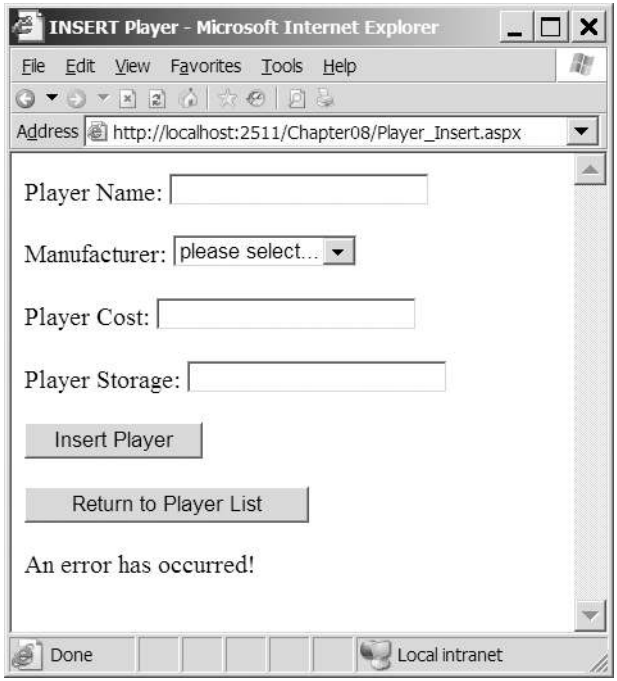

Figure 8-4. *Invalid data causes exceptions, which thankfully are trapped.*

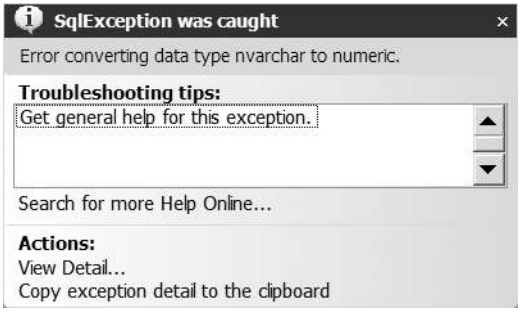

Figure 8-5. *When debugging, you can see the details of the raised exception.*

Database Record Insertion

The part of the example of particular interest is the `SubmitButton_Click` event handler:

```
protected void SubmitButton_Click(object sender, EventArgs e)
{
  // save the player to the database
  int intPlayerID = SavePlayer();
```

```
  // did an error occur?
  if (intPlayerID == -1)
  {
    QueryResult.Text = "An error has occurred!";
  }
  else
  {
    // show the result
    QueryResult.Text = "Save of player '" + intPlayerID.ToString()
      + "' was successful";

    // disable the submit button
    SubmitButton.Enabled = false;
  }
}
```

The `SubmitButton_Click` event handler calls the `SavePlayer()` method to save the Player to the database, and this method returns the PlayerID value for the new Player, or it returns `-1` if an error has occurred. If you don't have a valid PlayerID, you know that something has gone wrong, and you display an error message to the user. If the returned PlayerID is valid (not equal to `-1`), you can assume that the Player has been added to the database, and you display a success message showing the PlayerID of the Player just added to the database. If the Player has been added successfully, you also disable the `SubmitButton`, so that you can't save the details for the same Player twice by accident. (Of course, there is nothing stopping the user from adding the exact same Player again and again!)

The `SavePlayer()` method is responsible for taking the details entered by the user and saving these details, using an `INSERT` query to the database:

```
INSERT Player (PlayerName, PlayerManufacturerID, PlayerCost, PlayerStorage)
VALUES (@Name, @ManufacturerID, @Cost, @Storage);
```

This should look familiar, as it was the example used in the earlier introduction to the `INSERT` query, but instead of having actual values in the *`column value list`*, you're using parameters that you add using the `AddWithValue` method:

```
// add the parameters
myCommand.Parameters.AddWithValue("@Name", PlayerName.Text);
myCommand.Parameters.AddWithValue("@ManufacturerID",
  ManufacturerList.SelectedValue);
myCommand.Parameters.AddWithValue("@Cost", PlayerCost.Text);
myCommand.Parameters.AddWithValue("@Storage", PlayerStorage.Text);
```

You're using a parameterized `INSERT` query in the interest of security. You could just build the query with string concatenation, but as you learned in Chapter 4, parameters prevent users from trying to harm your database by sending malevolent SQL instructions through the `TextBox`.

Once the Command object is created and the parameters added correctly, you execute the query against the database. But if you look at the code for the page, you'll see that the query that you're going to execute isn't quite what you just saw. In fact, the query that you send to the database is actually a *query batch* of two separate SQL queries: an `INSERT` and a `SELECT` query:

```
INSERT Player (PlayerName, PlayerManufacturerID, PlayerCost, PlayerStorage)
  VALUES (@Name, @ManufacturerID, @Cost, @Storage);
SELECT SCOPE_IDENTITY();
```

When sending a query to a SQL Server 2005 database, you can actually send multiple queries, separated by semicolons. You want to insert the Player into the database, but you also want to know the PlayerID of the Player that you've added. In this example, you display the PlayerID as a confirmation that the Player has been added to the database. In the next example, you'll use this PlayerID when adding the details of the Formats that the Player supports.

When using a column defined as an identity column, you can use the `SCOPE_IDENTITY()` function to retrieve the value of that column. In order to return the value from this function, you can use it as a column in a `SELECT` query.

■ **Note** The system variable `@@IDENTITY` returns the value of the identity column last entered. In this instance, both the `SCOPE_IDENTITY()` function and the `@@IDENTITY` system variable would return the same value. However, in cases when you're using triggers, the `@@IDENTITY` system variable may return the wrong value; if the trigger also does an `INSERT`, it may return the identity value from a different table, whereas the `SCOPE_IDENTITY()` function returns the identity value from the original table. You should always use the `SCOPE_IDENTITY()` function to prevent any problems if triggers are added to your tables later.

Thus, when you execute this query, you do so by calling `ExecuteScalar()` rather than `ExecuteNonQuery()` so you can capture the new PlayerID. `ExecuteScalar()` returns a generic object, rather than a string or an integer, so you cast it to an integer to make it easier to handle:

```
// execute the query
intPlayerID = Convert.ToInt32(myCommand.ExecuteScalar());
```

If there was an error when inserting the Player, an exception is thrown. You catch this and set the PlayerID to `-1` to indicate that the `INSERT` query failed:

```
catch
{
  // return -1 to indicate error
  intPlayerID = -1;
}
```

Although all you're doing here is setting a flag to indicate that there has been an error, you're free to perform any other actions you want. If you want to send an e-mail message to the Web site administrator informing her that a problem has occurred, you can do so. Just be careful that your error-handling code doesn't throw an exception, as that would cause a runtime error to be displayed to the user!

Once you've executed the query and returned the PlayerID, you exit from the `SavePlayer()` method and either display an error message or a confirmation to the user. At this point, the user can return to the list of Players to confirm that the new Player has been added and to add another Player if desired.

Queries in MySQL 5.0 and Microsoft Access

Before we move on to the next example and add the Format information, we'll quickly look at two areas where SQL Server 2005 differs from MySQL 5.0 and Microsoft Access:

- Only when using the SQL Server 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 OLE DB data provider (which we use to connect to Microsoft Access) support named parameters. With those data providers, you need to add parameters in the order in which they appear in the query.
- Neither MySQL 5.0 nor Microsoft Access allows multiple queries to be executed as part of the same query batch, and neither supports the `SCOPE_IDENTITY()` function.

Parameters and Queries in MySQL 5.0 and Microsoft Access

As you've learned in earlier chapters, you can't use named parameters with MySQL 5.0 or Microsoft Access using the `Odbc` or `OleDb` data providers. You need to change the query that you want to execute and add the parameters to the parameters collection in the correct order.

For MySQL 5.0, replace the named parameters with the question mark character:

```
INSERT Player (PlayerName, PlayerManufacturerID, PlayerCost, PlayerStorage)
VALUES (?, ?, ?, ?)
```

The query required for Microsoft Access is similar, except you must also specify the `INTO` keyword:

```
INSERT INTO Player (PlayerName, PlayerManufacturerID, PlayerCost,
  PlayerStorage)
VALUES (?, ?, ?, ?)
```

Once the query is defined correctly, the parameters are added in the order in which they're required:

```
myCommand.Parameters.AddWithValue("?", PlayerName.Text);
myCommand.Parameters.AddWithValue("?", ManufacturerList.SelectedValue);
myCommand.Parameters.AddWithValue("?", PlayerCost.Text);
myCommand.Parameters.AddWithValue("?", PlayerStorage.Text);
```

Identity Values and MySQL 5.0 and Microsoft Access

Retrieving the identity value for a new row in a table requires two different queries to be executed. SQL Server 2005 allows you to execute these queries as part of the same query batch to the database by separating the queries with semicolons. However, neither MySQL 5.0 nor Microsoft Access supports this functionality. Therefore, you need to make two distinct queries to the database: the `INSERT` query to add the Player and a `SELECT` query to return the PlayerID.

For MySQL 5.0, this is relatively easy, as there is a corresponding function: `LAST_INSERT_ID()` returns the value you're after. So, you create two queries and execute these one after the other:

```
// create the INSERT query
string strQuery1 = "INSERT Player (PlayerName, PlayerManufacturerID, ➥
  PlayerCost, PlayerStorage) VALUES (?, ?, ?, ?);";
OdbcCommand myCommand1 = new OdbcCommand(strQuery1, myConnection);

// add the parameters
myCommand1.Parameters.AddWithValue("?", PlayerName.Text);
myCommand1.Parameters.AddWithValue("?", ManufacturerList.SelectedValue);
myCommand1.Parameters.AddWithValue("?", PlayerCost.Text);
myCommand1.Parameters.AddWithValue("?", PlayerStorage.Text);

// create the SELECT query
string strQuery2 = "SELECT LAST_INSERT_ID();";
OdbcCommand myCommand2 = new OdbcCommand(strQuery2, myConnection);

// open the connection
myConnection.Open();

// execute the queries we need to execute
myCommand1.ExecuteNonQuery();
intPlayerID = Convert.ToInt32(myCommand2.ExecuteScalar());

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

You'll still wrap all of the above code in a `try..catch..finally` block, so that if there is a problem, you can set the PlayerID value to `-1` to indicate that an error occurred.

To get the identity value in Microsoft Access, you can use the `@@IDENTITY` system variable. So, simply execute a different query to return the identity value:

```
// create the SELECT query
string strQuery2 = "SELECT @@IDENTITY;";
OleDbCommand myCommand2 = new OleDbCommand(strQuery2, myConnection);

// open the connection
myConnection.Open();

// execute the queries we need to execute
myCommand1.ExecuteNonQuery();
intPlayerID = Convert.ToInt32(myCommand2.ExecuteScalar());
```

You'll see these versions of getting the value of the PlayerID column in the code download for this book.

Try It Out: Setting the Player's Supported Formats

Now that we've looked at how to add the basic details for the Player, let's see how to add the Player's supported Formats.

1. Open `Players_Insert.aspx` and switch to the Design view.

2. Add a new `CheckBoxList` before the Insert Player button. Rename it `FormatList`, set its `RepeatColumns` property to 4 and its `RepeatDirection` to Horizontal. The layout should now be as shown in Figure 8-6.

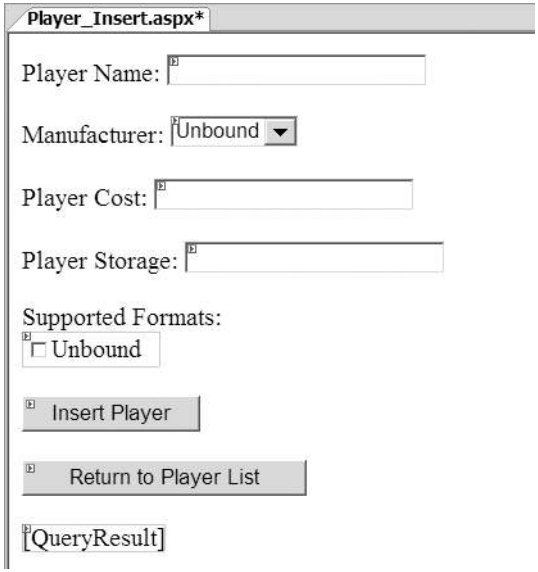

Figure 8-6. *The new layout showing the Supported Formats CheckBoxList*

3. Switch to the Source view and modify the `Page_Load` event as follows:

```
protected void Page_Load(object sender, EventArgs e)
{
  if (Page.IsPostBack == false)
  {
    // populate the list of manufacturers
    PopulateManufacturers();

