.NET Development for Java Programmers

PAUL GIBBONS

Apress™

.NET Development for Java Programmers
Copyright ©2002 by Paul Gibbons

Printed and bound in the United States of America 1 2 3 4 5 6 7 8 9 10

Trademarked names may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, we use the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 175 Fifth Avenue, New York, NY, 10010 and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany.
In the United States, phone 1-800-SPRINGER, email `orders@springer-ny.com`, or visit `http://www.springer-ny.com`.
Outside the United States, fax +49 6221 345229, email `orders@springer.de`, or visit `http://www.springer.de`.

For information on translations, please contact Apress directly at 2560 9th Street, Suite 219, Berkeley, CA 94710.Phone 510-549-5930, fax: 510-549-5939, email `info@apress.com`, or visit `http://www.apress.com`.

The source code for this book is available to readers at `http://www.apress.com` in the Downloads section. You will need to answer questions pertaining to this book in order to successfully download the code.

Exploring ADO.NET

ACTIVEX DATA OBJECTS for the .NET Framework (ADO.NET) is your database interface in .NET, the equivalent of Java's Java Database Connectivity (JDBC). This chapter covers database access using ADO.NET and how it differs from JDBC. It closely looks at the `DataSet` object, which is the core of the ADO.NET disconnected model and how to use data binding to WinForms controls and ASP.NET controls, in particular the `DataGrid` controls. First, you will discover the tools provided by Visual Studio .NET to assist with database development.

Using Visual Studio .NET Database Features

Although not strictly ADO.NET, this is an ideal time to touch on the Visual Studio .NET tools that facilitate database development. After all, you need to set up a database to develop and run the examples in this chapter. These tools are geared to Microsoft SQL Server development. However, Visual Studio .NET includes a copy of Microsoft SQL Server 2000 Desktop Engine (MSDE), which is what I used to develop these examples.

You might be used to using SQL Enterprise Manager and SQL Query Analyzer to administer SQL Server, but some functions are readily performed from within Visual Studio .NET, making it unnecessary to launch a different tool. It is also important to realize you cannot do everything from the Integrated Development Environment (IDE).

By default, the Server Explorer is a separate tab in the same slide-out panel that holds the Toolbox. You can use the Server Explorer to connect to SQL Server instances on the local or remote systems. Once connected, you can manipulate databases, tables, and stored procedures, providing you have permission to do so.

You cannot modify permissions using the Visual Studio .NET Server Explorer. You will need to launch SQL Enterprise Manager to manipulate these.

Creating Databases

To create a new database, select the SQL Server instance. In this case, choose the local MSDE instance, which will be called VSDOTNET. Right-click the node to open the context menu and select New Database, as shown in Figure 6-1.
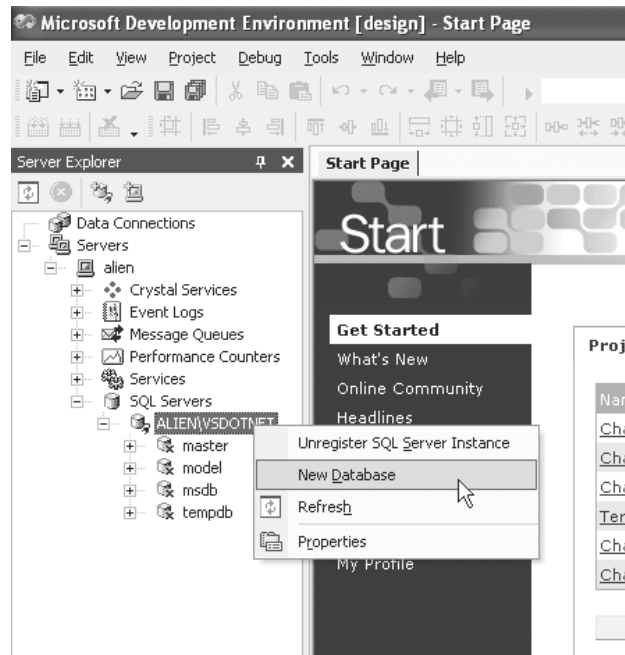
Figure 6-1. Creating a new database

