



Stored Procedures

All of the SQL queries you've executed against the database in previous chapters have been contained within the page where the execution will apply. Although you saw that this works effectively with simple SELECT, INSERT, UPDATE, and DELETE queries, formulating a query this way isn't always a suitable solution for more complex requirements.

Stored procedures provide another solution for executing queries against a database. Put simply, a stored procedure is a construct to store queries on the server, so the same query is available for any application that wants to use it.

SQL Server 2005 and MySQL 5.0 support stored procedures, and we'll look at both of them in this chapter. Microsoft Access has its own form of stored procedures, which are called, confusingly, *queries*. We won't look at Microsoft Access queries, because you need a copy of Microsoft Access to create them; however, if you do have a copy of Microsoft Access, you'll see that it does have quite a good query designer that bears a resemblance to the Visual Web Developer Query Builder.

This chapter covers the following topics:

- Advantages of using stored procedures
- How to create stored procedures and give users access to them
- How to execute stored procedures through a Command object
- How to alter and delete stored procedures you've already created
- How to make stored procedures more flexible using input parameters
- How to return information from stored procedures using output parameters

Why Should You Use Stored Procedures?

The queries you've looked at so far have always been single queries that performed only one action; for example, you've performed SELECT queries that return one set of results and INSERT queries that insert data into a single table. When you step into more complex data (and need more complex results), you'll find this way of working constrictive.

Stored procedures can contain just single SQL queries, and when they do so, they're naturally direct replacements for single queries. However, stored procedures are a powerful tool that can deliver much more than this. As you'll learn in this chapter, stored procedures can

increase the performance of your queries and make maintaining Web sites a whole lot easier. They may contain multiple queries and exploit the power of SQL itself.

In a nutshell, using stored procedures gives you the following benefits:

Maintenance: Hard-coded queries on individual pages mean a string of SQL on each and every one of those pages. If you use the same query on several pages and need to change the query, you have to make the changes in every page. Stored procedures make maintaining the site easier by having only one copy of the query.

Security: Allowing direct access to the tables within the database to applications, as you've already seen in Chapter 2, forces you to grant "too much" access to the database. By using stored procedures, you allow the user access to the tables only through stored procedures, and you can apply suitable controls.

Speed: Depending on the database server that you're using, you may also gain a speed advantage from using stored procedures. If you're using MySQL 5.0 and pass a query to the server, the query must be parsed and an execution plan calculated for the query. If you pass the same query three times, MySQL 5.0 calculates the execution plan three times. MySQL 5.0 stored procedures are cached when they're created, and thereafter the precached versions are used. With SQL Server 2005, stored procedures do not process much more quickly than queries, because it also caches and reuses the execution plans of queries passed directly.

Reduced network traffic: Stored procedures allow you to process the results at the database and return only the required results to the page.

Note This chapter won't present all the intricacies of SQL and all the different queries that are supported. You can find an introduction to SQL in Appendix B. It also will not cover the ability to write stored procedures in C# that was introduced with SQL Server 2005, as it's an extremely advanced topic. If you're interested, see *Pro SQL Server 2005 Assemblies* by Robin Dewson and Julian Skinner (1-59059-566-1; Apress, 2005).

Configuring MySQL 5.0 to Use Stored Procedures

When using SQL Server 2005 to connect to a database, it works "straight out of the box." By specifying the correct connection string, you can call stored procedures; no other configuration is required.

When using MySQL 5.0, things aren't as simple. The Odbc data provider that you've been using doesn't support the full range of features that are available in MySQL 5.0. In particular, the support for stored procedures isn't adequate and won't allow you to use stored procedures to their fullest extent.

To work around this limitation, you're not going to use the Odbc data provider for connecting to MySQL 5.0; instead, you're going to use the native provider for MySQL. The `MySqlClient` data provider, `MySQL Connector/Net`, handles stored procedures in the same way as the SQL Server 2005 data provider does.

Note As I said earlier in this book, if there is a native data provider, you should always use that provider. We've been using the Odbc data provider to connect to MySQL 5.0 only to show all three of the supplied data providers in action.

Download Connector/Net from <http://dev.mysql.com/downloads/connector/net/1.0.html>. Once you've installed Connector/Net (by running the installer), configure it as follows:

- Give the account you're using permission to SELECT from a table in the mysql database. The mysql database is the *master* database that controls the operation of the database server, and you must give SELECT permission on the proc table. You can do this by executing the following query in MySQL Query Browser:

```
GRANT SELECT ON mysql.proc TO band
```

- Add a reference to Connector/Net to your Web site by selecting Add Reference from the context menu for the Web site. Click the Browse tab and navigate to C:\Program Files\MySQL\MySQL Connector Net 1.0.7\bin\ .NET 2.0 (your directory will be different if you have a different version of Connector/Net installed). The DLL that you want to reference is `MySQL.Data.dll`, as shown in Figure 10-1.

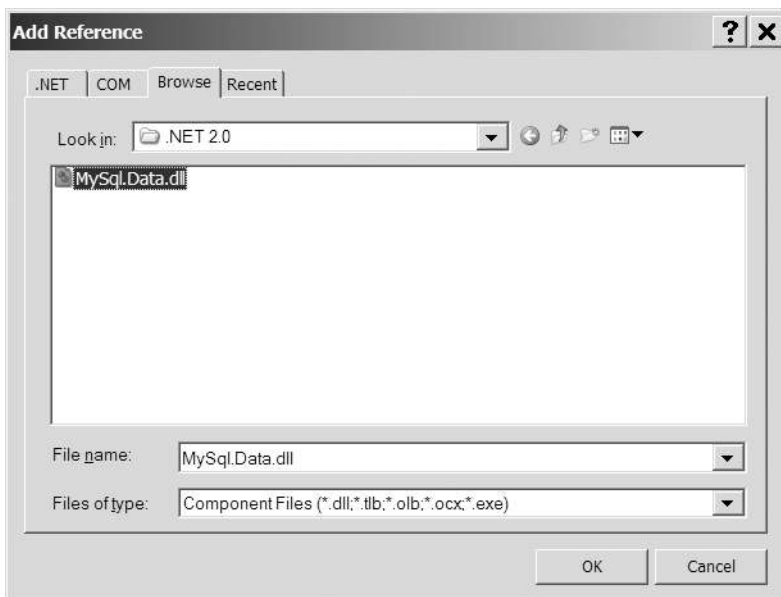


Figure 10-1. Referencing Connector/Cased as Net in your Web site

Once you've added the correct reference to your Web site, you can use Connector/Net in the same way as you would any other data provider.

As you've already seen, the different data providers all define their own objects, and Connector/Net is no different. All of the classes are contained in the `MySQL.Data.MySqlClient` namespace and are prefixed with `MySQL`. So you have `MySQLConnection`, `MySQLCommand`, and so on.

The one thing that the current version of Connector/Net doesn't handle is being used by the `SqlDataSource`. At the moment, you're limited to using `Command` or `DataAdapter` objects to access the database.

Note Although this chapter includes the steps for creating stored procedures in MySQL Query Browser, we're not going to walk through code examples for using Connector/Net. As you've seen in previous chapters, the process for using all of the different data providers is the same. In the code download for the chapter, you'll find the three Command-based pages rewritten using Connector/Net in the `mysql` folder.

Creating Stored Procedures

To create a stored procedure, you must use a Data Definition Language (DDL) query. You'll look at DDL queries in more detail in Chapter 11, but for now, all you need to know is that DDL queries allow you to change the structure of the database.

The DDL query to create stored procedures is `CREATE PROCEDURE`. Both SQL Server 2005 and MySQL 5.0 use the same query to create a stored procedure, but their syntax is slightly different.

For SQL Server 2005, `CREATE PROCEDURE` in its simplest form is as follows:

```
CREATE PROCEDURE <name>
AS
<queries>
```

You give the stored procedure a name, and you can include whatever SQL queries you want after the `AS` statement. Any parameters that are required by the stored procedure are defined between the stored procedure's name and the `AS` statement.

For MySQL 5.0, the corresponding query to create a stored procedure is as follows:

```
CREATE PROCEDURE <name> ()
BEGIN
  <queries>
END;
```

The actual structure of the `CREATE PROCEDURE` query itself is quite similar to the SQL Server 2005 equivalent. You give the stored procedure a name, and you can include whatever SQL queries you want between the `BEGIN` and `END` statements. Any parameters to the stored procedure are defined within the brackets after the stored procedure's name.

We'll now look at creating a simple stored procedure in both SQL Server 2005 and MySQL 5.0.

Try It Out: Creating a Stored Procedure in SQL Server 2005

In this example, you'll use SQL Server Management Studio to create a stored procedure that returns all the Manufacturers from the database. You'll see that the tools you have at your disposal make it quite easy to create stored procedures.

1. Open SQL Server Management Studio and connect to the localhost\BAND database server using the sa account (the password is bandpass). In the Object Explorer, expand the Databases node, the Players database node, and then the Programmability node.
2. Right-click the Stored Procedures node and select New Stored Procedure from the context menu. This will load a template in the main design window, as shown in Figure 10-2.

```

localhost\BAND....- SQLQuery1.sql
-- =====
-- Template generated from Template Explorer using:
-- Create Procedure (New Menu).SQL
--
-- Use the Specify Values for Template Parameters
-- command (Ctrl-Shift-M) to fill in the parameter
-- values below.
--
-- This block of comments will not be included in
-- the definition of the procedure.
-- =====
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
-- =====
-- Author:      <Author,,Name>
-- Create date: <Create Date,,>
-- Description: <Description,,>
-- =====
CREATE PROCEDURE <Procedure_Name, sysname, ProcedureName>
-- Add the parameters for the stored procedure here
  <@Param1, sysname, @p1> <Datatype_For_Param1, , int> = <Default_Value_For_Param1, , 0>,
  <@Param2, sysname, @p2> <Datatype_For_Param2, , int> = <Default_Value_For_Param2, , 0>
AS
BEGIN
-- SET NOCOUNT ON added to prevent extra result sets from
-- interfering with SELECT statements.
  SET NOCOUNT ON;

-- Insert statements for procedure here
  SELECT <@Param1, sysname, @p1>, <@Param2, sysname, @p2>
END
GO

```

Figure 10-2. SQL Server Management Studio makes the creation of stored procedures a breeze.

3. Replace the auto-generated template with the following stored procedure declaration:

```

CREATE PROCEDURE spGetManufacturers
AS

SELECT ManufacturerName
FROM Manufacturer
ORDER BY ManufacturerName

```

4. Click Execute on the toolbar to run the query and create the stored procedure.
5. Return to the Object Explorer and expand the Stored Procedures node. You'll see that the spGetManufacturers stored procedure has been added, as shown in Figure 10-3. (You may need to click Refresh on the Stored Procedures node to update the Object Explorer.)