    // populate the list of formats
    PopulateFormats();
  }
}
```

4. Add the new `PopulateFormats()` method:

```
private void PopulateFormats()
{
  // create the connection
  string strConnectionString = ConfigurationManager.
    ConnectionStrings["SqlConnectionString"].ConnectionString;
  SqlConnection myConnection = new SqlConnection(strConnectionString);
```

```
    try
    {
      // query to execute
      string strQuery = "SELECT FormatID, FormatName FROM Format ➥
        ORDER BY FormatName";

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

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

      // run query
      SqlDataReader myReader = myCommand.ExecuteReader();

      // set the data source and bind
      FormatList.DataSource = myReader;
      FormatList.DataTextField = "FormatName";
      FormatList.DataValueField = "FormatID";
      FormatList.DataBind();

      // close the reader
      myReader.Close();
    }
    finally
    {
      // always close the database connection
      myConnection.Close();
    }
  }
```

5. Modify the `SubmitButton_Click` event handler as follows:

```
protected void SubmitButton_Click(object sender, EventArgs e)
{
  // save the player to the database
  int intPlayerID = SavePlayer();

  // did an error occur?
  if (intPlayerID == -1)
  {
    QueryResult.Text = "An error has occurred!";
  }
  else
  {
```

```
    // save the formats for the player
    bool blnError = SaveFormats(intPlayerID);

    // did an error occur?
    if (blnError == true)
    {
      QueryResult.Text = "An error has occurred!";
    }
    else
    {
      // show the result
      QueryResult.Text = "Save of player '" + intPlayerID.ToString()
        + "' was successful";

      // disable the submit button
      SubmitButton.Enabled = false;
    }
  }
}
```

6. Add the new `SaveFormats()` method as follows:

```
private bool SaveFormats(int intPlayerID)
{
  bool blnError = false;

  // create the connection
  string strConnectionString = ConfigurationManager.
    ConnectionStrings["SqlConnectionString"].ConnectionString;
  SqlConnection myConnection = new SqlConnection(strConnectionString);

  try
  {
    // query to execute
    string strQuery = "INSERT WhatPlaysWhatFormat(WPWFPlayerID, ➥
      WPWFFormatID) VALUES (@PlayerID, @FormatID)";

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

    // add the two parameters
    myCommand.Parameters.AddWithValue("@PlayerID", intPlayerID);
    myCommand.Parameters.Add("@FormatID", System.Data.SqlDbType.Int);

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

```
        // loop through each of the formats
        foreach (ListItem objFormat in FormatList.Items)
        {
          // save if selected
          if (objFormat.Selected == true)
          {
            // set the parameter value
            myCommand.Parameters["@FormatID"].Value = objFormat.Value;

            // execute the query
            myCommand.ExecuteNonQuery();
          }
        }
      }
      catch
      {
        // indicate that we have an error
        blnError = true;
      }
      finally
      {
        // close the connection
        myConnection.Close();
      }

      // return the error flag
      return(blnError);
    }
```

7. Save the page, and then open the Web site in your browser. In the list of Players, click the Add Player link, and you'll see that the list of Formats is populated. Enter the details for a new Player, and this time, specify the Formats that the Player supports.

8. Click the Insert Player button to save the Player to the database, along with the Formats it supports, as shown in Figure 8-7.

9. To see that the Format details have been saved to the database correctly, you can perform a `SELECT` query against the WhatPlaysWhatFormat table. The Player added has a PlayerID of 27, so look for this in the WPWFPlayerID column. As you can see in Figure 8-8, the two Formats have been added correctly.

Figure 8-7. *The supported Formats are also saved.*

	19	4
	20	4
	20	2
	20	6
	27	3
	27	2

Figure 8-8. *The supported Formats have been saved to the database.*

How It Works

With a bit more work, you've added the ability to save the supported Formats for a Player to the database. In order to do this, you need to perform two steps:

- Display the options on the page that may be selected in a suitable Web control.
- Save the selected options correctly to the database.

Because of the way that you structured the code for the first example, you can quite easily perform these two steps by adding two new functions: `PopulateFormats()` and `SaveFormats()`.

The `PopulateFormats()` method displays the available Formats. You're showing a list of Formats, so you're going to be using one of the Web controls that support list binding, as described in Chapter 6. You need to allow the user to select multiple options, so you have only two choices: a `CheckBoxList` or a `ListBox`. You've used a `CheckBoxList` as it allows the selection of multiple entries with a single click, so the user doesn't need to use a combination of keyboard presses and mouse clicks to select multiple Formats.

The code within the `PopulateFormats()` method should be familiar to you by now. You return a list of FormatID and FormatName pairs using a simple query, and then set the `DataTextField` and `DataValueField` properties on the `CheckBoxList`.

It's within the `SaveFormats()` method that the real work occurs. The first part of the method should be familiar by now. You create a Connection object to connect to the database and a Command object populated with the correct `INSERT` query:

```
INSERT WhatPlaysWhatFormat(WPWFPlayerID, WPWFFormatID)
VALUES (@PlayerID, @FormatID)
```

Both parameters are then added to the `Parameters` collection:

```
// add the two parameters
myCommand.Parameters.AddWithValue("@PlayerID", intPlayerID);
myCommand.Parameters.Add("@FormatID", System.Data.SqlDbType.Int);
```

You already know the `@PlayerID` parameter value, as you retrieved it from the `SavePlayer()` method, and it is fixed for this Player, so you can use the `AddWithValue()` method to add it. But the `@FormatID` parameter is different. You're allowing the user to select multiple values, so you can't just add the parameter value and execute the query. You can add the parameter without a value using the `Add()` method, and then set its value later before you execute the query.

You check which Formats have been selected by using a `foreach` loop to work through all of the possible Formats (returned as `ListItem` objects from the `Items` collection of the `CheckBoxList`) and execute the `INSERT` query for each Format that is selected:

```
// loop through each of the formats
foreach (ListItem objFormat in FormatList.Items)
{
  // save if selected
  if (objFormat.Selected == true)
  {
    // set the parameter value
    myCommand.Parameters["@FormatID"].Value = objFormat.Value;

    // execute the query
    myCommand.ExecuteNonQuery();
  }
}
```

If the Format is selected, the `Selected` property will return `true`, and you can set the parameter value to be the `Value` of the `ListItem`. The `ExecuteNonQuery()` method is then used to execute the `INSERT` query against the database.

Note that you open the connection to the database only once, and you close the database connection only once. You're reusing the connection for each of the `INSERT` queries that you're executing. Now this goes a little against my "open late, close early" mantra, as you're keeping the connection open for longer than is necessary. However, in this case, the `foreach` loop is quick enough for the open connection to not be an issue.

Validating Data

The previous two examples have demonstrated how to add data to the database using `INSERT` queries. You also saw that it's quite easy to cause runtime errors by not entering valid data. Although you trapped these errors and handled them before the user saw the dreaded ASP.NET runtime error page, it would be much better to guard against these errors before they occur. You need to *validate* the data that the user enters before you attempt to insert the data into the database. The same is also true when you update data, as you'll see later in this chapter.

Whether you want to ensure that the user has entered a value of the required format (such as an e-mail address), entered a value within a range (such as a number between 1 and 10), or entered any value, ASP.NET provides Web controls to perform validation for you. The different validation Web controls are shown in Table 8-1. In addition to the Web controls for actually validating the user's input, another Web control displays the results of the validation: `ValidationSummary`.

Table 8-1. *The Validation Controls*

Name	Description
`CompareValidator`	The `CompareValidator` compares the value entered by the user with either a constant value or the value entered in another Web control.
`RangeValidator`	The `RangeValidator` checks that the value entered is between two specified values.
`RegularExpressionValidator`	The `RegularExpressionValidator` checks that the value entered matches the specified regular expression.
`RequiredFieldValidator`	The `RequiredFieldValidator` checks that a Web control contains a value.
`CustomValidator`	If none of the other four validators match your requirements, the `CustomValidator` allows you to define your own validation routines.

Each of the validation Web controls can be executed either at the client or at the server. If you choose to use client-side validation (the default), a postback will not occur, giving a richer user experience. However, in certain cases, a postback must occur before validation can continue. For example, if you're validating for a unique username when creating a user account, you must check against the database that the username hasn't already been used.

■ **Caution** In the default state of the validation Web controls, the validation is performed at the client as well as at the server. However, it is quite possible for users to disable JavaScript in their browser and turn off the client-side validation, and post the page to the server with invalid data. The validation always runs at the server, even if you have client-side validation turned on. You should always check that any validation has been done before allowing changes to be made to the database.

The validation Web controls all expose a `ControlToValidate` property that specifies the ID of the Web control that is to be validated. The Web controls that can be validated automatically are shown in Table 8-2. For those Web controls that don't support automatic validation, you'll need to use a `CustomValidator` control for validation. For example, since there's no way to automatically validate that a user selected a value from a `CheckBoxList`, in our example, you'll have to write a `CustomValidator` to ensure that the user has selected at least one supported Format for the new Player.

Table 8-2. *Controls That Can Be Validated Automatically*

Control	Property Validated
`DropDownList`	`SelectedItem`
`FileUpload`	`FileBytes`
`HtmlInputFile`	`Value`
`HtmlInputPassword`	`Value`
`HtmlInputText`	`Value`
`HtmlSelect`	`Value`
`HtmlTextArea`	`Value`
`ListBox`	`SelectedItem`
`RadioButtonList`	`SelectedItem`
`TextBox`	`Text`

Table 8-2 also lists the property of the Web control that the validator accesses to perform the validation. At this point, warning signs should be flashing. How can you use the `SelectedItem` from a `ListBox` with a `RegularExpressionValidator`? The validation Web controls are a little more clever than you may initially think, and for the Web list controls, the validator will actually look at the `Value` property of the `SelectedItem`.

Now let's try using the various validation Web controls to improve the page for entering a new Player.

Try It Out: Validating Entered Data

In this example, you'll update the previous example to add validation Web controls to prevent the user from entering incorrect data into the database.

1. Open `Players_Insert.aspx` and switch to the Design view.
2. Add a `ValidationSummary` from the Validation tab of the Toolbox to the top of the page.
3. Add a `RequiredFieldValidator` to the start of the Player Name line. Set its `Display` property to `Dynamic`, `Text` property to *, and `ErrorMessage` property to **You must enter a name**. Finally set the `ControlToValidate` property to PlayerName. The page should look like the one shown in Figure 8-9.

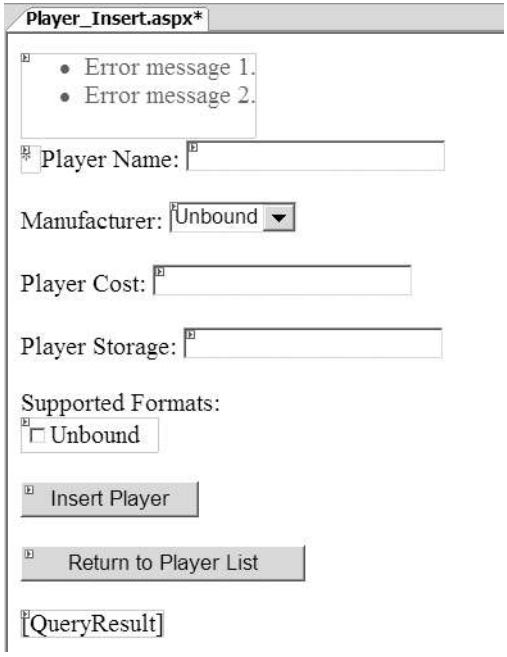

Figure 8-9. *Adding the first validation Web controls*

4. Save the page, and then view it in your browser. Try saving a Player without a name. As soon as you click the Insert Player button, you'll receive an error, as shown in Figure 8-10, without a postback to the server being made.