When the Create Database dialog box appears, as in Figure 6-2, supply a name for the database. In this case, enter **chapter6** in the New Database Name box, and click OK. Note that although this is convenient for development purposes, you are not able to control any aspect other than the name. Everything else such as the location, log location, and so on used their default values.
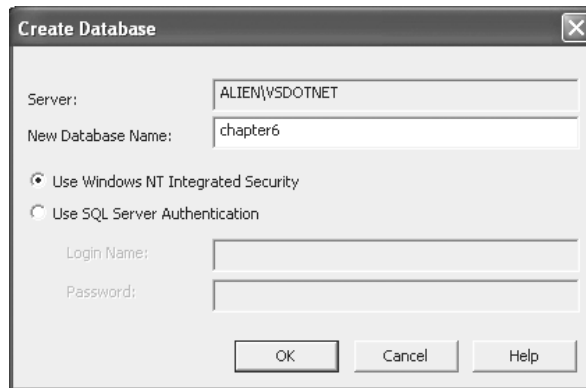
Figure 6-2. Naming a new database

Creating Tables

The first step to create a table is to expand the new database node. You will see a Tables node. Open the context menu for the Tables node and select New Table, as in Figure 6-3.
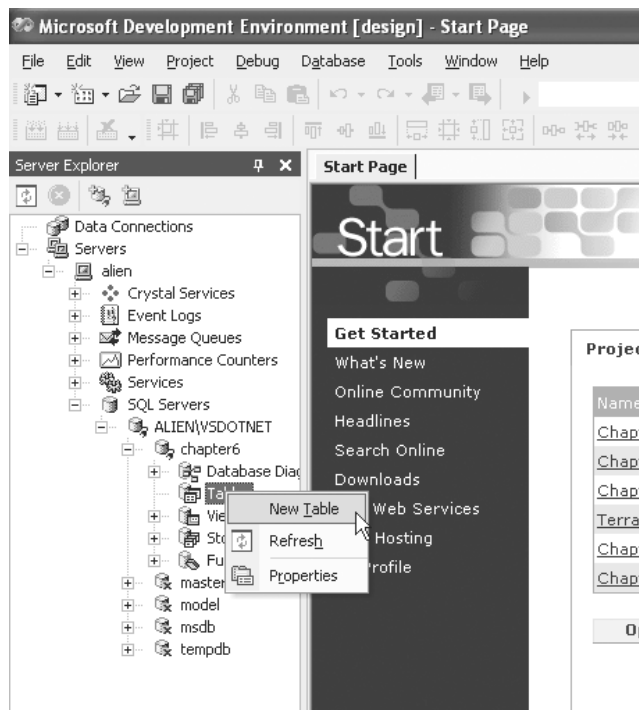
Figure 6-3. Creating a new table

You will then be placed in the design view for a table. This enables you to specify the characteristics for the rows and specify any indexes or constraints. Figure 6-4 shows the rows for a table to hold details of .NET languages.
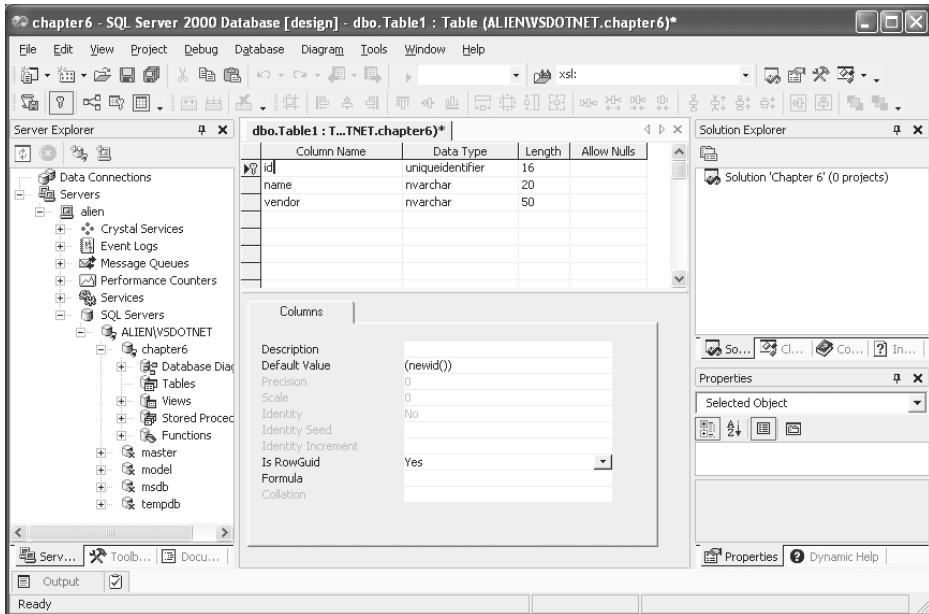
Figure 6-4. Designing a table