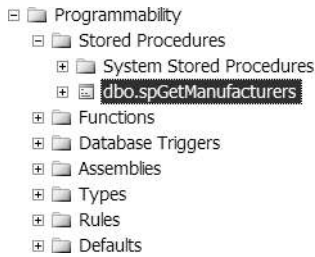


Figure 10-3. The new stored procedure has been saved to the SQL Server 2005 database.

- Right-click `spGetManufacturers` and select `Execute` from the context menu. This is a parameterless query, so in the `Execute Procedure` dialog box, click `OK`. You'll see the `Output` window with the results of executing the query, as shown in Figure 10-4.

	ManufacturerName
1	Apple
2	Cowon
3	Creative
4	Frontier Labs
5	iRiver
6	MSI
7	Rio
8	Samsung
9	SanDisk
10	Sony

Figure 10-4. SQL Server Management Studio can also execute stored procedures.

- In the `Object Explorer`, right-click the `Players` database node and select `New Query` from the context menu. Enter the following query:

```
GRANT EXEC ON spGetManufacturers TO band
```

- Click `Execute` to execute the `GRANT` query against the database. If this query executes correctly, the `Results` pane will change to show a success message, as shown in Figure 10-5.

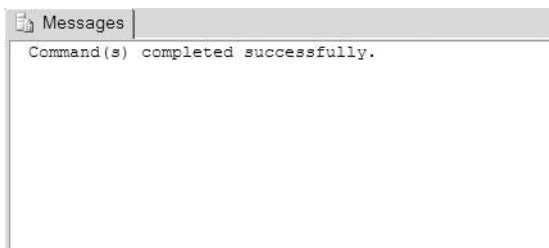


Figure 10-5. Confirmation that the `GRANT` query has executed correctly

How It Works

In this example, you've created your first stored procedure in SQL Server 2005. Granted, it's a simple stored procedure wrapping a query that you've already seen, but it does demonstrate the basic concept.

The first thing to look at is the account you use to connect to the database. To execute DDL queries, you must use an account with administrator privileges within the database. In this case, you're using the sa account, because the band account has permissions only on certain objects and doesn't have any administrator privileges.

As you saw in step 2, SQL Server 2005 automatically creates a stored procedure template for you. This template contains the basic definition of the stored procedure and hints at some of the possibilities, such as parameters, that we'll look at later in this chapter. However, it is quite a complex template, and there is a lot that you need to delete for most stored procedures. It's sometimes easier to simply discard the template and enter the stored procedure declaration from scratch, as you did in this example.

You created the following query in this example:

```
CREATE PROCEDURE spGetManufacturers
AS

SELECT ManufacturerName
FROM Manufacturer
ORDER BY ManufacturerName
```

This is a simple SELECT query that returns all of the Manufacturers in the database.

The one point to note is the name of the stored procedure. Although you created the stored procedure with the name spGetManufacturers, the name in Object Explorer is actually dbo.spGetManufacturers. All objects in the database need to be *owned*, and the name of the object is prefixed with the owner of the object. As you didn't specify the owner when creating the stored procedure, SQL Server Management Studio used the details of the currently logged-in user, sa, to determine the owner, dbo.

When specifying the owner of an object in SQL Server 2005, you don't actually specify a login as the owner of the object, but instead specify a schema. Schemas are quite an advanced topic but basically are a level of abstraction that SQL Server 2005 uses between logins and database objects. The sa login actually belongs to the dbo (database owner) schema, and that is why the spGetManufacturers stored procedure becomes dbo.spGetManufacturers.

Although the name of the stored procedure is prefixed with its owner, the name of the stored procedure is still spGetManufacturers, which is the name you use when you run the stored procedure.

After you click the Execute button, SQL Server Management Studio runs the query to create the stored procedure. If the query executes correctly, the stored procedure will be added to the database and be shown in the Object Explorer underneath the Stored Procedures entry. If, however, there's a problem with the query to create the stored procedure, an error message is returned, as shown in Figure 10-6.

Once any errors are corrected, the stored procedure will be created, and you can execute the stored procedure to check that it's working correctly. Once you're happy that the stored procedure performs as intended, you need to turn your attention to allowing users to access it.

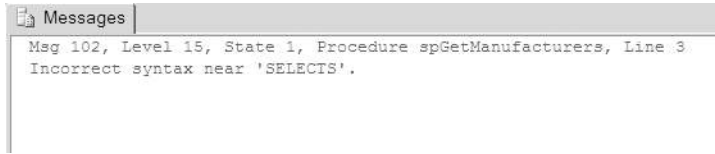


Figure 10-6. Errors are returned if you try to create an invalid stored procedure.

By default, this stored procedure won't be available for use with any of the accounts in the database other than the stored procedure's owner—in this case, the sa account. As you've already seen, running applications using this account isn't recommended. You have a login called band that you're using to access the database from your Web pages. You can give an account permission to execute the stored procedure by granting it the EXEC permission using a GRANT similar to the ones you saw in Chapter 2, like so:

```
GRANT EXEC ON spGetManufacturers TO band
```

As the GRANT query doesn't return any results, the only indication that the stored procedure has executed correctly is the confirmation in the Results pane. All you really want to know is that the query has executed, so the confirmation is adequate.

Try It Out: Creating a Stored Procedure in MySQL 5.0

To demonstrate creating stored procedures in MySQL 5.0, you will use MySQL Query Browser to create a stored procedure that returns all the Manufacturers from the database.

1. Open MySQL Query Browser and connect to the localhost database server using the root account (the password is bandpass).
2. In the Schemata pane, right-click the Players database and select Create New Procedure / Function from the context menu. Enter a name of spGetManufacturers in the dialog box, as shown in Figure 10-7.

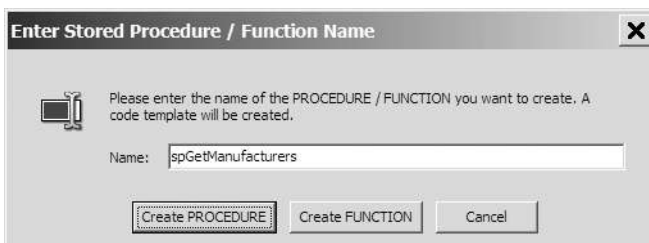


Figure 10-7. MySQL requires the stored procedure name to be specified in advance.

3. Replace the auto-generated template with the following stored procedure declaration:

```
DELIMITER $$

CREATE PROCEDURE spGetManufacturers ()
BEGIN
```



```

SELECT ManufacturerName
FROM Manufacturer
ORDER BY ManufacturerName;
END $$

```

DELIMITER ;

- Click Execute on the toolbar to run the query and create the stored procedure.
- Return to the Schemata pane and expand the Players database. You'll see that the stored procedure has been added to the database, as shown in Figure 10-8.



Figure 10-8. The stored procedure has been saved to the MySQL 5.0 database.

- Double-click the `spGetManufacturers` stored procedure. The query window at the top of the page will be populated with the correct SQL to execute the stored procedure, as shown in Figure 10-9.



Figure 10-9. The SQL required to execute the stored procedure

- Click Execute to run the stored procedure. The results will be displayed, as shown in Figure 10-10.

ManufacturerName
Apple
Cowon
Creative
Frontier Labs
iRiver
MSI
Rio
Samsung
SanDisk
Sony

Figure 10-10. MySQL Query Browser can also execute stored procedures.

8. In the Schemata pane, right-click the Players database node and select New Query from the context menu. Enter the following query:

```
GRANT EXECUTE ON PROCEDURE spGetManufacturers TO band
```

9. Click Execute to execute the GRANT query against the database. The only sign that the query has executed correctly will be that you don't receive an error, and MySQL Query Browser indicates that the "Query returned no resultset."

How It Works

As you can see, creating your first stored procedure in MySQL 5.0 is no more difficult than creating the stored procedure in SQL Server 2005.

Again, MySQL Query Browser provides a template that you can use to create the stored procedure. It's slightly more complete, as you've already provided the name of the stored procedure. You could modify this to suit your purposes, but it is often easier to specify the query from scratch, as you've done here:

```
DELIMITER $$

CREATE PROCEDURE spGetManufacturers ()
BEGIN
    SELECT ManufacturerName
    FROM Manufacturer
    ORDER BY ManufacturerName;
END $$

DELIMITER ;
```

You're using the same SELECT query as the previous example, but this time you have to do a little more work to create the stored procedure.

Because MySQL 5.0 uses the semicolon to delimit the end of a query, you can't actually use it within the body of the CREATE PROCEDURE query. Each query within the stored procedure needs to be delimited correctly, and as soon as the semicolon is added, the CREATE PROCEDURE query is seen as complete—at the point of the semicolon, and not after the closing END where it should be. To solve this problem, you need to use the DELIMITER query to change the character that is being used (in this case, to \$\$) to allow the CREATE PROCEDURE query to contain semicolons. Once the CREATE PROCEDURE query is complete, the delimiter is then set back to the semicolon.

Once the stored procedure has been created, it is available only to the account that created it—in this case, the root account. You need to grant permission to execute the stored procedure by granting the EXECUTE permission:

```
GRANT EXECUTE ON PROCEDURE spGetManufacturers TO band
```

This has a slightly different syntax than the GRANT query in SQL Server 2005 does.

Granting Permissions for Stored Procedures

As you know, you must explicitly grant permissions for users to access the objects in the database. In Chapter 2, you saw that to enable a SELECT query to be executed against a table, you needed to

give the user specific permissions to SELECT information from the table. So to give the band account SELECT permissions on the Manufacturer table, you would execute the following:

```
GRANT SELECT ON Manufacturer TO band
```

You would also need to give similar permission if you wanted to grant the band account INSERT, UPDATE, or DELETE permissions on the table.

Stored procedures also must have permissions applied in order for an account to be able to execute them. Stored procedures don't have SELECT, INSERT, UPDATE, or DELETE permissions, but instead have a single permission: EXEC in SQL Server 2005 and EXECUTE in MySQL 5.0. The syntax to grant this permission for a stored procedure is the same as granting any other permission. In SQL Server 2005, you grant the EXEC permission as follows:

```
GRANT EXEC ON spGetManufacturers TO band
```

And in MySQL 5.0, the syntax for granting the EXECUTE permission is as follows:

```
GRANT EXECUTE ON PROCEDURE spGetManufacturers TO band
```

Similarly, if you want to remove an existing permission, you can use the REVOKE query to remove the permission. For SQL Server 2005, the syntax for REVOKE is as follows:

```
REVOKE EXEC ON spGetManufacturers FROM band
```

For MySQL 5.0, use this syntax:

```
REVOKE EXECUTE ON PROCEDURE spGetManufacturers FROM band
```

SQL Server 2005 also allows you to explicitly deny the EXEC permission on a stored procedure with the following DENY query:

```
DENY EXEC ON spGetManufacturers TO band
```

Note Once a user has been granted execute permissions on a stored procedure, no other permissions are required for that user to execute the stored procedure. The user executing the stored procedure doesn't need any permissions to access the tables used by the stored procedure. The database assumes that the stored procedure is correct and allows it access to all the tables within the database.