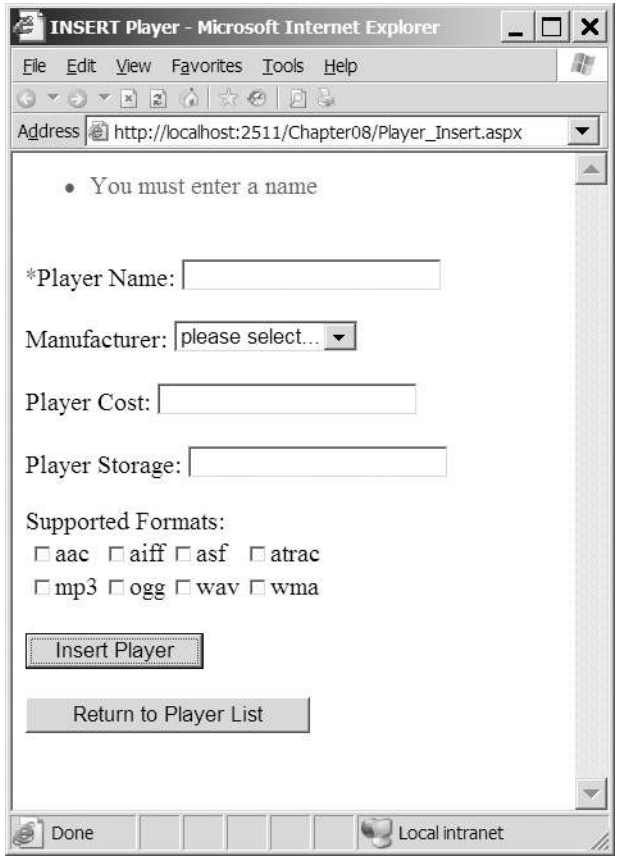

Figure 8-10. *The validation Web controls in action*

5. Add a `CompareValidator` to the start of the Manufacturer line. Set its properties as follows:
   - `Display`: Dynamic
   - `Text`: *
   - `ErrorMessage`: You must select a manufacturer
   - `ControlToValidate`: ManufacturerList
   - `Operator`: NotEqual
   - `ValueToCompare`: 0

6. Add a `RequiredFieldValidator` to the start of the Player Cost line. Set its properties as follows:
   - `Display`: Dynamic
   - `Text`: *
   - `ErrorMessage`: You must enter a cost
   - `ControlToValidate`: PlayerCost
7. Add a `RegularExpressionValidator` to the start of the Player Cost line. Set its properties as follows:
   - `Display`: Dynamic
   - `Text`: *
   - `ErrorMessage`: You must specify the cost as a decimal
   - `ControlToValidate`: PlayerCost
   - `ValidationExpression`: `^\d+(\.\d\d)`
8. Add a `RequiredFieldValidator` to the start of the Player Storage line. Set its properties as follows:
   - `Display`: Dynamic
   - `Text`: *
   - `ErrorMessage`: You must enter a storage type
   - `ControlToValidate`: PlayerStorage
9. Add a `CustomValidator` to the start of the Supported Formats text. Set its properties as follows:
   - `Display`: Dynamic
   - `Text`: *
   - `ErrorMessage`: You must select at least one format
10. Double-click the `CustomValidator` to add the server-side validation event. Add the following code to the event handler:

```
protected void CustomValidator1_ServerValidate(object source,
  ServerValidateEventArgs args)
{
  if (FormatList.SelectedIndex == -1)
  {
    args.IsValid = false;
  }
}
```

11. Save the page, and then view it in your browser. Try testing the different combinations of validators. You'll see that they won't let you save the Player until all of the data entered is valid. But there's still a problem. Enter the data correctly, but don't select any Formats. As you can see in Figure 8-11, you'll get the correct validation error, but the Player will still been saved.

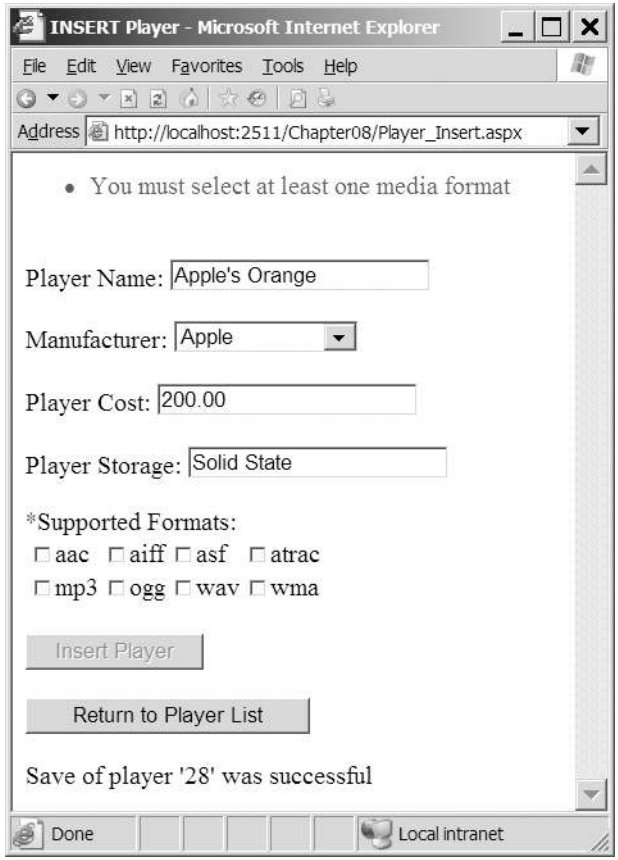

Figure 8-11. *A validation error occurs, but the Player is still saved.*

12. You want to save the page only if all of the validators on the page are valid. Modify the `SubmitButton_Click` event handler as follows:

```
protected void SubmitButton_Click(object sender, EventArgs e)
{
  // only save if valid
  if (Page.IsValid == true)
  {
    // save the player to the database
    int intPlayerID = SavePlayer();
```

```
        // did an error occur?
        if (intPlayerID == -1)
        {
          QueryResult.Text = "An error has occurred!";
        }
        else
        {
          // save the formats for the player
          bool blnError = SaveFormats(intPlayerID);

          // did an error occur?
          if (blnError == true)
          {
            QueryResult.Text = "An error has occurred!";
          }
          else
          {
            // show the result
            QueryResult.Text = "Save of player '" + intPlayerID.ToString()
              + "' was successful";

            // disable the submit button
            SubmitButton.Enabled = false;
          }
        }
      }
    }
```

13. Rerun the page, and now try to save the new Player without any Formats selected. This time, the Player won't be saved. Select at least one Format, and the page will now add the Player to the database.

How It Works

In this example, you used four out of the five available validators to prevent the user from saving a Player to the database with incorrect data, and you used both client and server-side validators.

Validator Properties

Although you used four different validator types on this page, three properties are common across all of the validators:

- `Display`: This determines how the validator is displayed on the page. The default value of `Static` always reserves space for the Web control on the page, even if it isn't being displayed. Setting this property to `None` will not show the Web control on the page (although the `ErrorMessage` will appear in a `ValidationSummary`). Setting it to `Dynamic` displays the Web control only if it's invalid.

- `ErrorMessage`: The `ErrorMessage` property sets the text that will be displayed in a `ValidationSummary` (if one exists on the page) if the validation for the Web control fails.
- `Text`: The `Text` property sets the text displayed as the validator when validation fails. When you're using a `ValidationSummary`, the `Text` property is usually used simply to highlight which validator has failed.

All of the validation Web controls, other than the `CustomValidator`, also set the `ControlToValidate` property. This specifies the ID of the Web control that is being validated. You can also set this property for a `CustomValidator`, but in the majority of cases, it won't be used. If you're using a `CustomValidator` because a normal validator won't work, you'll need to interrogate the Web control directly in code.

The ValidationSummary Web Control

The first Web control that you have on the page is the `ValidationSummary`. It is here that the different validation Web controls display their `ErrorMessage` if the validation fails. It isn't necessary to have this Web control on the page for validation to work, but it provides a handy location for all of the validation errors to be displayed.

The RequiredFieldValidator Web Control

The `RequiredFieldValidator` needs no configuration other than the `Display`, `ErrorMessage`, `Text`, and `ControlToValidate` properties that we've already discussed. As you saw in the example, if you don't specify a value for a Web control that has a `RequiredFieldValidator` attached to it, the validation will fail.

The CompareValidator Web Control

The `CompareValidator` allows you to compare the value of the attached Web control against either another Web control (using the `ControlToCompare` property) or against a specific value (using the `ValueToCompare` property). In this example, you need to ensure that a Manufacturer has been selected, and you can use a specific value comparison, since you know that the "please select..." entry has a value of 0 (zero). You can then set the `ValueToCompare` property to `0` and set the `Operator` property to `NotEqual` to ensure that the user has selected a value that isn't equal to 0.

As well as checking for `NotEqual` comparisons, the `Operator` also allows you to perform various other checks:

- `Equal` (default)
- `GreaterThan`
- `GreaterThanEqual`
- `LessThan`
- `LessThanEqual`
- `NotEqual`

The CompareValidator also allows you to perform a further validation check. By setting the Operator property to DataTypeCheck, you can check that the value of the ControlToValidate is of a specific type. If you choose this type of check, the ControlToCompare and ValueToCompare properties are ignored, and only a check for a value that is of the correct type is performed. You specify the type allowed using the Type property, which can be one of the following values:

- Currency
- Date
- Double
- Integer
- String (default)

■ **Note** In this example, you could have used a CompareValidator to ensure that the cost of the Player was entered correctly. However, you used a RegularExpressionValidator rather than a CompareValidator, simply because it demonstrated another type of validator.

The RegularExpressionValidator Web Control

The RegularExpressionValidator allows you to check that the value entered matches a specific regular expression by setting the ValidationExpression property to the regular expression you want to use.

Any valid regular expression can be used, and Visual Web Developer provides you with several standard ones (an e-mail address and Web address, for instance). You're checking that the entered value is a decimal, so you need to define your own regular expression as follows:

```
^\d+(\.\d\d)
```

A decimal is any number of digits followed by a decimal point, then two decimal digits.

■ **Tip** A good place to look for regular expressions that meet your requirements is http://www.regexlib.com. Also refer to *Regular Expression Recipes for Windows Developers: A Problem-Solution Approach* by Nathan A. Good (1-59059-497-5; Apress, 2005).

The CustomValidator Web Control

The CustomValidator allows you to perform any validation that you require. You can perform this validation on the client side if you write a function in JavaScript and then pass its name to the ClientValidationFunction property. However, in most cases, you'll run the validation on the server by providing an implementation for the ServerValidate event.

Within the ServerValidate event, if the validation fails, you indicate this by setting the IsValid property of the passed-in ServerValidateEventArgs parameter to false.

In this example, you determine if the user has selected a Format by checking the `SelectedIndex` property of the `FormatList` control. When validating, you're not actually concerned with what the user has selected, just that she has selected something. So the validation fails if the user hasn't selected anything—if the `SelectedIndex` is equal to `-1`:

```
if (FormatList.SelectedIndex == -1)
{
  args.IsValid = false;
}
```

Although the validation check that you're performing here is quite simple, there are no limits to the validation that you can perform in the `ServerValidate` event. As long as you set the `IsValid` property to `false`, you can make the validation as complex and complete as you require.

The Page IsValid Check

The final change to the code for the page is to change the `SubmitButton_Click` handler to save the page only if the page was valid:

```
// only save if valid
if (Page.IsValid == true)
{
  // save player
}
```

Without this check, as you saw, the data will be saved to the database, even if one of the validators—in this case, the `CustomValidator`—failed. When committing changes to the database, you need to make sure that the submitted page is valid before actually saving the changes.

■ **Caution** Always check that the page `IsValid` before inserting or updating data in the database. Never rely on the fact that the client-side validators will prevent incorrect data from being transmitted. All validators run the validation routines, even if the check is also made client side, and this extra check will make sure that malicious users don't deliberately send false data to your page.

Deleting Data from the Database

After completing the previous examples in this chapter, you have a number of extra Players in the database. So, you'll want to know how to delete data from tables in a database.

When you remove data, you still need to follow the rules laid out by the database, but this time, you must consider how a database deals with deleting data:

- Unlike Windows, databases don't have a Recycle Bin. Once a user says delete some data, it's gone; the only way to get it back is to reinsert it.
- The foreign key constraints you set on your tables may cause the database to delete additional data from related tables or prevent you from deleting data from a table.

Because of these considerations, users need to be absolutely sure that they want to delete information before they actually do, and you want to warn them if any other data will be removed. In fact, you could even take it out of their hands and not give them the opportunity to delete data in the first place.

Another possibility is that you won't actually want to delete the data, but instead *pretend* to delete the data. For example, you might decide to no longer display a Player to the users, but you need to keep the details for historical purposes. By adding an extra Boolean column to the Player table called Deleted, you can hide data from the user. If it's false, the row is available to the user. If it's true (the user has "deleted" the Player), the data is not available.

In the example here, you're going to really delete the data so that you can see the `DELETE` query in action, and also see some of the issues related to relationships that deleting data can cause. If you were following the "pretend delete" route, you wouldn't actually execute a `DELETE` query, but would instead execute an `UPDATE` query. We'll look at the `UPDATE` query in the "Updating Data in the Database" section later in this chapter.

The DELETE Query

Like any other database operation, data deletion is handled by sending a SQL query to a database—in this case, a `DELETE` query. The `DELETE` query is quite simple:

```
DELETE [FROM] <table name>
[ WHERE <constraints> ]
```

The `DELETE` query has four parts:

- The keyword `DELETE` denotes the action to the database.
- The optional keyword `FROM` makes the query more readable.
- The *`table name`* identifies the table from which the data will be deleted.
- An optional list of *`constraints`* as a `WHERE` clause constrains the rows to which the `DELETE` query applies.