Up to this point, the table has been assigned a dummy name, in this case `Table1`. When you save the table, you will be prompted to enter a name, as shown in Figure 6-5. For this example, enter the name **dotnetlanguages**.
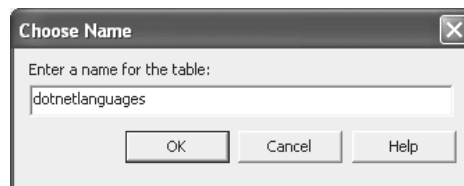
Figure 6-5. Naming a new table

Now you have created the table, you need to enter some data. Select the table and open its context menu. Select Retrieve Data from Table, as shown in Figure 6-6.
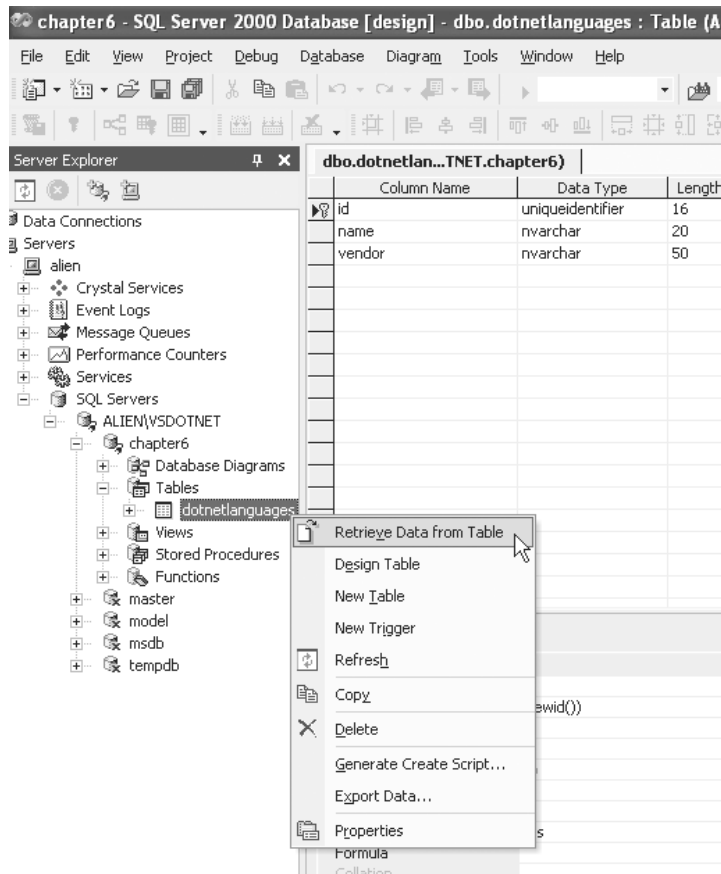
Figure 6-6. Retrieving data from a table

This opens a DataGrid control, which allows you to insert, edit, and delete rows from the table. At this point you need to enter some rows. Providing you correctly set the default for the id column to newid() as shown in Figure 6-4, you can just enter data into the name and vendor columns. As you enter each row, it appears to vanish, but if you rerun the query, by selecting Query ➢ Run or by pressing the toolbar button with the red exclamation mark (!), you will see that the rows are part of the table. If you make a mistake, press the Esc key to abandon entry into a cell or type over your error. If you create a row you do not want, right-click in the left column and select Delete. You should end up with something similar to Figure 6-7.

id	name	vendor
{53410B11-E99F-44C7-A840-09FE7CF1DF57}	J#	Microsoft
{41999612-61D4-4B44-B453-120FEE618520}	C#	Microsoft
{EE0CF9CD-A48C-45B7-8CEF-6AC8EE4EDD5A}	C++	Microsoft
{9436AC21-71B4-46AD-85D6-8D726A0C815D}	Eiffel	ISE
{8D92080C-DDF3-41FB-9809-AEB433939E03}	VB.NET	Microsoft
{3F06A55B-38EE-4CE8-84A9-CB345A6268C2}	Cobol	Fujitsu