Calling Stored Procedures

You now have a stored procedure that will return all the Manufacturers in the database. Next, you need to look at how you call the stored procedure for execution instead of passing a query. Thankfully, calling a stored procedure is similar to passing a query.

When accessing the database in code in previous chapters, you've just passed the query into the Command object and used the correct execute method. For example, to return a DataReader containing the results of the query, you would use the following code:

```
string myCommandText = "SELECT ManufacturerID, ManufacturerName
    FROM Manufacturer ORDER BY ManufacturerName";
```

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

```
SqlDataReader myReader = myCommand.ExecuteReader();
```

The same is true when using a `DataAdapter`. Instead of returning a `DataReader` from the `Command` object, you would create a new `DataAdapter` and use that to fill a `DataSet`:

```
SqlDataAdapter myAdapter = new SqlDataAdapter(myCommand);
DataSet myDataSet = new DataSet();
myAdapter.Fill(myDataSet, "Manufacturers");
```

Similarly, when using a `SqlDataSource`, you've set the `Command` property (`SelectCommand`, `DeleteCommand`, `InsertCommand`, or `UpdateCommand`) to the query to be executed:

```
<asp:SqlDataSource ID="SqlDataSource1" runat="server"
    ConnectionString="<%= ConnectionStrings:SqlConnectionString %>"
    SelectCommand="SELECT ManufacturerID, ManufacturerName
    FROM Manufacturer ORDER BY ManufacturerName">
</asp:SqlDataSource>
```

To use a stored procedure in place of a query, you need to set the `command type` property. Both the `Command` object and the `SqlDataSource` object assume that the command passed is a query and use the default value of `Text` for the `command type` property. As this is the default, you don't need to set it, which is why you have not seen this property before.

To call a stored procedure, you simply pass the name of the procedure instead of a SQL query and specify the `command type` as `StoredProcedure`. When using a `Command` object, you pass the name of the stored procedure to the `Command` object and specify the correct `CommandType`:

```
string myCommandText = "spGetManufacturers";
```

```
SqlCommand myCommand = new SqlCommand();
myCommand.Connection = myConnection;
myCommand.CommandText = myCommandText;
myCommand.CommandType = CommandType.StoredProcedure;
```

For the `SqlDataSource`, you follow the same pattern and simply tell the Web control that you're passing the name of a stored procedure using the `SelectCommandType`, `DeleteCommandType`, `InsertCommandType`, or `UpdateCommandType` property:

```
<asp:SqlDataSource ID="SqlDataSource1" runat="server"
    ConnectionString="<%= ConnectionStrings:SqlConnectionString %>"
    SelectCommand="spGetManufacturers"
    SelectCommandType="StoredProcedure">
</asp:SqlDataSource>
```

Note Although I've intimated that the enumeration value that you're setting for the Command object and SqlDataSource command type properties is the same, this is not the case. The Command object uses values from the System.Data.CommandType enumeration, whereas the SqlDataSource object uses the System.Web.UI.WebControls.SqlDataSourceCommandType enumeration. Both of these enumerations contain Text and StoredProcedure values.

Try It Out: Using a Command Object to Call a Stored Procedure

Now that you have a stored procedure in the database and have set its permissions correctly, you can use that stored procedure in place of a SQL query within a page. In this example, you'll use the stored procedure with SQL Server 2005 to create a simple page that lists all of the Manufacturers in the database.

1. Create a new Web site at C:\BAND\Chapter10 and delete the auto-generated Default.aspx file.
2. Add a new Web.config file to the application 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 application called Calling_DataReader.aspx. Set the <title> of the page to **Calling a Stored Procedure Using a DataReader**.
4. In the Design view, add a GridView to the page.
5. Switch to the Source view and make sure you've included the correct namespaces at the top of the page:

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

6. Add a Page_Load event to the page:

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

    try
    {
        // create the command
        SqlCommand myCommand = new SqlCommand();
        myCommand.Connection = myConnection;
```

```
// set up the command
myCommand.CommandText = "spGetManufacturers";
myCommand.CommandType = CommandType.StoredProcedure;

// open the connection
myConnection.Open();

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

// set up the grid
GridView1.DataSource = myReader;
GridView1.DataBind();

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

7. Execute the page. You'll see that the stored procedure you've created is executed and that the results are returned as expected, as shown in Figure 10-11.

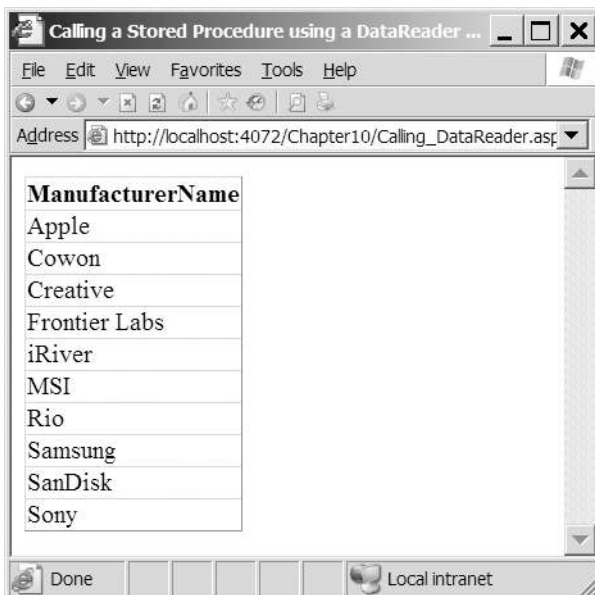


Figure 10-11. Results from executing the `spGetManufacturers` stored procedure

How It Works

The code for the `Page_Load` event is almost identical to the code that you've been using to access the database. The two lines of code that are of particular interest here are as follows:

```
myCommand.CommandText = "spGetManufacturers";  
myCommand.CommandType = CommandType.StoredProcedure;
```

Instead of a SQL query, you give the name of the stored procedure as the `CommandText` of the `Command` object. You then tell the `Command` object that what you're passing in is the name of a stored procedure and not a SQL query by setting the `CommandType` property to `CommandType.StoredProcedure`.

Note If you don't set the `CommandType` correctly for the stored procedure, the stored procedure will still execute because the database makes an intelligent guess; figuring that an invalid query is probably a stored procedure name. However, if you explicitly instruct the database that you're passing in a stored procedure name, it makes your code not only more readable but also slightly quicker because you're removing the cost of forcing the database to choose a "best fit" from the instruction it receives. There is one caveat to this and that concerns the use of parameters. If you pass parameters to the stored procedure and forget to change the `CommandType` to `CommandType.StoredProcedure`, you'll get a runtime error.

Choosing an Execute Method

As you learned in the earlier chapters, you can use three execute methods to execute a query against the database. Which one you use depends on what the query that you're executing is doing. To recap, the three methods that you can use are as follows:

- `ExecuteNonQuery()`: Use this when the query doesn't return any results from the database. It is typically used when you're executing `INSERT`, `UPDATE`, or `DELETE` queries.
- `ExecuteReader()`: Use this when you want to return a result set from a `SELECT` query.
- `ExecuteScalar()`: Use this when you want to return only the first column from the first row of the returned result set. This is almost always used to return the results of a scalar query. You also have the same choice when you use stored procedures, and you should choose the method that matches what the stored procedure does.

Note You also have the option of using the `Command` object to create a new `DataAdapter` to populate a `DataSet`. As you've seen in earlier chapters, the code to use a `DataAdapter` and a `DataSet` is very similar to the code to use a `DataReader`, so it should be easy for you to use a stored procedure this way.

Try It Out: Calling a Stored Procedure in a SqlDataSource

In this example, you'll use a `SqlDataSource` to call your stored procedure.

1. Add a new Web Form to the application called `Calling_DataSource.aspx`. Set the `<title>` of the page to **Calling a Stored Procedure in a `SqlDataSource`**.
2. Switch to the Design view and add a `SqlDataSource` to the page. In the Properties window, set the `ConnectionString` to `SqlConnectionString` and set the `SelectCommandType` property to `StoredProcedure`.
3. Click the ellipsis for the `SelectQuery` property and enter `spGetManufacturers` as the `SELECT` command.
4. Add a `GridView` to the page and set `SqlDataSource1` as its data source.
5. Execute the page. You'll see that the `GridView` shows the same list of `Manufacturers` as you saw in Figure 10-11.

How It Works

You can see how easy it is to use a stored procedure rather than a SQL query when using a `SqlDataSource`. If you look at the markup that is generated, you'll see that the `SelectCommand` has been changed to the name of the stored procedure and the `SelectCommandType` has changed to reflect the fact that you're executing a stored procedure:

```
<asp:SqlDataSource ID="SqlDataSource1" runat="server"
  ConnectionString="<%$ ConnectionStrings:SqlConnectionString %>"
  SelectCommand="spGetManufacturers"
  SelectCommandType="StoredProcedure">
</asp:SqlDataSource>
```

Note As with the `SqlCommand` object, if you don't set `SelectCommandType` correctly for the stored procedure, the database will make an intelligent guess and check that what you're after is a stored procedure. If you're using parameters, you must set `SelectCommandType` to `StoredProcedure`.

Altering and Deleting Stored Procedures

Being able to add stored procedures to your database is all well and good, but you also need some way of modifying or deleting them. You can accomplish both of these tasks using two other DDL commands.

To modify a stored procedure, you use the `ALTER PROCEDURE` query, specifying the name of the stored procedure you want to modify as well as the complete new contents of the stored procedure, like so:


```
ALTER PROCEDURE <name>  
AS  
<new queries>
```

To delete a stored procedure from the database, you simply use the `DROP PROCEDURE` query, like so:

```
DROP PROCEDURE <name>
```

SQL Server Management Studio and MySQL Query Browser provide options to complete both of these tasks from the user interface. If you select a stored procedure and open the context menu, you'll see that there are options for editing and deleting stored procedures: the `Modify` and `Delete` options in SQL Server Management Studio, and the `Edit Procedure` and `Drop Procedure` options in MySQL Query Browser.