Like the `INSERT` query, `DELETE` can work on only one table at a time, which is probably a good thing. A rogue query such as `DELETE *` could wipe out all the data at once, if it were a valid query, a bit like `del *.*` would do in a DOS prompt. Indeed, `DELETE` works with whole rows only. You never delete single columns from a row. If you needed to remove a column from a row, you would change the column to an empty value or null, if the database allowed it.

Sympathy for the User: GridView ButtonField Columns

In the `INSERT` example, you used a collection of individual Web controls to let the user specify the column values for a new Player, and then displayed the new Player in a `GridView` as confirmation. You could carry on using simple Web controls in this exercise—perhaps binding Player names to a `DropDownList` and deleting the one selected in the list when a button is clicked—but you can easily code a more elegant solution, which you'll look at here.

The `GridView` can display much more than just the results of a `SELECT` query. In fact, to make it more interactive, you can add columns of buttons and links to it, allowing you to work with a row of data in the grid, given the button that was clicked. In this special case, you'll use

a button to indicate that a row should be deleted from the database. Depending on the purpose of your page, the button could signify that the row should be added to a shopping cart, copied to another location, or selected to have an e-mail message sent to it.

The object that enables you to do all this is the `<asp:ButtonField>` object, which you add to the `GridView`'s `Columns` collection, like so:

```
<asp:GridView id="GridView1" runat="server">
  <Columns>
    <asp:ButtonField Text="Delete" ButtonType="Button"
     CommandName="DeletePlayer" />
  </Columns>
</asp:GridView>
```

When you `DataBind()` to the `DataGrid`, any auto-generated columns will appear as usual, but there will now also be a column of buttons to the left displaying the value of the `ButtonField`'s `Text` property, as shown in Figure 8-12.

Figure 8-12. *The ButtonField as rendered in a GridView*

The `<asp:ButtonField>` object has two other key properties that should be given values: `ButtonType` and `CommandName`. `ButtonType` lets you specify whether the new column contains actual buttons or hyperlink-like buttons. `CommandName` identifies the action associated with the button and ties into the event handler called when the button is clicked.

By default, when a button in a `ButtonField` is clicked, the `GridView` raises an event called `RowCommand`. Within this event handler, you're free to implement whatever code you require.

■ **Note** There is one exception to the event mechanism when using a `ButtonField`. Setting the `CommandName` to `Cancel`, `Delete`, `Edit`, `Insert`, `New`, `Page`, `Select`, `Sort`, or `Update` has a slightly different effect than simply raising the `RowCommand` event. For instance, setting `CommandName` to `Delete` will actually raise the `RowDeleting` event and, after deleting the row, the `RowDeleted` event. However, in all cases, the `RowCommand` event also fires, so be aware if you're using multiple button columns that you need to check the `CommandName` to ensure that you're running the correct code. If you can avoid it, don't use a predefined `CommandName`.

Try It Out: Deleting Players with DELETE

Now that you know how to select a row for deletion in a `GridView`, let's build a page that demonstrates the technique. In this example, you'll add a `ButtonField` that sends the user to a new page that confirms that the Player is to be deleted.

1. Open `Players.aspx`. In the Design view, set the `DataKeyNames` property of the `GridView` to PlayerID.

2. Switch to the Source view and add a `BoundField` to the `Columns` collection of the `GridView`:

```
<Columns>
  <asp:BoundField DataField="PlayerID" HeaderText="PlayerID" />
  <asp:BoundField DataField="PlayerName" HeaderText="Name" />
  <asp:BoundField DataField="PlayerCost" DataFormatString="{0:n}"
    HeaderText="Cost" />
  <asp:ButtonField Text="Delete" ButtonType="Button"
    CommandName="DeletePlayer" />
</Columns>
```

3. Switch to the Design view and add a `RowCommand` event to the `GridView`. Add the following code to the event handler:

```
protected void Grid1View_RowCommand(object sender,
  GridViewCommandEventArgs e)
{
  // get the PlayerID
  int intIndex = Convert.ToInt32(e.CommandArgument);
  string strPlayerID = Convert.ToString(GridView1.DataKeys[intIndex].Value);

  // perform the correct action
  if (e.CommandName == "DeletePlayer")
  {
    Response.Redirect("./Player_Delete.aspx?PlayerID=" + strPlayerID);
  }
}
```

4. Add a new Web Form to the Web site called `Player_Delete.aspx` and change the page title to **DELETE Player**.

5. Add a confirmation question to the page and two buttons, called `SubmitButton` and `ReturnButton`. You'll also need a `Label`, called `QueryResult`, to show the results from the query that was actually executed. You can see how the Web controls are laid out in Figure 8-13.

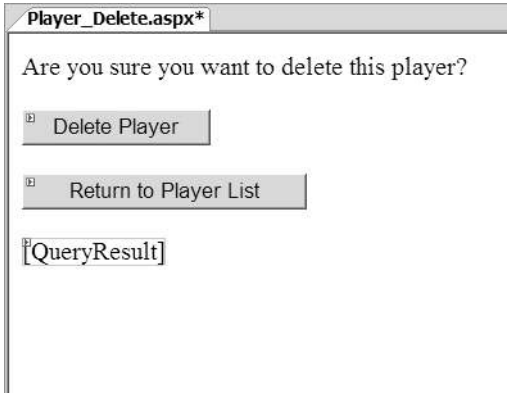

Figure 8-13. *You should always confirm deletions.*

6. Add a `Click` event for the `ReturnButton` control and add the following code to the event handler:

```
protected void ReturnButton_Click(object sender, EventArgs e)
{
  Response.Redirect("./Players.aspx");
}
```

7. Add the required namespace declaration to the top of the page:

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

8. Add a `Click` event for the `SubmitButton` and add the following code to the event handler:

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

```
    try
    {
      // query to execute
      string strQuery = "DELETE FROM WhatPlaysWhatFormat WHERE ➥
        WPWFPlayerID = @PlayerID; DELETE FROM Player ➥
        WHERE PlayerID = @PlayerID;";

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

      // add the parameter
      myCommand.Parameters.AddWithValue("@PlayerID",
        Request.QueryString["PlayerID"]);

      // open the connection
      myConnection.Open();

      // execute the query
      myCommand.ExecuteNonQuery();

      // show the result
      QueryResult.Text = "Delete of player '" +
        Request.QueryString["PlayerID"] + "' was successful";

      // disable the submit button
      SubmitButton.Enabled = false;
    }
    catch (Exception ex)
    {
      // show the error
      QueryResult.Text = "An error has occurred: " + ex.Message;
    }
    finally
    {
      // close the connection
      myConnection.Close();
    }
  }
```

9. Save both pages, and then start the Web site. Click the Delete button for a Player, and you'll be presented with the confirmation page. Clicking the Delete Player button will call the event handler to delete the Player and return a confirmation, as shown in Figure 8-14.

10. If you now click the Return to Player List button, you'll be able to confirm that the Player has indeed been deleted.

Figure 8-14. *Confirmation that the Player has been deleted*

How It Works

Adding button columns to a `GridView` isn't especially tricky. However, you should be aware of a couple of potential "gotchas" when you implement the event handler.

In this example, you're deleting only a row at a time, so the onus is on you to identify the row that has been selected for deletion using the primary key of the table and to relay that to the `DELETE` query. By using the primary key, you can ensure that only one row is deleted at a time. So, you discover the PlayerID for the row to be deleted and work with that. You append the PlayerID to the request for the Delete page as the `PlayerID` parameter:

```
Player_Delete.aspx?PlayerID=28
```

As you saw in Chapter 7, you can retrieve the ID of the row that you're dealing with from the `DataKeys` collection, as long as you've set the `DataKeyNames` property for the `GridView`. You can then extract the correct row from the collection:

```
int intIndex = Convert.ToInt32(e.CommandArgument);
string strPlayerID = Convert.ToString(GridView1.DataKeys[intIndex].Value);
```

Within `Player_Delete.aspx`, you first give the users the option of canceling the deletion by confirming that they wish to delete the Player. Remember that there is no Recycle Bin, so you should always confirm deletions before actually performing the `DELETE`. By clicking the Delete Player button, the `SubmitButton_Click` event handler is executed, and you can build the query to delete the Player from the database.

From the brief discussion of the `DELETE` query, you should be able to build a `DELETE` query to delete the Player relatively easily:

```
DELETE FROM Player WHERE PlayerID = @PlayerID
```

But if you look at the code, you'll see that this isn't the query that you execute. Now if you just ran this `DELETE` query by itself, you would get an error from the database saying that you

can't delete the Player row because there are rows in the WhatPlaysWhatFormat table that depend on this one, and it violates the relationship you set between the two when you built the database. The key then is to delete these dependent rows from the WhatPlaysWhatFormat table *before* you delete the row from the Player table. Fortunately, all you need to do for this is use the PlayerID again as the WPWFPlayerID column, so you can create the `DELETE FROM WhatPlaysWhatFormat` query, and then tack on the `DELETE FROM Player` query at the end:

```
DELETE FROM WhatPlaysWhatFormat WHERE WPWFPlayerID = @PlayerID;
DELETE FROM Book WHERE BookID = @PlayerID
```

The actual code is straightforward. You create a Connection object and a Command object containing the `DELETE` query. After adding the `@PlayerID` parameter, you call `ExecuteNonQuery()` on the Command object to execute the query.

Here are a couple of things to consider about this example:

Concurrency problems: You're executing two `DELETE` queries as a query batch, and if the second query fails (for whatever reason), the first query will still succeed. This causes quite a large concurrency problem, as you've deleted the supported Formats for the Player but not the actual details of the Player itself. What you need to do is ensure that either both queries succeed or both queries fail. You do this by using a transaction, as discussed in Chapter 12.

Deletions even after changes: What happens if changes are made by another user to the Player while you're confirming that you want to delete it? The other user's changes are made to the Player, and then you delete it! This may be what you want to happen, but it also may be incorrect. You may want to not allow the deletion if the data has changed. By creating a slightly more complex `DELETE` query, you can prevent any deletions from occurring if the Player has changed. We'll look at concurrency issues such as this in Chapter 12.

Checks for the number of rows deleted: You don't actually check that you're deleting the correct number of rows from the table. From the Web site's design, you know that if you delete a Player, you should affect only one row in the Player table. You could also check that this is true by looking at the number of rows affected returned from `ExecuteNonQuery()`. However, when executing a query batch, as you do here, the rows affected are the sum of all of the queries that are executed, so you couldn't use this value at the moment. You could split the two `DELETE` queries into separate calls to the database and ensure that the `DELETE` query against the Player table deleted only one row. If it didn't, you would have an error (and if you were using a transaction, you could then abort the transaction and prevent the deletion from occurring).

Deleting data is a relatively simple thing to do, but it's also the most final. It's worth repeating that you need to be careful when implementing deletions.

■ **Note** The code in the download for MySQL 5.0 and Microsoft Access both implement the `DELETE` functionality. However, as with the `INSERT` example, these databases can't handle multiple queries in the same query batch and don't support named parameters.

Updating Data in the Database

Suppose you want to allow users to edit data already in the database. You've designed the database to reduce the number of data-entry errors you may make, but that won't stop users from making mistakes, so you'll need to provide some way to correct them. Even if there aren't errors in your data, you still may need to update data. For example, an inventory system needs to keep updating the number of items in stock at any one time or the number of items sold; personalization systems need to update data when users update their preferences; and so on.

The process of editing data is much like adding it, except that you're starting the process with values that already exist. You still need to work with the rules of the database—the keys, the constraints, and so on. And you must try to use the best Web control for each column to allow users to edit the data. You may prefer to use a list Web control for foreign key values or a check box for Boolean values, rather than a text box.

The UPDATE Query

As usual, the whole operation of editing data comes down to generating and running a SQL query. In this case, it's an `UPDATE` query, which has the following syntax:

```
UPDATE < table name >
SET column1 name = expression1,
    column2 name = expression2,
       .
       .
       .
    columnM name = expressionM
[ WHERE <constraints> ]
```

The `UPDATE` query has the following five basic components:

- The keyword `UPDATE` denotes the action to the database.
- The *`table name`* determines the table from which the data will be updated.
- The keyword `SET` denotes the start of the updated information.
- A comma-separated list of assignments sets individual columns to given values.
- The `WHERE` clause constrains the number of rows that the `UPDATE` query affects.

`UPDATE` isn't limited to working with one table at a time, but it's probably easier to use it that way to start. Also, you may want to validate potential new values for the database before you update it, in the same fashion as when adding new rows.