Figure 6-7. Editing rows in a table

If you need to restrict the rows, select View ➢ Panes ➢ SQL to open a panel, which allows you to enter SQL much as you would in the SQL Query Analyzer. When you run the query, the results are shown in the `DataGrid` control. Figure 6-8 shows the `dotnetlanguages` table without the `id` field, which is an easier way to enter data in this case.

dbo.dotnetlang...NET.chapter6)*

```
SELECT    name, vendor
FROM      dotnetlanguages
```

name	vendor
J#	Microsoft
C#	Microsoft
C++	Microsoft
Fortran	Fujitsu
Eiffel	ISE
VB.NET	Microsoft
Cobol	Fujitsu

Figure 6-8. Using the SQL pane

Developing Stored Procedures

Creating and editing stored procedures is also made easy using the Server Explorer. Select the Stored Procedures node and open its context menu. Then select New Stored Procedure, as shown in Figure 6-9.
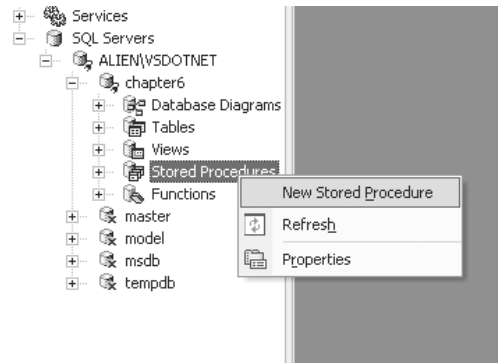
Figure 6-9. Creating a new stored procedure

You will be given the following skeleton stored procedure to start from in an edit panel:

```
CREATE PROCEDURE dbo.StoredProcedure1
/*
    (
        @parameter1 datatype = default value,
        @parameter2 datatype OUTPUT
    )
*/
AS
    /* SET NOCOUNT ON */
    RETURN
```

You can then edit this template to create your procedure. Create the following from the template:

```
CREATE PROCEDURE dbo.ListLanguagesForVendor
(
    @vendor nvarchar(50)
)
AS
    SELECT name
    FROM dotnetlanguages
    WHERE vendor = @vendor

    RETURN
```

Now save it by selecting File ➢ Save. If you have not changed the procedure name from the one suggested in the template, you will be prompted to enter one at this time.

To test the new procedure, open its context menu and select Run Stored Procedure. A dialog box will prompt you for any parameters, in this case `@vendor`. The Output panel will display the results.

Comparing a Simple Query in JDBC and ADO.NET

You will now create a simple query in JDBC and then again in ADO.NET to examine the differences between these two technologies.

Using JDBC

To use JDBC to create a simple query, follow these steps:

1. First create a typical JDBC query for the table `dotnetlanguages` on the local MSDE database `chapter6` that you created in the preceding section.

2. Start by obtaining a `Connection` object:

```
String cstr = "jdbc:microsoft:sqlserver://localhost:1494; "
    + "DataBaseName=Chapter6;User=sa";
Connection conn = null;
Statement stmt = null;
ResultSet rs = null;

try
{
    Class.forName("com.microsoft.jdbc.sqlserver.SQLServerDriver");
    conn = DriverManager.getConnection( cstr );
```

> **NOTE** *Determining which port MSDE is listening on can be difficult. Check the ERRORLOG.1 file located at* `C:\Program Files\Microsoft SQL Server\MSSQL$VSdotNET\LOG`.

3. Next create the Statement object and execute a SQL statement using it. As you are performing a query, use the executeQuery method. This will return a ResultSet object:

```
stmt = conn.createStatement();
rs = stmt.executeQuery( "select * from dotnetlanguages" );
```

4. The next step is to iterate over the rows in the ResultSet and extract the fields you are interested in:

```
while( rs.next() )
{
    System.out.println( rs.getString( "name" ) );
}
```

5. Finally, ensure everything gets closed to avoid waiting for garbage collection. In most cases there are not finalizers to close the ResultSet, Statement, or Connection objects:

```
}
finally
{
    if ( rs != null )
    {
        rs.close();
    }
    if ( stmt != null )
    {
        stmt.close();
    }
    if ( conn != null )
    {
        conn.close();
    }
}
```

Using ADO.NET

Listing 6-1 shows the equivalent .NET code.