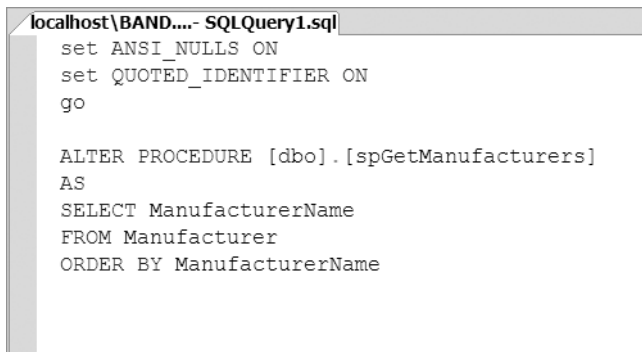
Deleting a stored procedure is straightforward. Simply right-click the procedure and select `Delete` or `Drop Procedure` from the context menu. You'll see a dialog box that asks you to confirm that you want to delete the stored procedure.

In the following examples, you'll see how to modify existing stored procedures.

Try It Out: Modifying a Stored Procedure in SQL Server 2005

In this example, you'll change the stored procedure you've already created using SQL Server Management Studio.

1. Open SQL Server Management Studio and connect to the `localhost\BAND` database server using the `sa` account.
2. Expand the `Databases`, `Players`, `Programmability`, then `Stored Procedures` node. Right-click the `dbo.spGetManufacturers` entry and select the `Modify` option to modify the `spGetManufacturers` stored procedure. Figure 10-12 shows the SQL that is generated to modify the current stored procedure.



```
localhost\BAND....- SQLQuery1.sql  
set ANSI_NULLS ON  
set QUOTED_IDENTIFIER ON  
go  
  
ALTER PROCEDURE [dbo].[spGetManufacturers]  
AS  
SELECT ManufacturerName  
FROM Manufacturer  
ORDER BY ManufacturerName
```

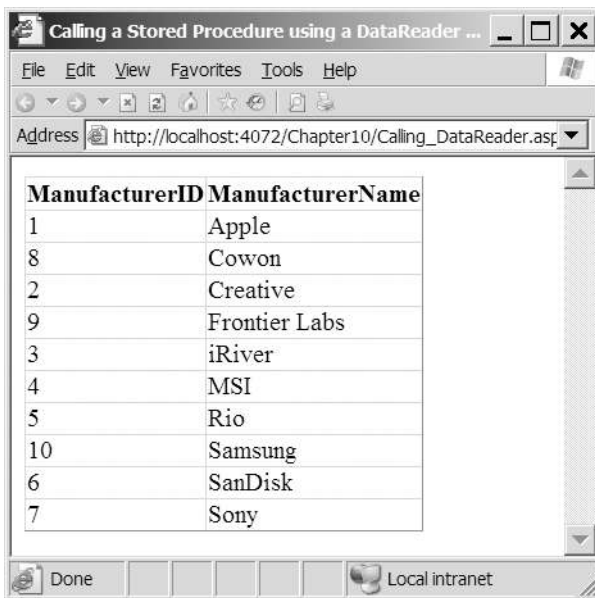
Figure 10-12. *Modifying an existing stored procedure in SQL Server Management Studio*

3. Change the stored procedure to the following:

```
ALTER PROCEDURE dbo.spGetManufacturers
AS
```

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

4. Click the Execute button on the toolbar to run the query and modify the stored procedure.
5. Open either of the pages that you've created from the earlier examples. As shown in Figure 10-13, this will return the results from the stored procedure, with the addition of the ManufacturerID column.



ManufacturerID	ManufacturerName
1	Apple
8	Cowon
2	Creative
9	Frontier Labs
3	iRiver
4	MSI
5	Rio
10	Samsung
6	SanDisk
7	Sony

Figure 10-13. Results from executing the modified stored procedure

How It Works

You've specified the new query for the stored procedure and saved it to the database. When you now execute the page and call the `spGetManufacturers` stored procedure, you get the results from the new query rather than the old one.

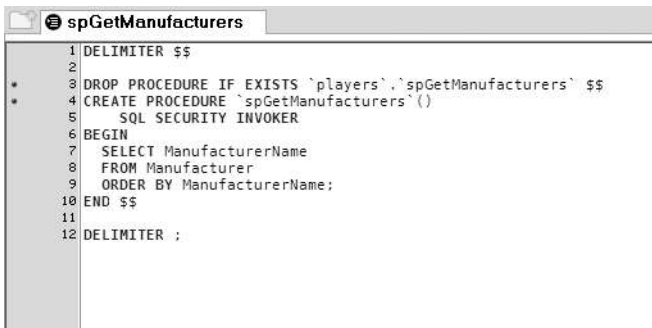
Editing a stored procedure in SQL Server Management Studio opens the stored procedure in the designer with the stored procedure ready to be modified as shown in Figure 10-12. It's then a simple task to modify the stored procedure.

Notice that you don't need to add any permissions for the stored procedure. As this is a modification of an existing stored procedure, any permissions that were applied to the original stored procedure will still be applied to the new stored procedure.

Try It Out: Modifying a Stored Procedure in MySQL 5.0

Modifying stored procedures in MySQL Query Browser is also quite simple. But, as you'll see, you don't use the ALTER PROCEDURE query.

1. Open MySQL Query Browser and connect to the localhost database server using the root account.
2. Expand the Players node in the Schemata pane. Right-click the spGetManufacturers entry and select the Edit Procedure option to modify the spGetManufacturers stored procedure. Figure 10-14 shows the SQL that is generated to modify the current stored procedure.



```

1 DELIMITER $$
2
3 DROP PROCEDURE IF EXISTS `players`.`spGetManufacturers` $$
4 CREATE PROCEDURE `spGetManufacturers`()
5     SQL SECURITY INVOKER
6 BEGIN
7     SELECT ManufacturerName
8     FROM Manufacturer
9     ORDER BY ManufacturerName;
10 END $$
11
12 DELIMITER ;

```

Figure 10-14. Modifying an existing stored procedure in MySQL Query Browser

3. Change the query between the BEGIN and END statements to the following:


```

SELECT ManufacturerID, ManufacturerName
FROM Manufacturer
ORDER BY ManufacturerName;

```
4. Click the Execute button on the toolbar to run the query and modify the stored procedure. The only sign that the stored procedure has been modified is that you did not receive an error.
5. Execute the stored procedure in MySQL Query Browser. You'll see that it has been modified, as shown in Figure 10-15.

ManufacturerID	ManufacturerName
1	Apple
8	Cowon
2	Creative
9	Frontier Labs
3	iRiver
4	MSI
5	Rio
10	Samsung
6	SanDisk
7	Sony

Figure 10-15. Results from executing the modified stored procedure in MySQL Query Browser

6. You now need to add the permissions for the new stored procedure. In the Schemata pane, right-click the Players database node and select New Query from the context menu. Enter the following query:

```
GRANT EXECUTE ON PROCEDURE spGetManufacturers TO band
```

7. Click Execute to execute the GRANT query against the database.

How It Works

MySQL Query Browser isn't as user-friendly as SQL Server Management Studio, but it still gets the job done, just not in the same way. It doesn't modify the existing stored procedure; it deletes it and creates a new one!

The first line changes the delimiter, and then the stored procedure is dropped if it exists:

```
DROP PROCEDURE IF EXISTS 'players'. 'spGetManufacturers'
```

The naming is slightly different here than SQL Server 2005. Instead of the owner, the name of the stored procedure is prefixed with the database that contains the stored procedure.

Once the existing stored procedure has been deleted, the stored procedure is re-created using a `CREATE PROCEDURE` query. After the new stored procedure has been created, you then need to add the permissions again.

This simple stored procedure returns all the Manufacturers in the database. This is just the beginning of the story. By responding dynamically to users and their interaction with your pages, you can impart real power to your applications. You can do this by using parameters with your stored procedures, as you'll learn in the remainder of this chapter.

Creating Stored Procedures with Input Parameters

In Chapter 4, you looked at two methods of modifying the queries when connecting to the database in code. You've also looked at using parameters when dealing with the `SqlDataSource` in Chapter 3 to pass values into the queries that were automatically executed. You pass parameters into stored procedures in the same way as you pass them to queries.

Whether using code or a `SqlDataSource`, you've been using *input parameters*—parameters that pass information to the query that being executed. SQL Server 2005 and MySQL 5.0 also define another type of parameter: an output parameter. Output parameters allow data to be returned from the stored procedure, as well as the results of any `SELECT` queries that have been executed. We'll look at using output parameters later in this chapter.

Using input parameters with stored procedures is a two-stage process:

- Define the parameters in the stored procedure declaration.
- Add the parameters to the `SqlCommand` or `SqlDataSource` before the call is made to execute the stored procedure.

Creating a stored procedure that requires parameters to be supplied isn't more complex than creating a stored procedure that doesn't accept parameters. You need to modify the stored procedure declaration slightly to list the parameters that are required.

For SQL Server 2005, you define the stored procedure as follows:

```
CREATE PROCEDURE <name>
<parameters>
AS
<queries>
```

For MySQL 5.0, you define the stored procedure like this:

```
CREATE PROCEDURE <name> (<parameters>)
BEGIN
  <queries>
END;
```

The parameters list is simply a list giving the name of the parameter and its type. If you have multiple parameters, separate them with commas.

SQL Server 2005 requires parameter names to be prefixed with the @ symbol. It's customary to put each parameter on its own line (it makes the stored procedure a lot more readable when editing), like so:

```
@name1 type,
@name2 type
```

MySQL 5.0 doesn't allow the @ prefix, so you use the name without the prefix:

```
name1 type, name2 type
```

All parameters that you include in the stored procedure declaration must be passed to the stored procedure by the calling application; if they aren't, an error is thrown.

SQL Server 2005 also allows you to override this default behavior. If you're sure you don't want to pass the parameter, you can override this behavior by giving the parameter a default value, as follows:

```
@name3 type = default
```

This will be used if the parameter isn't supplied a value by the calling application. If you always want a value to be passed to the stored procedure, then you don't specify a default value.

Note Although you can use default values with stored procedures in SQL Server 2005, you're perhaps better off not doing so. Personally, I find that they're another place that an error can creep into a Web site. It's too easy to forget to pass the parameter, and this may cause the stored procedure to give incorrect results. When using the `SqlDataSource`, you can't use default values set within the stored procedures; the default values must be set on the parameters themselves within the page.

You'll now look at creating a stored procedure that accepts a parameter that modifies the results that are returned as part of the query. This stored procedure will also introduce you to the idea of flow control. You'll see how to use the IF statement to control what actions the stored procedure takes.