Try It Out: Updating a Player with UPDATE

In this example, you'll add the final page to our solution so that Player details can be modified.

1. Open Players.aspx and add another <asp:ButtonField> to the GridView:

```
<asp:ButtonField Text="Edit" ButtonType="Button"
  CommandName="EditPlayer" />
```

2. Modify the RowCommand event of the GridView and add the following code:

```
protected void Grid1View_RowCommand(object sender,
  GridViewCommandEventArgs e)
{
  // get the PlayerID
  int intIndex = Convert.ToInt32(e.CommandArgument);
  string strPlayerID = Convert.ToString(
    GridView1.DataKeys[intIndex].Value);

  // perform the correct action
  if (e.CommandName == "DeletePlayer")
  {
    Response.Redirect("./Player_Delete.aspx?PlayerID=" + strPlayerID);
  }
  else if (e.CommandName == "EditPlayer")
  {
    Response.Redirect("./Player_Update.aspx?PlayerID=" + strPlayerID);
  }
}
```

3. In the Solution Explorer, copy Player_Insert.aspx and rename the copy Player_Update.aspx.

4. Change the page title to **UPDATE Player**, and change the Text property of the SubmitButton to **Update Player**.

5. Switch to the Source view and modify the Page_Load event as follows:

```
protected void Page_Load(object sender, EventArgs e)
{
  if (Page.IsPostBack == false)
  {
    // populate the list of manufacturers
    PopulateManufacturers();

    // populate the list of formats
    PopulateFormats();

    // retrieve existing player
    RetrieveExistingPlayer();
  }
}
```

6. Add the code for the `RetrieveExistingPlayer()` method:

```
private void RetrieveExistingPlayer()
{
  // create the connection
  string strConnectionString = ConfigurationManager.
    ConnectionStrings["SqlConnectionString"].ConnectionString;
  SqlConnection myConnection = new SqlConnection(strConnectionString);

  try
  {
    // create the first SELECT command
    string strQuery1 = "SELECT PlayerName, PlayerManufacturerID, ➥
      PlayerCost, PlayerStorage FROM Player WHERE PlayerID=@PlayerID;";
    SqlCommand myCommand1 = new SqlCommand(strQuery1, myConnection);
    myCommand1.Parameters.AddWithValue("@PlayerID",
      Request.QueryString["PlayerID"]);

    // create the first SELECT command
    string strQuery2 = "SELECT WPWFFormatID FROM WhatPlaysWhatFormat ➥
      WHERE WPWFPlayerID = @PlayerID;";
    SqlCommand myCommand2 = new SqlCommand(strQuery2, myConnection);
    myCommand2.Parameters.AddWithValue("@PlayerID",
      Request.QueryString["PlayerID"]);

    // open the connection
    myConnection.Open();

    // execute the first query
    SqlDataReader myReader1 = myCommand1.ExecuteReader();

    // if we have results, then we need to parse them
    if (myReader1.Read() == true)
    {
      PlayerName.Text = myReader1.GetString(
        myReader1.GetOrdinal("PlayerName"));
      ManufacturerList.SelectedValue = myReader1.GetInt32(
        myReader1.GetOrdinal("PlayerManufacturerID")).ToString();
      PlayerCost.Text = myReader1.GetDecimal(
        myReader1.GetOrdinal("PlayerCost")).ToString();
      PlayerStorage.Text = myReader1.GetString(
        myReader1.GetOrdinal("PlayerStorage"));
    }

    // close the first data reader
    myReader1.Close();
```

```
    // execute the second query
    SqlDataReader myReader2 = myCommand2.ExecuteReader();

    // if we have results, then we need to parse them
    while(myReader2.Read() == true)
    {
      foreach(ListItem objFormat in FormatList.Items)
      {
        if (objFormat.Value == myReader2.GetInt32(
          myReader2.GetOrdinal("WPWFFormatID")).ToString())
        {
          objFormat.Selected = true;
          break;
        }
      }
    }

    // close the second data reader
    myReader2.Close();
  }
  finally
  {
    // close the connection
    myConnection.Close();
  }
}
```

7. Change the code within the `SubmitButton_Click` event handler to the following:

```
protected void SubmitButton_Click(object sender, EventArgs e)
{
  // only save if valid
  if (Page.IsValid == true)
  {
    // save the player to the database
    bool blnPlayerError = SavePlayer();

    // did an error occur?
    if (blnPlayerError == true)
    {
      QueryResult.Text = "An error has occurred!";
    }
    else
    {
      // save the formats for the player
      bool blnFormatError = SaveFormats();
```

```
        // did an error occur?
        if (blnFormatError == true)
        {
          QueryResult.Text = "An error has occurred!";
        }
        else
        {
          // show the result
          QueryResult.Text = "Update of player '" +
            Request.QueryString["PlayerID"] + "' was successful";

          // disable the submit button
          SubmitButton.Enabled = false;
        }
      }
    }
  }
```

8. Modify the `SavePlayer()` method as follows:

```
private bool SavePlayer()
{
  bool blnError = false;

  // create the connection
  string strConnectionString = ConfigurationManager.
    ConnectionStrings["SqlConnectionString"].ConnectionString;
  SqlConnection myConnection = new SqlConnection(strConnectionString);

  try
  {
    // query to execute
    string strQuery = "UPDATE Player SET PlayerName = @Name, ➥
      PlayerManufacturerID = @ManufacturerID, PlayerCost = @Cost, ➥
      PlayerStorage = @Storage WHERE PlayerID = @PlayerID;";

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

    // add the parameters
    myCommand.Parameters.AddWithValue("@Name", PlayerName.Text);
    myCommand.Parameters.AddWithValue("@ManufacturerID",
      ManufacturerList.SelectedValue);
    myCommand.Parameters.AddWithValue("@Cost", PlayerCost.Text);
    myCommand.Parameters.AddWithValue("@Storage", PlayerStorage.Text);
    myCommand.Parameters.AddWithValue("@PlayerID",
      Request.QueryString["PlayerID"]);
```

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

    // execute the query
    myCommand.ExecuteNonQuery();
  }
  catch
  {
    // indicate that we have an error
    blnError = true;
  }
  finally
  {
    // close the connection
    myConnection.Close();
  }

  // return the error flag
  return (blnError);
}
```

9. Modify the `SaveFormats()` method as follows:

```
private bool SaveFormats()
{
  bool blnError = false;

  // create the connection
  string strConnectionString = ConfigurationManager.
    ConnectionStrings["SqlConnectionString"].ConnectionString;
  SqlConnection myConnection = new SqlConnection(strConnectionString);

  try
  {
    // create the DELETE query
    string strQuery1 = "DELETE FROM WhatPlaysWhatFormat WHERE ➥
      WPWFPlayerID = @PlayerID;";
    SqlCommand myCommand1 = new SqlCommand(strQuery1, myConnection);
    myCommand1.Parameters.AddWithValue("@PlayerID",
      Request.QueryString["PlayerID"]);

    // create the INSERT query
    string strQuery2 = "INSERT WhatPlaysWhatFormat ➥
      (WPWFPlayerID, WPWFFormatID) VALUES (@PlayerID, @FormatID)";
    SqlCommand myCommand2 = new SqlCommand(strQuery2, myConnection);
    myCommand2.Parameters.AddWithValue("@PlayerID",
      Request.QueryString["PlayerID"]);
    myCommand2.Parameters.Add("@FormatID", System.Data.SqlDbType.Int);
```

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

        // execute the DELETE query
        myCommand1.ExecuteNonQuery();

        // loop through each of the formats
        foreach (ListItem objFormat in FormatList.Items)
        {
          // save if selected
          if (objFormat.Selected == true)
          {
            // set the parameter value
            myCommand2.Parameters["@FormatID"].Value = objFormat.Value;

            // execute the INSERT query
            myCommand2.ExecuteNonQuery();
          }
        }
      }
      catch
      {
        // indicate that we have an error
        blnError = true;
      }
      finally
      {
        // close the connection
        myConnection.Close();
      }

      // return the error flag
      return (blnError);
    }
```

10. Save the two pages, and then start the Web site. If you edit a Player, you'll see that the modified details are saved to the database. Also notice that the validation routines prevent incorrect data from being entered.

How It Works

This example has more code than you've seen in the previous examples, but it just builds on those examples.

When editing data in the database, two tasks are paramount:

- Show the data that is already in the database so that the user can make the necessary changes.
- Save the data to the database, making sure that the modified data is saved and any old results are removed.

Due to the way that the Insert page is designed, you can add the ability to modify an existing Player by adding a new method, `RetrieveExistingPlayer()`, to populate the Web controls on the page with the existing details for the Player. Then you modify the `SavePlayer()` and `SaveFormats()` methods to handle updating the existing data, rather than adding new data.

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

■ **Note** In most cases, you'll have only one page that is used for both adding and updating data. You'll see this when we look at the `DataSet` examples later in this chapter. However, you should be able to combine the `Player_Insert.aspx`, `Player_Update.aspx`, and `Player_Delete.aspx` pages together to have one page in the same way, as you will for the `DataSet` examples.

Retrieving the Existing Player

The `RetrieveExistingPlayer()`method is responsible for retrieving the existing details for the Player from the database and showing these details on the page. In order to retrieve the Player from the database, you need to execute two queries, because the required data is stored in two different tables. The first query returns all of the information from the Player table:

```
SELECT PlayerName, PlayerManufacturerID, PlayerCost, PlayerStorage
FROM Player
WHERE PlayerID = @PlayerID
```

The `@PlayerID` parameter is set to the value retrieved from the query string using the `AddWithValue()` method of the `Parameters` collection:

```
myCommand1.Parameters.AddWithValue("@PlayerID",
  Request.QueryString["PlayerID"]);
```

To populate the Web controls on the page, you return the results of the query as a DataReader and attempt to fill the Web controls only if you've returned results. You can check this by using the `Read()` method, which returns `true` if there is a row of data:

```
if (myReader1.Read() == true)
{
  PlayerName.Text = myReader1.GetString(
    myReader1.GetOrdinal("PlayerName"));
  ManufacturerList.SelectedValue = myReader1.GetInt32(
    myReader1.GetOrdinal("PlayerManufacturerID")).ToString();
  PlayerCost.Text = myReader1.GetDecimal(
    myReader1.GetOrdinal("PlayerCost")).ToString();
  PlayerStorage.Text = myReader1.GetString(
    myReader1.GetOrdinal("PlayerStorage"));
}
```

You use the `Get`*XXX* methods of the DataReader to return the required column in the correct format. For PlayerName, PlayerCost, and PlayerStorage, you use text boxes to enter the data, so you can set the `Text` property of the `TextBox` to the string version of the column.

The PlayerManufacturerID column is a little trickier. You're showing the list of Manufacturers in a DropDownList, so you need to set the SelectedValue property. The only caveat for setting this property is that you need to have populated the Web control before you can set the SelectedValue (you'll get a runtime error if you don't, as the value that you're trying to select won't be one of the values that the Web control is displaying). If you take a look at the Page_Load event handler, you'll see that PopulateManufacturers() is called before you retrieve the Player from the database.

The Page_Load event also calls PopulateFormats() to populate the CheckBoxList with all of the available Formats before you retrieve the Player. That should give you a big hint as to the second query that you need to execute.

The second query retrieves the supported Formats for the Player from the database. All you require is the list of FormatID values for the required Player:

```
SELECT WPWFFormatID
FROM WhatPlaysWhatFormat
WHERE WPWFPlayerID = @PlayerID
```

Again, the @PlayerID parameter is set directly from the query string value and you retrieve the results as a DataReader. However you're returning multiple results into a list Web control that supports the selection of multiple items, so you can't simply set the SelectedValue for the Web control as you did for the ManufacturerList control. You need to step through each of the returned values and set the individual items within the CheckBoxList. You step through all of the returned results in the DataReader using the Read() method in conjunction with a while loop:

```
while(myReader2.Read() == true)
{
```

Then you loop through all of the ListItem entries in the CheckBoxList and set the Selected property of the ListItem to true if the Value of the ListItem is the FormatID from the DataReader:

```
  foreach(ListItem objFormat in FormatList.Items)
  {
    if (objFormat.Value == myReader2.GetInt32(
      myReader2.GetOrdinal("WPWFFormatID")).ToString())
    {
      objFormat.Selected = true;
      break;
    }
  }
}
```

If you select the ListItem, you'll see that you use break to exit from the foreach loop immediately. You're looping through every entry in the CheckBoxList for each Format that you retrieve from the database, and using break allows you to stop executing when you're never going to select another ListItem.

This is quite a convoluted way of setting the entries in the CheckBoxList. Because you're allowing the reader to select multiple values from the list, you must manually select the individual

items. You don't have the luxury of using the `SelectedValue` property, as you do for the `ManufacturerList` control.