Listing 6-1. The .NET Code

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

namespace SimpleQuery
{
    class SimpleQuery
    {
        [STAThread]
        static void Main(string[] args)
        {
            string cstr = @"Server=.\VSdotNET;"
                + @"Database=Chapter6;"
                + @"Integrated Security=SSPI";
            using ( SqlConnection conn = new SqlConnection( cstr ) )
            {
                conn.Open();

                SqlCommand cmd
                    = new SqlCommand( "select * from dotnetlanguages", conn );
                SqlDataReader rdr = cmd.ExecuteReader();

                while ( rdr.Read() )
                {
                    System.Console.WriteLine( "{0}", rdr.GetString( 1 ) );
                }
                rdr.Close();
            }
        }
    }
}
```

Start in a similar way to JDBC by creating a connection object. However, you must explicitly open the connection before using it:

```
string cstr = @"Server=.\VSdotNET;"
    + @"Database=Chapter6;"
    + @"Integrated Security=SSPI";
```

```
using ( SqlConnection conn = new SqlConnection( cstr ) )
{
    conn.Open();
  ....
}
```

In this case you create a `SqlConnection` instance because you are using the SQL Server managed data provider. Whereas JDBC and Microsoft's own OLE DB and ODBC use a factory pattern to return abstract interfaces to the database drivers/providers, ADO.NET uses separate providers whose classes you use directly. This is not a return to chaos, as the providers are built on top of a base class hierarchy and therefore follow a consistent pattern.

There are three managed database providers available at the time of writing: the SQL Server provider used in this example, an OLE DB provider, and an ODBC provider. The ODBC provider is not part of the base install and must be downloaded separately.

The connection string will vary with the provider and driver, just as it does in JDBC. In the SQL Server managed data provider case, the `Server` parameter specifies the server machine and SQL Server instance. Using the MSDE from Visual Studio .NET, my system results in `Server=.\VSdotNet`. If you are not using the local machine, you must substitute your own system name. If you use a full version of SQL Server rather than the MSDE that comes with Visual Studio .NET, remove `\VSdotNET`, unless your DBA instructs you to use a specific SQL Server instance name which would replace `VSdotNET`. The `Database` property specifies which database on the server to use. If you accepted the default authentication option when you installed MSDE, you choose Windows Authentication. In this case, use the `Integrated Security` parameter and specify the value `SSPI`. In this mode, the underlying Windows logon is used to authenticate to SQL Server or MSDE. This avoids the need to hard-code user credentials in the connection string. If you choose mixed authentication, you will have specified a password for the internal SQL Server user account sa during installation. sa is the SQL Server administrator account. Replace the `Integrated Security` parameter with `User ID` and `Password` parameters. The result should be similar to this:

```
string cstr = @"Server=.\VSdotNET;"
    + @"Database=Chapter6;"
    + @"User ID=sa;Password=xxxx";
```

If you did not install SQL Server, you will have to consult you system or database administrator to find out which mode and accounts to use.

> **NOTE** *If you need to adjust your connection string for this example you will need to make similar changes in the connection strings for all other database examples in this book.*

The ADO.NET equivalent of the JDBC `Statement` is the command object, `SqlCommand` in this case. There is no separation equivalent to `Statement` and `PreparedStatement` in ADO.NET. In place of the `ResultSet`, obtain a data reader (a `SqlDataReader` in this example) by invoking the `ExecuteReader` method of the command as follows:

```
SqlCommand cmd =
    new SqlCommand( "select * from dotnetlanguages", conn );
SqlDataReader rdr = cmd.ExecuteReader();
```

Iterating over the rows in the data reader is similar to performing the same operation on a `ResultSet`. A data reader is forward only, and you cannot reposition the current record pointer the way you can with a `ResultSet`:

```
while ( rdr.Read() )
{
    System.Console.WriteLine( "{0}", rdr.GetString( 1 ) );
}
```

When extracting the fields from a data reader's row, use the field index. You cannot use field names with a data reader.

In JDBC closing the `ResultSet` can be an optional step (however, not closing the `ResultSet` can tie up resources when using some databases, such as Oracle). In ADO.NET you cannot use the connection for anything else until you have closed the data reader:

```
rdr.Close();
```

In C#, employ the `using` statement to simplify the code and implicitly call `Dispose` to close the connection:

```
using ( SqlConnection conn = new SqlConnection( cstr ) )
{
    conn.Open();
    ...
}
```

> **NOTE** *The data reader classes do not implement* `IDisposable`, *so you cannot employ a* `using` *statement in those cases.*

Most of the time, ADO.NET providers close the connection in their finalizer if you have not already done so. However, as garbage collection is inherently unpredictable, it would be unwise to rely on it for such a precious resource.

Using `CommandBehavior.CloseConnection`

You will find many examples that use the following form of `ExecuteReader`:

```
SqlDataReader rdr =
    cmd.ExecuteReader( CommandBehavior.CloseConnection );
```

This form of `ExecuteReader` specifies that the connection is to be closed when the data reader is closed. You might use `CommandBehavior.CloseConnection` in situations where you open a connection, execute a single SQL statement, read the results, and then close the connection. Listing 6-2 is an example of this.

Listing 6-2. Typical Usage of `CommandBehavior.CloseConnection` *(Do Not Do This)*

```
string cstr = @"Server=.\VSdotNET;"
    + @"Database=Chapter6;"
    + @"Integrated Security=SSPI";
SqlConnection conn = new SqlConnection( cstr );

conn.Open();
SqlCommand cmd
    = new SqlCommand( "select * from dotnetlanguages", conn );
SqlDataReader rdr
    = cmd.ExecuteReader( CommandBehavior.CloseConnection );

while ( rdr.Read() )
{
    System.Console.WriteLine( "{0}", rdr.GetString( 1 ) );
}

rdr.Close();
```

Examining this code reveals that there is no guarantee that the connection will be closed when an error occurs. This is not acceptable in a production application. Your next thought might be to ensure the reader gets closed by adding a `try...catch` block. Even if you do this there is still a window between the connection being opened and the reader being created, which could suffer a failure and result in the connection being left open. You could add another `try...catch` block around the whole thing to ensure the connection gets closed. At this point you have added a `try...catch` block to close the connection, just to avoid having to explicitly close the connection!

Forget `CommandBehavior.CloseConnection` and employ a `using` statement to ensure the connection gets closed. This results in much cleaner and more maintainable code (see Listing 6-3).

Listing 6-3. Employing a using *Statement to Ensure the Connection Gets Closed*

```
string cstr = @"Server=.\VSdotNET;"
    + @"Database=Chapter6;"
    + @"Integrated Security=SSPI";
using ( SqlConnection conn = new SqlConnection( cstr ) )
{
    conn.Open();

    SqlCommand cmd =
        new SqlCommand( "select * from dotnetlanguages", conn );
    SqlDataReader rdr = cmd.ExecuteReader();

    while ( rdr.Read() )
    {
        System.Console.WriteLine( "{0}", rdr.GetString( 1 ) );
    }
    rdr.Close();
}
```

Using Command Parameters

ADO.NET does not have separate objects that parallel JDBC's `Statement` and `PreparedStatement`. The command object you have already seen can be used with parameters.

When using providers other than the SQL Server managed provider, parameters look much as they do in JDBC. The SQL statement itself is passed to the command with question marks (?)where the parameters are to be substituted:

```
string selstr =
    "select vendor from dotnetlanguages where name = ?";
OleDbCommand cmd = new OleDbCommand( selstr, conn );
```

In the SQL Server managed provider, the parameters are named and pre-ceded by an at (@) symbol:

```
string selstr =
    "select vendor from dotnetlanguages where name = @name";
SqlCommand cmd = new SqlCommand( selstr, conn );
```

For each parameter, you must specify the type by creating an instance of the appropriate parameter class, such as `SqlParameter` or `OleDbParameter`. With the SQL Server managed provider, the name identifies the parameter within the SQL statement:

```
SqlParameter name =
    cmd.Parameters.Add( "@name", SqlDbType.NVarChar, 15 );
```

In all other cases, the order they are added to the command's `Parameters` property determines the question mark placeholder to which they correspond. The parameter name is irrelevant, although it must be specified:

```
OleDbParameter name =
    cmd.Parameters.Add( "@name", OleDbType.VarChar, 15 );
```

The `Value` property of the parameter object is set to the desired value prior to each call.