Try It Out: Creating a Stored Procedure with Input Parameters in SQL Server 2005

In this example, you'll create a new stored procedure that accepts a `ManufacturerID` and returns a list of `Players` or, if the `ManufacturerID` is passed as zero, returns all the `Players` in the database.

1. Open SQL Server Management Studio and connect to the `localhost\BAND` database server using the `sa` account.
2. Expand the `Databases`, then `Players`, then `Programmability` node. Select `New Stored Procedure` from the `Stored Procedure` node's context menu.

3. Give the stored procedure the following declaration:

```
CREATE PROCEDURE spGetPlayersByManufacturer
@manufacturer int
AS

IF (@manufacturer = 0) BEGIN
    SELECT Player.PlayerID, Player.PlayerName, Player.PlayerStorage,
           Player.PlayerCost, Manufacturer.ManufacturerName
    FROM Player INNER JOIN Manufacturer
        ON Player.PlayerManufacturerID = Manufacturer.ManufacturerID
    ORDER BY Player.PlayerName
END ELSE BEGIN
    SELECT PlayerID, PlayerName, PlayerStorage, PlayerCost
    FROM Player
    WHERE PlayerManufacturerID = @manufacturer
    ORDER BY Player.PlayerName
END
```

4. Click `Execute` to run the query and create the stored procedure.
5. In the `Object Explorer`, expand the `Stored Procedures` node and then expand the `spGetPlayersByManufacturer` stored procedure (you may need to refresh the `Stored Procedures` node to see the new stored procedure). Expand the `Parameters` node, and you'll be able to see the parameters for a stored procedure, without needing to open the stored procedure declaration, as shown in Figure 10-16.
6. Select the `spGetPlayersByManufacturer` stored procedure and select `Execute Stored Procedure` from the context menu. You'll be prompted to enter the values for the parameters, as shown in Figure 10-17.
7. Enter a value of `0` for the `@manufacturer` parameter and click `OK`. You'll see that all of the `Players` in the database are returned. Rerun the stored procedure and enter a value of `1` for the `@manufacturer` parameter, and you'll see that only the `Players` made by `Apple` are returned.

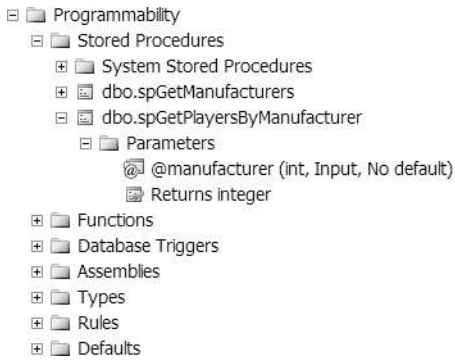


Figure 10-16. You can easily view the parameters for a stored procedure in SQL Server Management Studio.

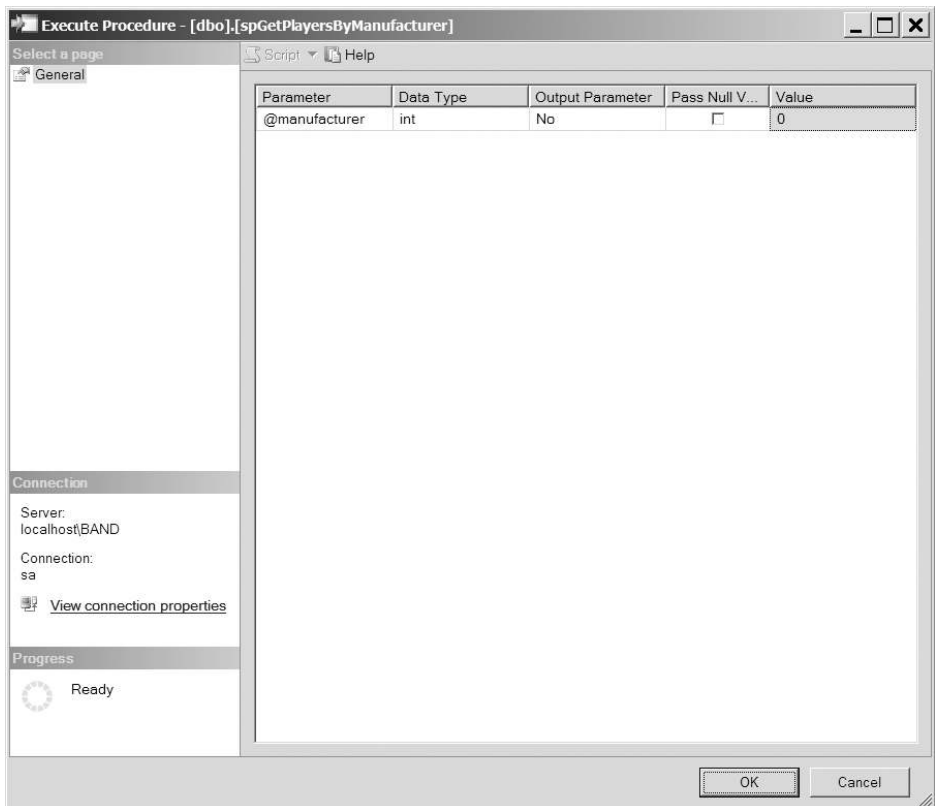


Figure 10-17. SQL Server Management Studio allows you to test parameterized stored procedures.

8. In the Object Explorer, right-click the Players database node and select New Query from the context menu. Execute the following query to set the permissions for the `spGetPlayersByManufacturer` stored procedure:

```
GRANT EXEC ON spGetPlayersByManufacturer TO band
```

How It Works

The way you've created this new stored procedure is the same way you created the `spGetManufacturers` stored procedure, and, unsurprisingly, you'll create every stored procedure this way. Here, you're interested in the structure of the stored procedure.

You add the parameters to the stored procedure definition between the stored procedure name and the AS statement, like so:

```
CREATE PROCEDURE dbo.spGetPlayersByManufacturer
@manufacturer int
AS
```

You have only one parameter in this particular query, and it's called `@manufacturer`. The name must be prefixed by the `@` symbol to indicate that it's a user variable (as opposed to a system variable, which will have a prefix of `@@`). As you'll see when you call the stored procedure in the next example, this is the name you'll need to use when adding the parameter to the Command object and the `SqlDataSource`.

After the name of the parameter, you have the parameter's type. For types that require a size as well (such as the `varchar` type), you also need to include the size you're expecting.

Once you have declared the parameter, you can then use the parameter within the stored procedure. In this example, you use the parameter value to determine the results that are returned from the stored procedure.

If you don't want to filter the Players that are returned as part of the query, you can assume that a `ManufacturerID` of zero is passed to the stored procedure. You can use the `ManufacturerID` to determine the route through the stored procedure by using the IF statement:

```
IF (@manufacturer = 0) BEGIN
    <queries>
END ELSE BEGIN
    <queries>
END
```

As in C#, the IF statement lets you control what's executed within the stored procedure. As the condition of the IF statement, you can use any valid SQL that returns a Boolean value. You can do simple comparisons as you have here (all the usual operators are available), or you can use scalar functions (such as `EXISTS`) to control execution.

Depending on whether the condition evaluates to true or false, you take a different path. If it's true, you follow the path before the ELSE statement. If it's false, you follow the path after the ELSE statement.

Note The BEGIN and END statements in SQL Server 2005 are equivalent to the opening and closing braces in C#. As with C#, you don't need them if you have only one query as part of the conditional path, but it makes the stored procedure a lot easier to read if they're present.

If you have a @manufacturer value of 0, you know that you don't want to filter the query, and you execute a SELECT query that returns the PlayerID, PlayerName, PlayerStorage, PlayerCost, and ManufacturerName for all the Players in the database.

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

If, however, you have a nonzero value for @manufacturer, indicating that a value has been passed into the stored procedure, you use this to constrain the SELECT query and return a slightly different list of columns, like so:

```
SELECT PlayerID, PlayerName, PlayerStorage, PlayerCost
FROM Player
WHERE PlayerManufacturerID = @manufacturer
ORDER BY Player.PlayerName
```

Try It Out: Creating a Stored Procedure with Input Parameters in MySQL 5.0

You'll now create the corresponding spGetPlayersByManufacturer stored procedure in MySQL 5.0.

1. Open MySQL Query Browser and connect to the localhost database server using the root account (the password is bandpass).
2. In the Schemata pane, right-click the Players database and select Create New Procedure / Function from the context menu. Enter a name of **spGetPlayersByManufacturer** and click the Create PROCEDURE button.
3. Enter the following stored procedure declaration:

```
DELIMITER $$

CREATE PROCEDURE spGetPlayersByManufacturer (manufacturer int)
BEGIN
  IF (manufacturer = 0) THEN
    SELECT Player.PlayerID, Player.PlayerName, Player.PlayerStorage,
           Player.PlayerCost, Manufacturer.ManufacturerName
```

```

FROM Player INNER JOIN Manufacturer
  ON Player.PlayerManufacturerID = Manufacturer.ManufacturerID
ORDER BY Player.PlayerName;
ELSE
  SELECT PlayerID, PlayerName, PlayerStorage, PlayerCost
  FROM Player
  WHERE PlayerManufacturerID = manufacturer
  ORDER BY Player.PlayerName;
END IF;
END $$
DELIMITER ;

```

4. Click Execute on the toolbar to run the query and create the stored procedure.
5. Expand the Players database in the Schemata pane. Expand the `spGetPlayersByManufacturer` stored procedure, and you'll see the parameters that the stored procedure requires, as shown in Figure 10-18.

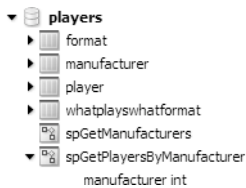


Figure 10-18. You can easily view the parameters for a stored procedure in MySQL Query Browser.

6. Double-click the `spGetPlayersByManufacturer` stored procedure. The query window at the top of the page will be populated with a SQL query to execute the stored procedure.
7. Enter a value of **0** between the brackets and click Execute to run the stored procedure. A list of all of the Players in the database will be returned.
8. Change the query and enter a value of **1** between the brackets, and then click Execute to run the stored procedure. This time, the list of Players will be filtered for Apple.
9. In the Schemata pane, right-click the Players database node and select New Query from the context menu. Enter the following query:

```
GRANT EXECUTE ON PROCEDURE spGetPlayersByManufacturer TO band
```

10. Click Execute to execute the GRANT query against the database.

How It Works

When creating a parameterized stored procedure in MySQL 5.0, the parameters are specified between the brackets of the `CREATE PROCEDURE` query, as follows:

```
CREATE PROCEDURE spGetPlayersByManufacturer (manufacturer int)
```

Any value that is passed for the `ManufacturerID` will be available as the `manufacturer` variable (note the lack of the `@` prefix), and you can then use this value to determine the `SELECT` query that is executed:

```
IF (manufacturer = 0) THEN
    <queries>
ELSE
    <queries>
END IF;
```

This is slightly different from the SQL Server 2005 `IF` construct, but it should be readily apparent what is happening.