Saving the Updated Player

In order to save the modified Player details to the database, you need to execute an `UPDATE` query to modify the existing Player in the database:

```
UPDATE Player
SET PlayerName = @Name, PlayerManufacturerID = @ManufacturerID,
  PlayerCost = @Cost, PlayerStorage = @Storage
WHERE PlayerID = @PlayerID
```

You're updating the Player table, and by adding the `WHERE` clause, you're constraining the `UPDATE` query to work only against the PlayerID that you require (which again takes its value from the query string).

The `SET` part of the query specifies the actual updating. You're updating the PlayerName, PlayerManufacturerID, PlayerCost, and PlayerStorage columns with the values from the `@Name`, `@ManufacturerID`, `@Cost`, and `@Storage` parameters.

You already know how to add parameters to the query, and you're simply specifying the reverse of the Web control population in the `RetrieveExistingPlayer()` method. You add the four parameters that take their values from the Web controls on the page:

```
myCommand.Parameters.AddWithValue("@Name", PlayerName.Text);
myCommand.Parameters.AddWithValue("@ManufacturerID",
  ManufacturerList.SelectedValue);
myCommand.Parameters.AddWithValue("@Cost", PlayerCost.Text);
myCommand.Parameters.AddWithValue("@Storage", PlayerStorage.Text);
```

And you also add the `@PlayerID` parameter from the query string value:

```
myCommand.Parameters.AddWithValue("@PlayerID",
  Request.QueryString["PlayerID"]);
```

Once you've added the five parameters to the query, you open the connection to the database and execute the query using the `ExecuteNonQuery()` method. If an error occurs, you trap it and set the error flag, causing any further processing of the update to be halted.

One thing you're not doing is checking that the `UPDATE` query has worked. You could use the return value from this method to determine the number of rows that were updated, and check that to determine if the update was a success—if you haven't updated a single row, then something has gone wrong. And if, in this instance, you've updated more than one row, then something has gone more wrong.

You're also not taking into account any changes that may be made by another user. What happens if changes are made by another user to the Player while you're making your changes? Whoever updates the Player first will lose those changes, and only the latest changes will be stored in the database. This may be what you want to happen, but it also may be incorrect. You may not want to allow the second update if the data has changed. By creating a slightly more complex `UPDATE` query, you can prevent any changes from occurring if the Player has changed. We'll look at this in Chapter 12.

Saving the Modified Formats

In order to save the supported Formats for the Player, you can't simply save the details of the selected Formats directly to the database. You need to insert multiple rows of information into the WhatPlaysWhatFormat table, and you can't do this using an `UPDATE` query. You saw how to add the Format details to the database using an `INSERT` query earlier, when you added a new Player. You use the same `INSERT` query to save the Formats to the database, but here, you can't simply add new information to the database; if you did, you might encounter the following problems:

- You add a Format that is already in the database, and you get a primary key error, as you're trying to add the same primary key (the combination of PlayerID and FormatID) into the WhatPlaysWhatFormat table.
- You manage to add the new information into the database (if you don't have a Format selected that was retrieved from the database initially), but now have incorrect data, as the Player will appear to support the original list of Formats as well as the new list of Formats that you saved.

To avoid these problems, you must remove the existing information from the database before you can add the new information. You execute a `DELETE` query to remove the existing Format details for the Player:

```
DELETE FROM WhatPlaysWhatFormat WHERE WPWFPlayerID = @PlayerID
```

Once the existing Format information has been saved to the database, you can then add the new information into the database. If you compare the `SaveFormat()` method in this page with the method you saw earlier in `Player_Insert.aspx`, you'll see that the method for adding the new information to the database is indeed the same.

Using a DataSet to Make the Changes

We've spent quite a considerable amount of time looking at how to insert, update, and delete data from the database by calling `ExecuteNonQuery()` and `ExecuteScalar()` on the Command object to propagate the changes to the database. Now we'll explore how to use a `DataSet` to make changes to the database. In the next chapter, you'll learn yet another way to edit data in a database: using the `SqlDataSource` with the `GridView`, `DetailsView`, and `FormView`.

As you learned in Chapter 5, the `DataSet` is disconnected from the database and allows you to make changes to the database and then propagate those changes back to the database in one go.

The key to making everything work is the DataAdapter (or to be provider-specific, the `SqlDataAdapter`, `OdbcDataAdapter`, and `OleDbDataAdapter`) being used in conjunction with `INSERT`, `UPDATE`, and `DELETE` queries. When using the DataAdapter to retrieve data, you set the `SelectCommand` to a Command object containing a `SELECT` query. You can do the same for `INSERT`, `UPDATE`, and `DELETE` queries by setting the `InsertCommand`, `UpdateCommand`, and `DeleteCommand` to the Command objects that you want to execute. You also have the option of letting a CommandBuilder (the `SqlCommandBuilder`, `OdbcCommandBuilder`, or `OleDbCommandBuilder`) automatically generate the `INSERT`, `UPDATE`, and `DELETE` queries based on the `SELECT` query that you use to populate the `DataTable`.

But how do you actually perform the changes? As you know, the DataAdapter has a `Fill()` method that populates a `DataTable` with the results of the `SelectCommand`. It also has a corresponding `Update()` method that propagates any changes made (either to the specified `DataTable` or to the `DataSet` as a whole) back to the database.

We'll start by looking at how the DataAdapter knows what changes need to be propagated to the database, and then see how the `DataRow` stores the changes that are made to its columns. Once we've finished this, admittedly brief, tour of the DataAdapter and `DataRow`, you'll build an example that allows you to insert, update, and delete Manufacturers.

■ **Note** This section provides only an introduction to what is possible using a `DataSet` to modify the database. For more information, see *Professional ADO.NET 2.0* by Sahil Malik (1-59059-512-2; Apress, 2005).

The Role of the DataAdapter

To make changes to the database, the DataAdapter needs to know what has actually changed so that it knows which of the SQL queries it needs to execute. It does this by looking at the `RowState` for each `DataRow` in the `DataTable`.

The `RowState` is set to a value from the `System.Data.DataRowState` enumeration. Table 8-3 shows the possible values for the `DataRowState` enumeration, as well as the Command object that will be used during the call to `Update()`.

Table 8-3. *The DataRowState Enumeration Values*

Name	Description	Command
`Added`	The `DataRow` has been added to the `DataTable`.	`InsertCommand`
`Deleted`	The `DataRow` has been deleted by calling the `Delete()` method of the `DataRow`.	`DeleteCommand`
`Detached`	A `DataRow` is detached when it is first created and before it is added to a `DataTable` (at which point, its state changes to `Added`).	N/A
`Modified`	The `DataRow` has been modified.	`UpdateCommand`
`Unchanged`	The `DataRow` hasn't been modified.	None

On calling the `Update()` method, each `DataRow` is interrogated for its `RowState`, and the necessary Command object used to propagate the changes to the database. After the changes have been propagated to the database, the `RowState` is set to `Unchanged` for added and modified rows, and any deleted rows are actually removed from the `DataTable`.

■ **Note** It is possible to turn off the modifications to the `DataTable` during an `Update()` by setting the `AcceptChangesDuringUpdate` property to `false` (it has a default of `true`). If you do this, the changes will have been made to the database, but the `DataTable` will still show that it has rows that have been changed. In order to accept the changes, you would need to call the `AcceptChanges()` method to complete the update of the `DataTable`. You can also call `AcceptChanges()` on an individual `DataRow` to commit any changes to that row, without affecting any other rows in the table.

The Role of the DataRow

As you've just learned, the `DataRow` knows that it has changed, and its `RowState` property tells you what change has been made. The `DataRow` also keeps a track of the value that it originally contained, as well as the value that it currently contains.

You can retrieve the value of a column from a `DataRow` by specifying either the column's index or its name. For example, to retrieve the ManufacturerName from a `DataRow`, you would use the following:

```
ManufacturerName.Text = drManufacturer["ManufacturerName"].ToString();
```

It is also possible to retrieve the value that the column originally contained:

```
ManufacturerName.Text =
  drManufacturer["ManufacturerName",DataRowVersion.Original].ToString();
```

Along with these two versions of the data, you can also retrieve two other versions. Table 8-4 shows the four versions you can retrieve.

Table 8-4. *The DataRowVersion Enumeration Values*

Name	Description
`Current`	The default, which returns the value that will be propagated to the database or returned when you don't specify a `DataRowVersion`. When the `DataRow` is created the `Current` and `Original` values are the same. When the column is modified the `Current` value will change.
`Default`	Returns either the `Current` version (if the `RowState` is `Added`, `Modified`, or `Deleted`) or the `Proposed` version (if the `RowState` is `Detached`).
`Original`	Returns the value that was retrieved from the database. You can use the `Original` version to check that the row you're updating or deleting hasn't changed before setting its value to the `Current` version (see Chapter 12).
`Proposed`	Returns the value that is proposed for the column.

Try It Out: Inserting Data Using a DataSet

In this example, you'll add a page that allows you to add Manufacturers in the database. You'll first create a page to view the Manufacturers that are in the database.

1. Add a new Web Form to the Web site called `Manufacturers.aspx`. Make sure that the Place Code in Separate File check box is unchecked.

2. In the Source view, find the `<title>` tag within the HTML at the bottom of the page and change the page title to **Manufacturers**.

3. Switch to the Design view and add a `SqlDataSource` to the page. Choose to configure the data source and use `SqlConnectionString` to connect to the database. Select the ManufacturerID, ManufacturerName, and ManufacturerCountry columns from the Manufacturer table to configure the `SELECT` query.

4. Switch back to the Source view and add the following markup after the declaration of the `SqlDataSource`:

```
<asp:HyperLink ID="HyperLink1" runat="server"
  NavigateUrl="./Manufacturer_Edit.aspx">Add manufacturer</asp:HyperLink>
<br /><br />
<asp:GridView ID="GridView1" runat="server" AutoGenerateColumns="False"
  DataSourceID="SqlDataSource1">
  <Columns>
    <asp:BoundField DataField="ManufacturerID"
      HeaderText="ManufacturerID" />
    <asp:BoundField DataField="ManufacturerName"
      HeaderText="ManufacturerName" />
    <asp:BoundField DataField="ManufacturerCountry"
      HeaderText="ManufacturerCountry" />
  </Columns>
</asp:GridView>
```

5. Add a new Web Form to the Web site called `Manufacturer_Edit.aspx`. Make sure that the Place Code in Separate File check box is unchecked.

6. In the Source view, find the `<title>` tag within the HTML at the bottom of the page and change the page title to **Edit Manufacturer**. Then add the required `Import` statements to the top of the page:

```
<%@ Page Language="C#" %>
<%@ Import Namespace="System.Data" %>
<%@ Import Namespace="System.Data.SqlClient" %>
```

7. Add the Web controls to allow the addition of a Manufacturer to the database. Add a `Button` to save the Manufacturer; a `Button` to return to the list of Manufacturers; and four `TextBox` controls for the user to add the Manufacturer's name, country, e-mail address, and Web site address. Name these `SaveButton`, `ReturnButton`, `ManufacturerName`, `ManufacturerCountry`, `ManufacturerEmail`, and `ManufacturerWebsite`, respectively. Also add a `Label`, called `QueryResult`, to show the results from the query that was executed. You can see how the Web controls are laid out in Figure 8-15.