Passing Parameters to Stored Procedures

You were introduced to passing parameters to a `SqlDataSource` in Chapter 3 and to `Command` objects in Chapter 4. You've also made extensive use of them in the intervening chapters to pass parameters into queries that you sent directly to the database. You use the same techniques to pass parameters to stored procedures.

When using the `Command` object, you need to create a `Parameter` object and set the name, type, and value before adding it to the `Parameters` collection. The `SqlDataSource` allows you to add parameters that can automatically bind to a variety of different values.

We'll look at both of these situations in turn to create pages that can accept a `ManufacturerID` as a query string parameter and modify the `Players` list that is displayed accordingly.

Try It Out: Using Input Parameters with a Command Object

You'll now build a slightly more complex example that displays the `Players` in the database and uses the `spGetPlayersByManufacturer` stored procedure to filter for which `Manufacturer` you're returning the `Players`.

1. Open Visual Web Developer. Open the `Calling_DataReader.aspx` page and save it as `Input_DataReader.aspx`.
2. Switch to the Source view and modify the code in the `Page_Load` event as follows:

```
// set up the command
myCommand.CommandText = "spGetPlayersByManufacturer";
myCommand.CommandType = CommandType.StoredProcedure;

// get the manufacturer value from the querystring
string strManufacturerID = Request.QueryString["manufacturerid"];

// determine the correct value as an integer
int intManufacturerID = 0;
if (strManufacturerID != null)
```

```

{
    intManufacturerID = Convert.ToInt32(strManufacturerID);
}

// create the parameter
SqlParameter myParameter1 = new SqlParameter();
myParameter1.ParameterName = "@manufacturer";
myParameter1.SqlDbType = SqlDbType.Int;
myParameter1.Value = intManufacturerID;

// add it to the command object
myCommand.Parameters.Add(myParameter1);

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

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

```

3. Execute the page. You'll see that the list of Players returned includes all of the Players in the database, as shown in Figure 10-19.

The screenshot shows a web browser window titled "Calling a Stored Procedure using a DataReader - Microsoft Internet Explorer". The address bar shows the URL "http://localhost:4072/Chapter10/Input_DataReader.aspx". The main content area displays a table with the following data:

PlayerID	PlayerName	PlayerStorage	PlayerCost	ManufacturerName
18	Carbon	Hard Disk	169.00	Rio
7	Digital Audio Player	Solid State	119.00	SanDisk
6	Forge	Solid State	93.00	Rio
16	H10	Hard Disk	189.00	iRiver
17	H300 Series	Hard Disk	319.00	iRiver
12	iAudio M3	Hard Disk	249.00	Cowon
3	iFP-700 Series	Solid State	149.00	iRiver
4	iFP-900 Series	Solid State	149.00	iRiver
9	iPod	Hard Disk	209.00	Apple
10	iPod Mini	Hard Disk	169.00	Apple
11	iPod Photo	Hard Disk	309.00	Apple
1	iPod Shuffle	Solid State	99.00	Apple
15	L1	Hard Disk	149.00	Frontier Labs
5	MegaPlayer 521	Solid State	199.00	MSI
2	MuVo V200	Solid State	96.00	Creative
10	Newton VII 030	Hard Disk	170.00	Sony

Figure 10-19. Results showing all the Players in the database

4. Modify the address that you're viewing and add `?manufacturerid=1` to the URL. Press Enter to load the page. You see all the Players manufactured by Apple, as shown in Figure 10-20.

PlayerID	PlayerName	PlayerStorage	PlayerCost
9	iPod	Hard Disk	209.00
10	iPod Mini	Hard Disk	169.00
11	iPod Photo	Hard Disk	309.00
1	iPod Shuffle	Solid State	99.00

Figure 10-20. Results showing the Players for a particular Manufacturer

How It Works

This simple example demonstrates that passing parameters to stored procedures is the same as passing parameters to queries.

You call a stored procedure by creating a Command object. You set the `CommandText` to the name of the stored procedure, `spGetPlayersByManufacturer`, and the `CommandType` to `CommandType.StoredProcedure`. You then check to see whether a `ManufacturerID` has been added to the query string. As you know, `Request.QueryString` returns null if the requested value isn't present, and you use this fact to default to a `ManufacturerID` of 0 if the query string value isn't present:

```
// do we need to add the @manufacturer parameter
string strManufacturerID = Request.QueryString["manufacturerid"];

// determine the correct value as an integer
int intManufacturerID = 0;
if (strManufacturerID != null)
{
    intManufacturerID = Convert.ToInt32(strManufacturerID);
}
```

You then need to add the parameter to the Command object. You create a `SqlParameter` object and give it the correct name. Because you're using the `SqlCommand` object, you need to use the name that the stored procedure expects, so you use `@manufacturer`, like so:

```
// create the parameter
SqlParameter myParameter1 = new SqlParameter();
myParameter1.ParameterName = "@manufacturer";
```

You then specify the type of the parameter from the `SqlDbType` enumeration and set the value of the parameter to the value from the query string, like so:

```
myParameter1.SqlDbType = SqlDbType.Int;  
myParameter1.Value = intManufacturerID;
```

Once the parameter has been created and correctly populated, you can add it to the `Parameters` collection of the `SqlCommand` object, like so:

```
// add it to the command object  
myCommand.Parameters.Add (myParameter1);
```

You then use the `ExecuteReader()` method of the `SqlCommand` object to return a `SqlDataReader` object and bind this to the data grid.

When the stored procedure is executed, the route that's taken depends on the value passed as the parameter to the `SqlCommand` object, as discussed when you created the stored procedure in the previous example.

Try It Out: Using Input Parameters with a `SqlDataSource`

In this example, you'll build the same page as you saw in the previous example, but this time, you'll use a `SqlDataSource`.

1. Open Visual Web Developer. Open the `Calling_DataSource.aspx` page and save it as `Input_DataSource.aspx`.
2. Switch to the Design view and select the `SqlDataSource`. In the Properties window, click the ellipsis next to the `SelectQuery` property.
3. Change the `SELECT` query to `spGetPlayersByManufacturer`.
4. Click the Add Parameter button and change the name of the added parameter to `manufacturer`.
5. Change the Parameter source to `QueryString` and set the `QueryStringField` value as `manufacturerid`. Enter a default value of `0`.
6. Click OK to close the Command and Parameter Editor dialog box.
7. Execute the page. You'll see that the list of Players returned includes all of the Players in the database, as shown earlier in Figure 10-18.
8. Modify the address that you're viewing and add `?manufacturerid=1` to the URL. Press Enter to load the page. You'll see that it displays only Players manufactured by Apple, as shown earlier in Figure 10-19.

How It Works

In Chapter 3, you learned that you can automatically pass several different types of parameters to the various commands of the `SqlDataSource`. In this case, you're using the `manufacturerid` query string value:

```
<SelectParameters>
  <asp:QueryStringParameter DefaultValue="0" Name="manufacturer"
    QueryStringField="manufacturerid" Type="Int32" />
</SelectParameters>
```

The only new detail to the `QueryStringParameter` is the addition of the `DefaultValue` property. The `DefaultValue` allows you to specify the value that you want to pass into the stored procedure when, in this example, the query string value that you're after doesn't exist. For this stored procedure, you need to pass a value of 0 for the `@manufacturer` parameter when you don't have a `manufacturerid` value specified in the query string.

Using Parameters with MySQL 5.0

As you saw in earlier chapters, when using parameters in SQL Server 2005, you can add parameters to the `SqlCommand` object in whatever order you like, because the `SqlCommand` object supports named parameters. When connecting to MySQL 5.0 using the `Odbc` data provider, the parameters must be added in the correct order. Thankfully, using `Connector/Net` when connecting to MySQL 5.0 removes this limitation, as it does support named parameters.

You saw when creating the parameterized query in MySQL 5.0 that you specify parameters in brackets after the stored procedure name. In MySQL 5.0, local variables aren't prefixed with a `@` as they are in SQL Server 2005, so the definition of `spGetPlayersByManufacturer` is as follows:

```
CREATE PROCEDURE spGetPlayersByManufacturer (manufacturer int)
```

To add the `manufacturer` parameter to the `MySqlCommand` object, you need to use the parameter name prefixed with `?`, as follows:

```
MySqlParameter myParameter1 = new MySqlParameter();
myParameter1.ParameterName = "?manufacturer";
myParameter1.MySqlDbType = MySqlDbType.Int32;
myParameter1.Value = intManufacturerID;
myCommand.Parameters.Add(myParameter1);
```

Other than those few little changes, parameters—both input and output—work the same way in MySQL 5.0 as they do in SQL Server 2005.

Returning Data Using Output Parameters

Although you've now seen how you can pass parameters to a stored procedure, this isn't the end of what you can do with parameters. You can also use them to return values from a stored procedure.

You've already looked at two ways of returning data from the database: using the `ExecuteReader()` and `ExecuteScalar()` methods of the `Command` object, and by data binding when using a `SqlDataSource`.

Sometimes, however, you want to execute a query that doesn't return any results directly but returns information detailing what you've just done. If you're inserting information into the database, you may want to return a key for what you've just inserted. Or you may want to return more information than can be returned within the scope of the normal `SELECT` query. *Output parameters* are the key to this.

You can modify the values of user variables within stored procedures, and if they're parameters, the changed value may be returned to the calling application. If the parameter is an input parameter, the value isn't returned. When you use output parameters, the changed value is returned.

Note The name *output parameter* is perhaps a bit misleading because an output parameter is more accurately an input/output parameter. You can pass a value into a stored procedure using an output parameter, and any changes to the parameter will be reflected in the parameter once control has returned from the stored procedure.

To use output parameters instead of input parameters, you don't need to do a lot of work. You can accomplish it in the following two stages:

- Tell the stored procedure that the parameter is an output parameter.
- Tell the Command object or `SqlDataSource` that the stored procedure is an output parameter, and then retrieve the changed value from the parameter after the stored procedure has executed.

To use output parameters within the stored procedure you need to mark the definition of the parameter as being an output parameter. For SQL Server 2005, you use the `OUTPUT` statement with the definition of the parameter, like so:

```
@name type = default OUTPUT
```

In MySQL 5.0, you need to prefix the parameter declaration with `OUT`, like so:

```
OUT name type
```

You're free to mix input and output stored procedures however you want in the stored procedure declaration.

As with input parameters, SQL Server 2005 also allows you to give output parameters default values that will be used if the parameter isn't passed to the stored procedure by the calling application.