Figure 8-15. *The Web control layout for Manufacturer_Edit.aspx*

8. Switch to the Source view and add the following variable declarations to the top of the `<script>` block:

```
SqlDataAdapter myAdapter;
DataSet myDataSet;
```

9. Add the private method to retrieve the Manufacturers from the database:

```
private void RetrieveManufacturers()
{
  // set the SQL query we need to get the manufacturers
  string strQuery = "SELECT ManufacturerID, ManufacturerName, ➥
    ManufacturerCountry, ManufacturerEmail, ManufacturerWebsite ➥
    FROM Manufacturer";

  // create the Connection to the database
  string ConnectionString = ConfigurationManager.
    ConnectionStrings["SqlConnectionString"].ConnectionString;
  SqlConnection myConnection = new SqlConnection(ConnectionString);
```

```
  // create the DataAdapter
  myAdapter = new SqlDataAdapter(strQuery, myConnection);

  // set up the INSERT/UPDATE/DELETE queries
  SqlCommandBuilder myCommandBuilder = new SqlCommandBuilder(myAdapter);

  // create a new DataSet
  myDataSet = new DataSet();

  // fill the DataSet
  myAdapter.Fill(myDataSet, "Manufacturer");

  // now add the primary key details
  DataColumn[] myPrimaryKey = {
    myDataSet.Tables["Manufacturer"].Columns["ManufacturerID"] };
  myDataSet.Tables["Manufacturer"].PrimaryKey = myPrimaryKey;
}
```

10. Switch to the Design view and double-click the Save Player button to add the `Click` event handler. Add the following code:

```
protected void SaveButton_Click(object sender, EventArgs e)
{
  // only save if valid
  if (Page.IsValid == true)
  {
    // get the Manufacturers
    RetrieveManufacturers();

    // create a new DataRow
    DataRow drManufacturer = myDataSet.Tables["Manufacturer"].NewRow();

    // now set the column values
    drManufacturer["ManufacturerName"] = ManufacturerName.Text;
    drManufacturer["ManufacturerCountry"] = ManufacturerCountry.Text;
    drManufacturer["ManufacturerEmail"] = ManufacturerEmail.Text;
    drManufacturer["ManufacturerWebsite"] = ManufacturerWebsite.Text;
```

```
        // add a temporary primary key value
        drManufacturer["ManufacturerID"] = "-1";

        // add the DataRow to the table
        myDataSet.Tables["Manufacturer"].Rows.Add(drManufacturer);

        try
        {
          // now update the database
          myAdapter.Update(myDataSet, "Manufacturer");

          // show the result
          QueryResult.Text = "Save of manufacturer was successful";

          // disable all the controls we don't want to allow changes to
          SaveButton.Enabled = false;
          ManufacturerName.Enabled = false;
          ManufacturerCountry.Enabled = false;
          ManufacturerEmail.Enabled = false;
          ManufacturerWebsite.Enabled = false;
        }
        catch (Exception ex)
        {
          // show the error
          QueryResult.Text = "An error has occurred: " + ex.Message;
        }
      }
    }
```

11. Switch back to the Design view and add the `Click` event handler for the Return to Manufacturer List button. Add the following code:

```
protected void ReturnButton_Click(object sender, EventArgs e)
{
  Response.Redirect("./Manufacturers.aspx");
}
```

12. Save both pages, and then open `Manufacturers.aspx` in your browser. Click the Add Manufacturer link, and then enter the details for a new Manufacturer on the `Manufacturer_Edit.aspx` page.

13. Click the Save Manufacturer button to the database, and you'll see that the Manufacturer has been saved, as shown in Figure 8-16.

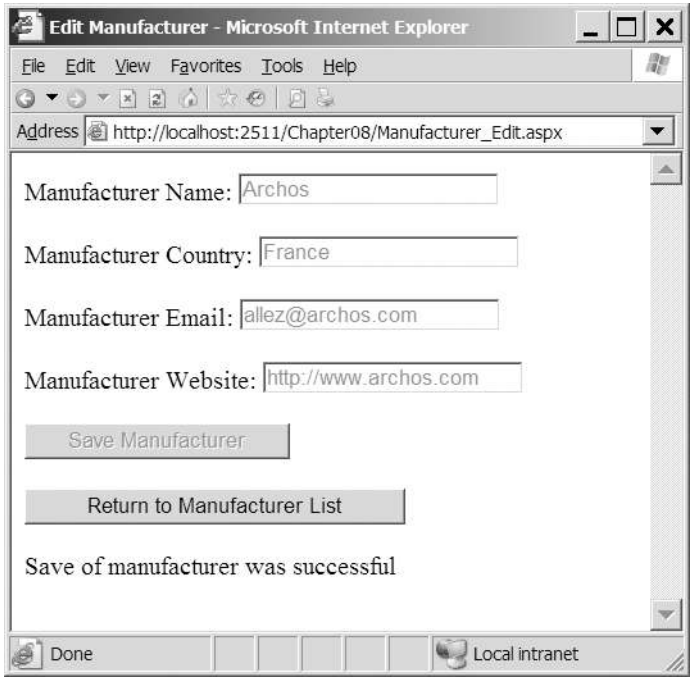

Figure 8-16. *Adding a new Manufacturer to the database*

How It Works

As you can see, using a DataSet to add to the database requires substantially less code than the equivalent page for the earlier Player example.

The addition of a new Manufacturer to the database requires three basic steps:

- Retrieve the existing Manufacturers from the database into a `DataTable` within the `DataSet`.
- Add a new `DataRow` to the `DataTable` and populate it correctly with the information entered by the user.
- Save the changes made by to the `DataTable` to the database using the DataAdapter.

We'll look at each of these steps in turn.

Retrieving the Manufacturers

Before you can add a new Manufacturer to the `DataTable`, you need to know the structure of the `DataTable`. As you saw in Chapter 5, you can build the `DataTable` from scratch, populate it, and send the changes back to the database. However, you didn't do that here. Instead, you retrieve the existing Manufacturer table from the database to ensure that you have the correct format.

You retrieve the list of Manufacturers using the `RetrieveManufacturers()` method, which you call before you attempt to add a new row to the `DataTable`. It may seem a little strange that you're doing this in its own method and storing the `DataSet` and DataAdapter as global variables

for the page. You use this approach because it allows you to use the same code to retrieve the list of Manufacturers for updating and deleting data as well.

The first thing that you do within the `RetrieveManufacturers()` method is to create the SQL for the `SELECT` query and create the Connection object.

```
// set the SQL query we need to get the manufacturers
string strQuery = "SELECT ManufacturerID, ManufacturerName, ➥
  ManufacturerCountry, ManufacturerEmail, ManufacturerWebsite ➥
  FROM Manufacturer";

// create the Connection to the database
string ConnectionString = ConfigurationManager.
  ConnectionStrings["SqlConnectionString"].ConnectionString;
myConnection = new SqlConnection(ConnectionString);
```

You then create the DataAdapter that you're going to use to propagate any changes back to the database, passing in the `SELECT` query and the Connection that you want to use:

```
// create the DataAdapter
myAdapter = new SqlDataAdapter(strQuery, myConnection);
```

Once you have a DataAdapter object, you can then use a CommandBuilder to automatically create the `INSERT`, `UPDATE`, and `DELETE` queries. All you need to do is create a new instance of the CommandBuilder, passing the DataAdapter to the constructor. You don't even need to keep a reference to the CommandBuilder after it has been created:

```
// set up the INSERT/UPDATE/DELETE queries
SqlCommandBuilder myCommandBuilder = new SqlCommandBuilder(myAdapter);
```

You then create a new `DataSet` and use the `Fill()` method of the DataAdapter to fill the correct table:

```
// create a new DataSet
myDataSet = new DataSet();

// fill the DataSet
myAdapter.Fill(myDataSet, "Manufacturer");
```

Next, you need to add the primary key details for the `DataTable`. You create a new array of `DataColumn` objects and add the required columns—in this case, ManufacturerID—to the array. You can then set the `PrimaryKey` property on the `DataTable`:

```
// now add the primary key details
DataColumn[] myPrimaryKey = {
  myDataSet.Tables["Manufacturer"].Columns["ManufacturerID"] };
myDataSet.Tables["Manufacturer"].PrimaryKey = myPrimaryKey;
```

You don't manually open or close the connection to the database within the `RetrieveManufacturers()` method. Instead, you let the DataAdapter open and close the connection as it requires. It will do this when it needs to `Fill()` or `Update()` the `DataSet`, and will open the connection only when it is required.

Adding a New Row to the Table

We've already looked at how to add a new row to a DataTable in Chapter 5. The code that you execute here is the same apart, from one little caveat.

You first call the NewRow() method on the Manufacturer DataTable to return a new DataRow object (which has its RowState set to Detached):

```
// create a new DataRow
DataRow drManufacturer = myDataSet.Tables["Manufacturer"].NewRow();
```

You can then set the values of the four columns directly from the TextBox controls from the page:

```
// now set the column values
drManufacturer["ManufacturerName"] = ManufacturerName.Text;
drManufacturer["ManufacturerCountry"] = ManufacturerCountry.Text;
drManufacturer["ManufacturerEmail"] = ManufacturerEmail.Text;
drManufacturer["ManufacturerWebsite"] = ManufacturerWebsite.Text;
```

Then you need to add a ManufacturerID to the DataRow, even though it's a primary key and an auto-generated column in the database. By setting the primary key on the DataTable, you're no longer allowed to have a null value for the ManufacturerID column, so you choose a value that cannot appear in the database:

```
// add a temporary primary key value
drManufacturer["ManufacturerID"] = "-1";
```

By using the value of -1, you don't risk picking a value that is (or could be) a ManufacturerID value in the database. This value is never sent to the database and is ignored, as the INSERT query that is generated doesn't pass it to the database.

You can then add the DataRow to the DataTable, changing its RowState to Added in the process:

```
// add the DataRow to the table
myDataSet.Tables["Manufacturer"].Rows.Add(drManufacturer);
```

Now that you've added the DataRow to the DataTable, you can propagate the changes to the database.

Saving the Changes to the Database

You propagate the changes to the database by calling the Update() method of the DataAdapter specifying the DataSet and the name of the DataTable you want to update. Then you inform the user that the save was successful:

```
// now update the database
myAdapter.Update(myDataSet, "Manufacturer");

// show the result
QueryResult.Text = "Save of manufacturer was successful";
```

Although the code looks simple, quite a lot of work is going on under the covers. The DataAdapter loops through all of the DataRows in the DataTable and checks the RowStatus. If the

status is Added, Modified, or Deleted, the changes are propagated back to the database using the InsertCommand, UpdateCommand, or DeleteCommand, respectively.

As you've made only one change the DataTable, you have only one change to propagate to the database. As it's a DataRow addition, the InsertCommand will be used.

The CommandBuilder created the InsertCommand automatically, and as you saw in the example, it works. It's not all magic though, and it is possible to view the query that is being executed by tapping into the events that the DataAdapter raises.

As in common with most of the other data-access Web controls and classes in ASP.NET, the DataAdapter has before and after events concerned with updating the database:

- RowUpdating: This event occurs before the update takes place. The event arguments allow you to inspect the query that is being executed.
- RowUpdated: Once the update has taken place, the RowUpdated event allows you to see the query that has been executed, as well see the number of RecordsAffected by the update.

These two events are common to each type of update (INSERT, UPDATE, or DELETE).

If you wanted to, you could add the event handler. Visual Web Developer makes it very easy to do this, and it will auto-complete most of the code to add the event handler (and will even add the signature for the event handler itself if you press Tab when asked). You can add the following to add the event handler:

```
myAdapter.RowUpdating += new
  SqlRowUpdatingEventHandler(myAdapter_RowUpdating);
```

And then do whatever you want within the event handler itself.

If you look at the query that is executed (e.Command.CommandText), you'll see that it is a quite simple INSERT query:

```
INSERT INTO Manufacturer (ManufacturerName, ManufacturerCountry,
  ManufacturerEmail, ManufacturerWebsite)
VALUES (@p1, @p2, @p3, @p4)
```

The CommandBuilder has created an INSERT query simply from the definition of the SELECT query. The CommandBuilder has parameterized it as well, and if you look at the Parameters collection, you'll see that it picks up the values from the DataTable automatically using the SourceColumn and SourceVersion properties of the SqlParameter. You'll learn more about these properties a little later in this chapter, in the "Manually Creating the Commands" section.

Try It Out: Updating Data Using a DataSet

You'll now modify the example so that it also allows you to update an existing Manufacturer. In the earlier Player examples, a lot of the code was repeated in the INSERT and UPDATE examples. By combining the two pages, you can remove a lot of repeated code.