Try It Out: Creating a Stored Procedure with Output Parameters in SQL Server 2005

You'll now build on the stored procedure in the previous example to include an output parameter that returns the number of Players for the selected Manufacturer.

1. Open SQL Server Management Studio and connect to the `localhost\BAND` database server using the `sa` account.
2. Expand the Databases, Players, then Programmability node and select New Stored Procedure from the Stored Procedure node's context menu.

3. Add the following stored procedure declaration:

```
CREATE PROCEDURE dbo.spGetPlayersWithCountByManufacturer
@manufacturer int,
@rowcount int OUTPUT
AS

IF (@manufacturer = 0) BEGIN
    SELECT Player.PlayerID, Player.PlayerName, Player.PlayerStorage,
        Player.PlayerCost, Manufacturer.ManufacturerName
    FROM Player
        INNER JOIN Manufacturer
            ON Player.PlayerManufacturerID = Manufacturer.ManufacturerID
    ORDER BY Player.PlayerName

    SET @rowcount = @@ROWCOUNT
END ELSE BEGIN
    SELECT PlayerID, PlayerName, PlayerStorage, PlayerCost
    FROM Player
    WHERE PlayerManufacturerID = @manufacturer
    ORDER BY Player.PlayerName

    SET @rowcount = @@ROWCOUNT
END
```

4. Click Execute to run the query and save the modified stored procedure to the database.

5. Expand the Parameters node underneath the spGetPlayersByManufacturer stored procedure. You'll see that output parameters are also shown, but with a different icon, indicating that they're output parameters, as shown in Figure 10-21.

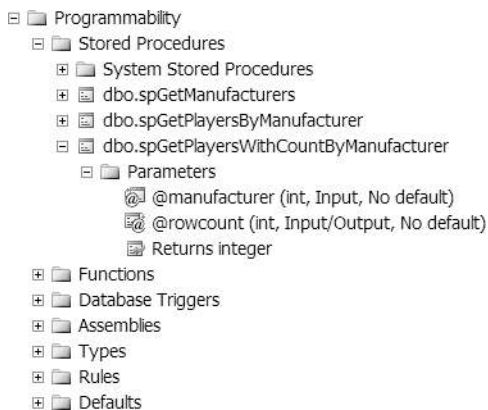


Figure 10-21. Output parameters are displayed with a different icon in SQL Server Management Studio.

6. In the Object Explorer, right-click the Players database node and select New Query from the context menu. Execute the following query to set the permissions for the `spGetPlayersByManufacturer` stored procedure:

```
GRANT EXEC ON spGetPlayersWithCountByManufacturer TO band
```

How It Works

As you can see from the stored procedure declaration, you've added the following output parameter of type `int` called `@rowcount`:

```
@rowcount int OUTPUT
```

You'll use this to return the count of the number of rows that are returned for the Manufacturer you've selected. You use the `SET` statement to assign values to variables, and you can set `@rowcount` to the number of rows returned using the `@@ROWCOUNT` system variable, like so:

```
SET @rowcount = @@ROWCOUNT
```

`@@ROWCOUNT` returns the number of rows that the previous SQL query returned or affected and is valid not only for `SELECT`, but also for `DELETE`, `INSERT`, and `UPDATE`.

Try It Out: Creating a Stored Procedure with Input Parameters in MySQL 5.0

You'll now create the corresponding `spGetPlayersWithCountByManufacturer` stored procedure in MySQL 5.0.

1. Open MySQL Query Browser and connect to the `localhost` database server using the root account (the password is `bandpass`).
2. In the Schemata pane, right-click the Players database and select Create New Procedure / Function from the context menu. Enter a name of `spGetPlayersWithCountByManufacturer` and click the Create PROCEDURE button.
3. Enter the following stored procedure declaration:

```
DELIMITER $$
```

```
CREATE PROCEDURE spGetPlayersByManufacturer (manufacturer int,  
      OUT rowcount int)  
BEGIN  
    IF (manufacturer = 0) THEN  
        SELECT Player.PlayerID, Player.PlayerName, Player.PlayerStorage,  
               Player.PlayerCost, Manufacturer.ManufacturerName  
        FROM Player INNER JOIN Manufacturer  
            ON Player.PlayerManufacturerID = Manufacturer.ManufacturerID  
        ORDER BY Player.PlayerName;
```

```

SET rowcount = (SELECT COUNT(*) FROM Player);
ELSE
SELECT PlayerID, PlayerName, PlayerStorage, PlayerCost
FROM Player
WHERE PlayerManufacturerID = manufacturer
ORDER BY Player.PlayerName;

SET rowcount = (SELECT COUNT(*) FROM Player
WHERE PlayerManufacturerID = manufacturer);
END IF;
END $$
DELIMITER ;

```

4. Click Execute on the toolbar to run the query and create the stored procedure.
5. Expand the Players database in the Schemata pane, and then expand the `spGetPlayersWithCountByManufacturer` stored procedure. You can see the parameters that the stored procedure requires, as shown in Figure 10-22.

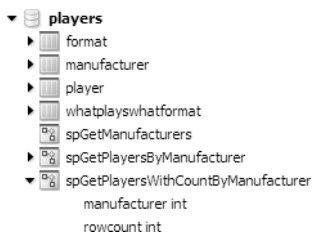


Figure 10-22. Output parameters aren't shown differently in MySQL Query Browser.

6. In the Schemata pane, right-click the Players database node and select New Query from the context menu. Enter the following query:

```
GRANT EXECUTE ON PROCEDURE spGetPlayersWithCountByManufacturer TO band
```

7. Click Execute to execute the GRANT query against the database.

How It Works

Adding an output parameter to a stored procedure in MySQL 5.0 is as simple as prefixing the parameter declaration with `OUT`:

```
OUT rowcount int
```

You'll use this to return the count of the number of rows that are returned for the Manufacturer you've selected. Unfortunately, the `ROW_COUNT()` function in MySQL doesn't work in stored procedures, so you need to use the `COUNT` scalar function to return the number of rows selected. For the unfiltered list, this is as follows:

```
SET rowcount = (SELECT COUNT(*) FROM Player);
```

The filtered list is returned the same way:

```
SET rowcount = (SELECT COUNT(*) FROM Player
WHERE PlayerManufacturerID = manufacturer);
```

Returning Parameters from Stored Procedures

You've already seen how you add input parameters to the `SqlCommand` object and `SqlDataSource`. Adding output parameters is similar, except that you need to change the direction of the parameter. This is because parameters are defined as input parameters by default.

When using the `SqlCommand` object and the `SqlDataSource`, you specify the direction for a parameter using the `Direction` property and setting it to one of the values in the `System.Data.ParameterDirection` enumeration. The default value for the `Direction` property is `Input`. To specify it for an input parameter to the `SqlCommand` object, use this form:

```
myParameter.Direction = ParameterDirection.Input
```

Similarly, for an input parameter to a `SqlDataSource`, you would set the property as follows:

```
Direction="Input"
```

For output parameters, you can use either of the following for the `Direction` property:

- `Output`: Returns the value from the stored procedure, but any value you attempt to send to the stored procedure is ignored.
- `InputOutput`: Allows values to be passed into the stored procedure by the parameter and will return the parameter value from the stored procedure.

Although SQL Server and MySQL allow all `OUTPUT` parameters to accept an input value, `ADO.NET` makes a distinction between these two options.

If you declare a parameter as solely an `Output` parameter, then even if you give it a value before calling the stored procedure, the value will not be passed to the stored procedure. The output value will be set correctly, but any input value is completely ignored.

On the other hand, if you define the parameter as an `InputOutput` parameter, then any value you give it will also be passed to the stored procedure.

Note This is one of the few instances where passing parameters to SQL Server 2005 and MySQL 5.0 is different. If you're using a parameter direction of `Output` or `InputOutput` with MySQL 5.0, you don't need to provide a value for the parameter. However, if you're using SQL Server 2005, you must provide a value if the parameter is an `InputOutput` parameter. If you don't, an error will occur when you try to execute the stored procedure.

Try It Out: Using Output Parameters with a SqlCommand Object

In this example, you'll build on the previous example and return the number of rows from the `spGetPlayersWithCountByManufacturer` stored procedure using an output parameter.

1. Open Visual Web Developer. Open the `Input_DataReader.aspx` page and save it as `Output_DataReader.aspx`.
2. Add the following HTML immediately before the `GridView`:

```
<p>  
Returned <asp:Label id="Label1" runat="server">0</asp:Label> players.  
</p>
```

3. Modify the code in the `Page_Load` event to add the second parameter and retrieve the output parameter after the stored procedure. Add the following code after the declaration of the `@manufacturer` parameter:

```
// add the @rowcount parameter  
SqlParameter myParameter2 = new SqlParameter();  
myParameter2.ParameterName = "@rowcount";  
myParameter2.SqlDbType = SqlDbType.Int;  
myParameter2.Direction = ParameterDirection.Output;  
myCommand.Parameters.Add (myParameter2);
```

4. Add the following to retrieve the output parameter after the line to close the `DataReader`:

```
// now get the output parameter  
Label1.Text = Convert.ToString(myCommand.Parameters["@rowcount"].Value);
```

5. Execute the page. This will execute the stored procedure, return all the `Players` in the database, and update the count of the number of `Players` returned, as shown in Figure 10-23.
6. Modify the address that you're viewing and add `?manufacturerid=1` to the URL. Press Enter to load the page, displaying the `Players` manufactured by Apple as well as the count, as shown in Figure 10-24.