1. Open Manufacturers.aspx. In the Design view, set the DataKeyNames property of the GridView to ManufacturerID.
2. Switch to the Source view and add a ButtonField to the Columns collection of the GridView:

```
<Columns>
  <asp:BoundField DataField="ManufacturerID"
    HeaderText="ManufacturerID" />
  <asp:BoundField DataField="ManufacturerName"
    HeaderText="ManufacturerName" />
  <asp:BoundField DataField="ManufacturerCountry"
    HeaderText="ManufacturerCountry" />
  <asp:ButtonField Text="Edit" ButtonType="Button"
    CommandName="EditManufacturer" />
</Columns>
```

3. Add a `RowCommand` event to the `GridView`. Add the following code to the event handler:

```
protected void Grid1View_RowCommand(object sender,
  GridViewCommandEventArgs e)
{
  // get the ManufacturerID
  int intIndex = Convert.ToInt32(e.CommandArgument);
  string strManufacturerID = Convert.ToString(
    GridView1.DataKeys[intIndex].Value);

  // perform the correct action
  if (e.CommandName == "EditManufacturer")
  {
    Response.Redirect("./Manufacturer_Edit.aspx?ManufacturerID=" +
      strManufacturerID);
  }
}
```

4. Open `Manufacturers_Edit.aspx` and add a `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)
  {
    // only load if we have a manufacturer
    if (Request.QueryString["ManufacturerID"] != null)
    {
      // load all the manufacturers
      RetrieveManufacturers();

      // find the one we're after
      DataRow drManufacturer = myDataSet.Tables["Manufacturer"].
        Rows.Find(Request.QueryString["ManufacturerID"]);
```

```
        // set the four controls
        ManufacturerName.Text =
          drManufacturer["ManufacturerName"].ToString();
        ManufacturerCountry.Text =
          drManufacturer["ManufacturerCountry"].ToString();
        ManufacturerEmail.Text =
          drManufacturer["ManufacturerEmail"].ToString();
        ManufacturerWebsite.Text =
          drManufacturer["ManufacturerWebsite"].ToString();
      }
    }
}
```

5. Modify the SaveButton_Click event handler as follows:

```
protected void SaveButton_Click(object sender, EventArgs e)
{
  // only save if valid
  if (Page.IsValid == true)
  {
    // get the Manufacturers
    RetrieveManufacturers();

    // create new or use existing?
    DataRow drManufacturer = null;
    if (Request.QueryString["ManufacturerID"] == null)
    {
      // create a new DataRow
      drManufacturer = myDataSet.Tables["Manufacturer"].NewRow();
    }
    else
    {
      // find the one we're after
      drManufacturer = myDataSet.Tables["Manufacturer"].Rows.
        Find(Request.QueryString["ManufacturerID"]);
    }

    // now set the column values
    drManufacturer["ManufacturerName"] = ManufacturerName.Text;
    drManufacturer["ManufacturerCountry"] = ManufacturerCountry.Text;
    drManufacturer["ManufacturerEmail"] = ManufacturerEmail.Text;
    drManufacturer["ManufacturerWebsite"] = ManufacturerWebsite.Text;

    // if new, must add to table
    if (Request.QueryString["ManufacturerID"] == null)
    {
      // add a temporary primary key value
      drManufacturer["ManufacturerID"] = "-1";
```

```
        // add the DataRow to the table
        myDataSet.Tables["Manufacturer"].Rows.Add(drManufacturer);
      }

      try
      {
        // now update the database
        myAdapter.Update(myDataSet, "Manufacturer");

        // show the result
        QueryResult.Text = "Save of manufacturer was successful";

        // disable all the controls we don't want to allow changes to
        SaveButton.Enabled = false;
        ManufacturerName.Enabled = false;
        ManufacturerCountry.Enabled = false;
        ManufacturerEmail.Enabled = false;
        ManufacturerWebsite.Enabled = false;
      }
      catch (Exception ex)
      {
        // show the error
        QueryResult.Text = "An error has occurred: " + ex.Message;
      }
    }
  }
```

6. Save both pages, and then open `Manufacturer.aspx` in your browser. Clicking any of the Edit buttons in the `GridView` will allow you to modify the selected Manufacturer. You can confirm the changes have been made by looking at the complete list of Manufacturers.

How It Works

You've added the ability to edit Manufacturers with only minimal changes to the page. We'll look at each of these changes in turn.

Populating the Controls on Load

Now that you're editing an existing Manufacturer, rather than adding a new Manufacturer, you must retrieve the list of Manufacturers and then select the correct Manufacturer. This is accomplished quite easily by using the `Find()` method of the `Rows` collection for the `DataTable` and specifying the primary key for the row that you want to retrieve:

```
// find the one we're after
DataRow drManufacturer = myDataSet.Tables["Manufacturer"].
  Rows.Find(Request.QueryString["ManufacturerID"]);
```

This returns a `DataRow` that you can interrogate to retrieve the column values that you're after and set the `TextBox` controls correctly.

Editing a Row in the Table

When editing a row, you no longer need to add a new `DataRow` to the `DataTable`; you can use the existing row. Therefore, you check whether you're performing an insert or an update by checking to see if there is a ManufacturerID in the query string. If there isn't, you want to create a new row, as before:

```
if (Request.QueryString["ManufacturerID"] == null)
{
  // create a new DataRow
  drManufacturer = myDataSet.Tables["Manufacturer"].NewRow();
}
```

If the query string does have a ManufacturerID, you retrieve the existing `DataRow` using the primary key to find the correct ManufacturerID:

```
else
{
  // find the one we're after
  drManufacturer = myDataSet.Tables["Manufacturer"].Rows.
    Find(Request.QueryString["ManufacturerID"]);
}
```

You then set the column values based on the `TextBox` controls in exactly the same way as before.

Next, you check to see if you're adding or editing a Manufacturer. If you're adding one, you set the temporary primary key value and add the new `DataRow` to the `DataTable`.

Saving the Changes to the Database

The code to save the changes to the database is exactly the same as before. The `Update()` method commits all changes to the database and chooses the `UpdateCommand` in this instance.

If you look at the query in the `RowUpdating` event handler, you'll see that the `UPDATE` query is a lot more complex than you might expect:

```
UPDATE Manufacturer SET ManufacturerName = @p1 WHERE
((ManufacturerID = @p2) AND (ManufacturerName = @p3)
AND ((@p4 = 1 AND ManufacturerCountry IS NULL) OR
(ManufacturerCountry = @p5)) AND ((@p6 = 1 AND
ManufacturerEmail IS NULL) OR (ManufacturerEmail = @p7)) AND
((@p8 = 1 AND ManufacturerWebsite IS NULL)
OR (ManufacturerWebsite = @p9)))
```

This is the query generated when you change only the ManufacturerName. If you changed all four values (name, country, e-mail address, and Web site address), the query becomes even more complex, with twelve parameters rather than the nine here. The `UPDATE` query is generated on the fly and updates only the columns that have been modified, but does it really need to be this complex?

Although this `UPDATE` query works, it isn't as efficient as it could be. You'll see shortly that you can add your own queries, rather than using the CommandBuilder auto-generated ones.

Try It Out: Deleting Data Using a DataSet

You'll now add the final part of the page to delete Manufacturers from the database.

1. Open `Manufacturers_Edit.aspx`. In the Design View, add a new `Button` called `DeleteManufacturer` to the page beside the Save Player button. You can see how the Web controls are laid out in Figure 8-17.

Figure 8-17. *The new Web control layout for Manufacturer_Edit.aspx*

2. Double-click the Delete Manufacturer button to add the `Click` event, and add the following code to the event handler:

```
protected void DeleteButton_Click(object sender, EventArgs e)
{
  // load all the manufacturers
  RetrieveManufacturers();

  // find the one we're after
  DataRow drManufacturer = myDataSet.Tables["Manufacturer"].Rows.
    Find(Request.QueryString["ManufacturerID"]);
```

```
    // delete it
    drManufacturer.Delete();

    try
    {
      // now update the database
      myAdapter.Update(myDataSet, "Manufacturer");

      // show the result
      QueryResult.Text = "Delete of manufacturer was successful";

      // disable all the controls we don't want to allow changes to
      SaveButton.Enabled = false;
      DeleteButton.Enabled = false;
      ManufacturerName.Enabled = false;
      ManufacturerCountry.Enabled = false;
      ManufacturerEmail.Enabled = false;
      ManufacturerWebsite.Enabled = false;
    }
    catch (Exception ex)
    {
      // show the error
      QueryResult.Text = "An error has occurred: " + ex.Message;
    }
  }
```

3. Modify the `Page_Load` event handler to disable the Delete Manufacturer button if you're adding a new Manufacturer, as follows:

```
protected void Page_Load(object sender, EventArgs e)
{
  if (Page.IsPostBack == false)
  {
    // only load if we have a manufacturer
    if (Request.QueryString["ManufacturerID"] != null)
    {
      // load all the manufacturers
      RetrieveManufacturers();

      // find the one we're after
      DataRow drManufacturer = myDataSet.Tables["Manufacturer"].
        Rows.Find(Request.QueryString["ManufacturerID"]);

      // set the four controls
      ManufacturerName.Text =
        drManufacturer["ManufacturerName"].ToString();
      ManufacturerCountry.Text =
        drManufacturer["ManufacturerCountry"].ToString();
```

```
        ManufacturerEmail.Text =
          drManufacturer["ManufacturerEmail"].ToString();
        ManufacturerWebsite.Text =
          drManufacturer["ManufacturerWebsite"].ToString();
      }
      else
      {
        // we want to disable the delete button
        DeleteButton.Enabled = false;
      }
    }
  }
```

4. Modify the end of the `SaveButton_Click` event handler and add the code to disable the Delete Manufacturer button:

```
// disable all the controls we don't want to allow changes to
SaveButton.Enabled = false;
DeleteButton.Enabled = false;
ManufacturerName.Enabled = false;
ManufacturerCountry.Enabled = false;
ManufacturerEmail.Enabled = false;
ManufacturerWebsite.Enabled = false;
```

5. Save the page, and then open `Manufacturers.aspx` in your browser. If you click the Edit button for a Manufacturer, you'll see that you can now delete the Manufacturer.

How It Works

Once again, you see that deleting is the easiest thing to do. First, you retrieve the `DataRow` that you're after:

```
// find the one we're after
DataRow drManufacturer = myDataSet.Tables["Manufacturer"].Rows.
  Find(Request.QueryString["ManufacturerID"]);
```

And then you delete it:

```
// delete it
drManufacturer.Delete();
```

This changes the `RowState` to `Deleted`. When the `Update()` method is called, the `DeleteCommand` is used, and the following auto-generated query is executed:

```
DELETE FROM Manufacturer WHERE ((ManufacturerID = @p1)
AND (ManufacturerName = @p2) AND ((@p3 = 1
AND ManufacturerCountry IS NULL) OR (ManufacturerCountry = @p4))
AND ((@p5 = 1 AND ManufacturerEmail IS NULL) OR
(ManufacturerEmail = @p6)) AND ((@p7 = 1 AND
ManufacturerWebsite IS NULL) OR (ManufacturerWebsite = @p8)))
```

As with the UPDATE query, the DELETE query works, but is perhaps overly complex. You should be able to delete a Manufacturer using one parameter (the ManufacturerID), but this DELETE query has nine parameters! It's definitely time to look at creating your own queries.

Manually Creating the Commands

You've seen in the previous three examples that you can use a CommandBuilder to automatically create the INSERT, UPDATE, and DELETE queries. Although the INSERT query was quite acceptable, the UPDATE and DELETE queries are a little complex (to say the least!).

It is easy to create your own INSERT, UPDATE, and DELETE queries, rather than letting a CommandBuilder do it for you. As an example, let's look at how to add an UPDATE query.

■ **Caution** If you're not using a CommandBuilder and don't define a Command object for the operation that you're performing, you'll get an InvalidOperationException if you try to perform that operation.

You first need to define the Command object that you want to use (assuming that myConnection is already defined as a Connection object):

```
// query to execute
string strQuery = "UPDATE Manufacturer SET
  ManufacturerName = @ManufacturerName,
  ManufacturerCountry = @ManufacturerCountry,
  ManufacturerEmail = @ManufacturerEmail,
  ManufacturerWebsite = @ManufacturerWebsite,
  WHERE ManufacturerID = @ManufacturerID;";

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

You then need to add all of the parameters to the Command object. But how do you know what the values are? How do you query the DataTable to get the correct parameters? You need to use the SourceColumn property to determine the specific column from the DataTable and the SourceVersion property to specify which version of the column (Current or Original) you're after. So, the ManufacturerName parameter is created as follows:

```
SqlParameter myNameParameter = new SqlParameter();
myNameParameter.ParameterName = "@ManufactuereID";
myNameParameter.SourceColumn = "ManufacturerID";
myNameParameter.SourceVersion = DataRowVersion.Current;
```

And then added to the Parameters collection as follows:

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

The remaining four columns are created in the same way. You specify the `SourceColumn` as the name of the column and the `SourceVersion` to be the `Current` version.

■ **Note** When we look at concurrency in Chapter 12, you'll see that you sometimes use the `Original` version of a column to ensure that the data that you're changing hasn't changed between deciding to change the data and actually getting around to changing the data.

Creating your own `INSERT` and `UPDATE` queries follows the same pattern.

Summary

While a SQL query is at the center of every data operation—`SELECT`, `INSERT`, `UPDATE`, or `DELETE`—you have many ways to get that SQL query defined and executed, and its results examined. You can do almost anything you like as long as you form the SQL correctly and obey the rules of the database you've defined.

You shouldn't regard the examples in these chapters as the dogmatic way to do any one particular task. Their purpose is to present various techniques that you may or may not choose to use in your own pages. Whether you use any one block of code is up to you, but you do at least now know where some code works and where other code doesn't work.

In the next chapter, we'll move away from writing code to modify the database and see that the `GridView` (and its siblings the `DetailsView` and `FormView`) allows you to write pages that will automatically propagate the changes to the database, provided that you specify the correct `INSERT`, `UDPATE`, and `DELETE` queries.