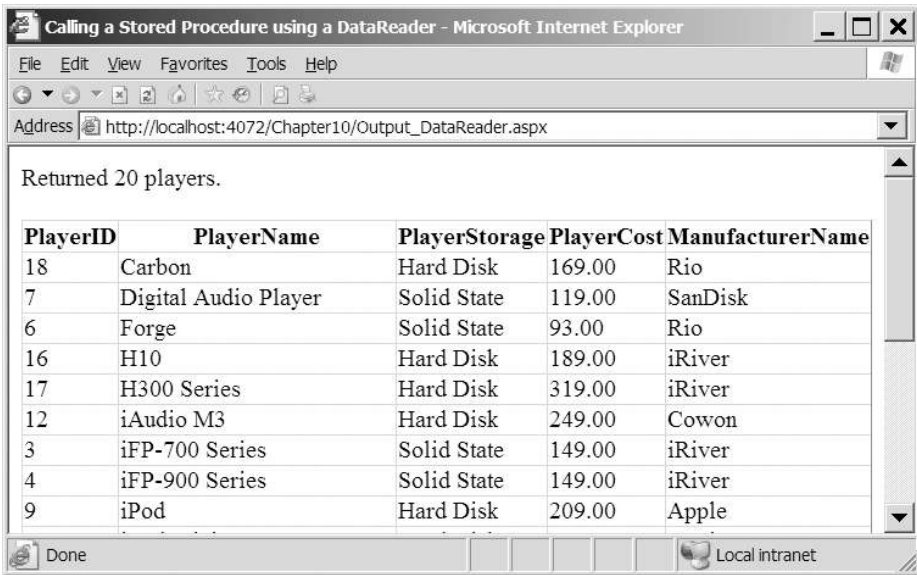


Figure 10-23. Results showing the Players and the count for all the Players

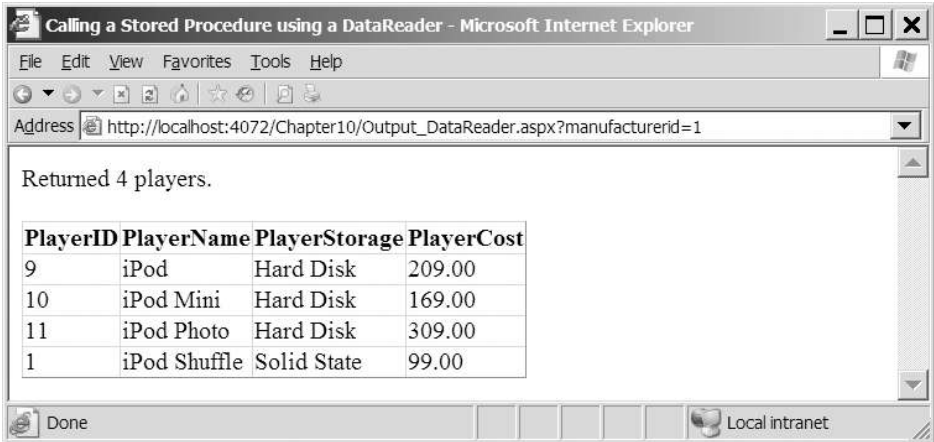


Figure 10-24. Players and their count for a particular Manufacturer

How It Works

The first change you make is to add the output parameter. You declare the parameter as you have all the other parameters you've used, but this time, you specify the direction of the parameter, like so:

```
// add the @rowcount parameter
SqlParameter myParameter2 = new SqlParameter();
myParameter2.ParameterName = "@rowcount";
myParameter2.SqlDbType = SqlDbType.Int;
myParameter2.Direction = ParameterDirection.Output;
myCommand.Parameters.Add (myParameter2);
```

You then execute the stored procedure as you normally would, bind the reader to the GridView, and then close the DataReader. Once the DataReader has been closed, you can retrieve the values of the output parameters simply by using the name of the parameter as the index to the Parameters collection of the Command object, like so:

```
// now get the output parameter
Label1.Text = Convert.ToString(myCommand.Parameters["@rowcount"].Value);
```

You're after the value of the parameter, so you use the Value property. This property returns an Object that you can then cast to whatever type you want. In this case, you want to set the Text property of a Label, so you convert the value to a string.

When using the Command object and output parameters, it's important to remember that you must close the DataReader object you're using before you can access the output parameters. If it isn't closed, the output parameters will not be populated with the results you expect.

Note If you were using `ExecuteScalar()` or `ExecuteNonQuery()` to execute a stored procedure that has output parameters, you wouldn't have any problems and wouldn't need to worry about closing things before you could access the output parameters. This isn't strictly a "problem" with the implementation of the `ExecuteReader()` method, but it's a big enough issue to warrant its own Microsoft Knowledge Base article (<http://support.microsoft.com/default.aspx?scid=kb;en-us;Q308621>). Although the article is about ADO.NET 1.0, the problem still occurs in ADO.NET 2.0. The same problem also affects Connector/Net. So, you should always close the DataReader before you try to use any of the output parameters.

Try It Out: Using Output Parameters with a SqlDataSource

Now you'll build on a previous example to demonstrate using output parameters with a SqlDataSource.

1. Open Visual Web Developer. Open the `Input_DataSource.aspx` page and save it as `Output_DataSource.aspx`.
2. Add the following HTML immediately before the GridView:

```
<p>
Returned <asp:Label id="Label1" runat="server">0</asp:Label> players.
</p>
```

3. Switch to the Design view and select the `SqlDataSource`. In the Properties window, click the ellipsis for the `SelectQuery` property.
4. Click the Add Parameter to add a new parameter. Name the parameter `rowcount`.
5. Click the Show Advanced Properties link and set `Direction` to `Output` and `Type` to `Int32`.
6. Click OK to close the Command and Parameter Editor dialog box.
7. Switch to the Events view for the `SqlDataSource` and add a `Selected` event. Change the code within the event handler as follows:

```
protected void SqlDataSource1_Selected(object sender,
    SqlDataSourceStatusEventArgs e)
{
    Label1.Text = Convert.ToString(e.Command.Parameters["@rowcount"].Value);
}
```

8. Execute the page. This will execute the stored procedure, return all the `Players` in the database, and update the count of the number of `Players` returned, as shown earlier in Figure 10-23.
9. Modify the address that you're viewing and add `?manufacturerid=1` to the URL, and then press Enter to load the page. You'll see the `Players` manufactured by Apple, as well as the count, as shown earlier in Figure 10-24.

How It Works

To return the value as the output parameter, you've created a new parameter without a parameter source and set its `Direction` to `Output` and its `Type` to `Int32`. You can see this more clearly if you look at the markup that is generated:

```
<SelectParameters>
  <asp:QueryStringParameter DefaultValue="0" Name="manufacturer"
    QueryStringField="manufacturerid" />
  <asp:Parameter Direction="Output" Name="rowcount" Type="Int32" />
</SelectParameters>
```

To access the output parameter, you need to catch the `Selected` event from the `SqlDataSource`. Each of the four operations that you can perform has a before and after event (such as `Selecting` and `Selected`), and you need to catch the after event, as you can look at the output parameter value only after the stored procedure has been executed.

Within the event, you access the parameters in much the same way as you do when manually accessing the `Command` object in code, except this time, you use the `Command` property of the event argument:

```
Label1.Text = Convert.ToString(e.Command.Parameters["@rowcount"].Value);
```

You need to get the `Value` property of the parameter. Because it's returned as an `Object`, you cast this to a string before setting the `Text` property for the label to display the count of the number of rows returned.

Note You may be thinking that the syntax for accessing the parameters for the `Command` object and `SqlDataSource` are quite similar. In fact, they're identical. The `Command` property within any of the eight before and after events for the `SqlDataSource` actually returns the `Command` object that is being used internally to execute the stored procedure. Once you have the `Command` object, you're free to interrogate it as you would a `Command` object that you created yourself.

Using Stored Procedures with Other Queries

In this chapter, we've looked at stored procedures that return results; that is, those that contain `SELECT` queries. Stored procedures aren't limited to just `SELECT` queries, though. `INSERT`, `UPDATE`, and `DELETE` queries, which were introduced in Chapter 8, also are perfectly valid queries to use in stored procedures.

We're not going to walk through any step-by-step examples of using these other queries, as the procedure for creating the stored procedure is the same. Instead, we'll look at stored procedure replacements for some of the queries and query batches that we looked at in Chapter 8.

For instance, consider the `UPDATE` query that you used in `Player_Update.aspx` in Chapter 8. You would create the stored procedure using the following query:

```
CREATE PROCEDURE spPlayerUpdate
    @PlayerID int,
    @Name varchar(50),
    @ManufacturerID int,
    @Cost decimal(10,2),
    @Storage varchar(50)
AS
```

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

If you look back to Chapter 8, you'll see that the `UPDATE` query itself is the same query as you passed to the database. Instead of passing the `UPDATE` query to the database, you would pass the name of the stored procedure and call `ExecuteNonQuery()` to execute the stored procedure.

You can call the stored procedure in the same way as the SQL query and also define the parameters in the same way:

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

The `spPlayerUpdate` stored procedure, in this case, is a direct replacement for a single query that you passed to the database. You have five parameters to the SQL query, and you have the same five parameters to the stored procedure. As you're using named parameters, the order that you add the parameters doesn't matter, but it is good practice to keep the ordering consistent in the stored procedure and code. If nothing else, it makes it easier to see whether any of the parameters have been missed.

As you saw with the output parameters example, a stored procedure can do more than execute one query. You can use this to your advantage to simplify the `Player_Insert.aspx` page. When adding a new Player to the database, you need to return the `PlayerID` of the newly created Player. This requires two queries that you executed as a query batch in Chapter 8. But with a stored procedure, you can execute both of the queries in the same stored procedure:

```
CREATE PROCEDURE spPlayerInsert
    @Name varchar(50),
    @ManufacturerID int,
    @Cost decimal(10,2),
    @Storage varchar(50)
AS

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

SELECT SCOPE_IDENTITY();
```

You first insert the Player into the database and then return the `SCOPE_IDENTITY()` value. This returns the `PlayerID` from the stored procedure, and you can access this using the `ExecuteScalar()` method. This is the same technique as you saw for retrieving the `PlayerID` from the query batch in Chapter 8:

```
intPlayerID = Convert.ToInt32(myCommand.ExecuteScalar());
```

The `DELETE` query from `Player_Delete.aspx` can also be used in a stored procedure very easily. In this stored procedure, you're again executing two queries that you previously ran as a query batch:

```
CREATE PROCEDURE spPlayerDelete
    @PlayerID int
AS

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

Nothing to it! Following the procedures you learned in the various examples in this chapter, you'll be able to modify the pages that you created in Chapter 8 to use stored procedures.

Summary

This chapter started by looking at the advantages of using stored procedures. Depending on the database server that you're using, there are several reasons for using stored procedures:

- Simplified maintenance
- Increased security
- Increased performance
- Reduced network traffic

After looking at why you might use stored procedures over direct SQL queries, you then took a step back from the relative complexities of the previous chapters. You created several stored procedures that used the following options you have for passing and returning data to and from stored procedures:

- Returning data using a SELECT query
- Passing parameters using input parameters
- Returning data using output parameters

You also saw that there's very little difference between calling a stored procedure and executing a SQL query. You pass the stored procedure name as the query to execute and tell the Command object or `SqlDataSource` that you're executing a stored procedure by specifying `StoredProcedure`.

We then looked briefly at how to use stored procedures for executing INSERT, UPDATE, and DELETE queries. You saw the definitions of three stored procedures that are replacements for the SQL queries that were introduced in Chapter 8.

In the next chapter, we'll look at the DDL subset of SQL and see what it can do. As you'll see in the next chapter, you can use SQL to create the entire database, without ever going near a graphical tool.