```
name.Value = textBox1.Text;
```

Using Stored Procedures

In JDBC you use the `Connection.prepareCall` method to specify the stored proce-dure and its parameters and may use question marks (?) as you would in `prepareStatement`. You can then specify which are output or input/output param-eters by invoking methods on the `CallableStatement` returned by `prepareCall`.

In ADO.NET you create the command object specifying only the stored pro-cedure name in place of the SQL statement. You then set the `CommandType` property of the command to `StoredProcedure`. You instantiate the parameter objects as you would with a regular command, but by specifying its `Direction` property you can indicate that a parameter is output or input/output. You can also designate one parameter as the return value of the stored procedure by spec-ifying a `Direction` of `ParameterDirection.ReturnValue`.

Using the following SQL Server stored procedure in the button handler of a WinForms Form demonstrates this:

```
CREATE PROCEDURE dbo.QueryVendor
(
    @name nvarchar(15),
    @vendor nvarchar(15) = NULL OUTPUT
)
AS
    SELECT @vendor = vendor
    FROM dotnetlanguages
    WHERE name = @name

    RETURN @@ROWCOUNT
```

Listing 6-4 is equivalent to the SQL used previously, but it uses parameters exclusively and demonstrates input, output, and return value parameter types.

Listing 6-4. Invoking a Stored Procedure with Parameters

```
private void button1_Click(object sender, System.EventArgs e)
{
    string cstr = @"Server=.\VSdotNET;"
        + @"Database=Chapter6;"
        + @"Integrated Security=SSPI";
    using ( SqlConnection conn = new SqlConnection( cstr ) )
    {
        conn.Open();

         // stored procedure name is used in place of SQL statement
         // and type is set to stored procedure
        SqlCommand cmd = new SqlCommand( "QueryVendor", conn );
        cmd.CommandType = CommandType.StoredProcedure;

        // input parm
        SqlParameter name =
            cmd.Parameters.Add( "@name", SqlDbType.NVarChar, 15 );
        name.Value = textBox1.Text;

        // output parm
        SqlParameter vendor =
            cmd.Parameters.Add( "@vendor",
                                    SqlDbType.NVarChar,
                                    15 );
        vendor.Direction = ParameterDirection.Output;
```

```
        // return value
        SqlParameter rowCount =
            cmd.Parameters.Add( "@rowCount", SqlDbType.Int );
        rowCount.Direction = ParameterDirection.ReturnValue;

        // use this since we return no rows - just parms
        cmd.ExecuteNonQuery();

        if ( (int)rowCount.Value > 0 )
        {
            MessageBox.Show( this,
                             textBox1.Text +
                             " is available from " +
                             vendor.Value );
        }
        else
        {
            MessageBox.Show( this,
                             textBox1.Text +
                             " is not available yet" );
        }
    }
}
```

If your stored procedure returns rows as well as output or return parameters, be aware that the values of the parameters will not be available until the data reader has been closed.

Using DataSets

Although the data reader is the closest analogy to a JDBC ResultSet, Microsoft is promoting the DataSet as the normal way to use ADO.NET. A DataSet has some of the characteristics of a JDBC CachedRowSet in that it provides a way to assemble a set of data and then disconnect from the database before processing it, possibly remotely.

The Simple Query Revisited

Re-create the example from the start of this chapter using a DataSet instead of a data reader to show how they differ:

1. First, create the `DataSet` to hold the data:

```
DataSet dset = new DataSet();
```

2. The connection is then obtained as in the previous example, but instead of creating a `SqlCommand`, a `SqlDataAdapter` is instantiated using the desired SQL statement:

```
SqlDataAdapter da =
    new SqlDataAdapter( "select * from dotnetlanguages", conn );
```

3. Next use the `DataAdapter` to populate a `DataTable` in the `DataSet`. A name is given to the table in the `DataSet`, but it does not need to match the name of the original table in the database:

```
da.Fill( dset, "languages" );
```

4. Now, by placing the data in a `DataSet` you can safely close the connection before processing the data:

```
using ( SqlConnection conn = new SqlConnection( cstr ) )
{
    conn.Open();

    SqlDataAdapter da =
        new SqlDataAdapter( "select * from dotnetlanguages",
                            conn );
    da.Fill( dset, "languages" );
}
```

5. Now iterate the rows in the table and access the required fields by name. Specify the name for the `DataTable` that you used in the `Fill` method:

```
foreach( DataRow dr in dset.Tables[ "languages" ].Rows )
{
    System.Console.WriteLine( "{0}", dr[ "name" ] );
}
```

Beyond the Single Table DataSet

In the preceding example you were able to read the results of a query into a collection and close the connection before processing it. However, a DataSet can do more than store the results of a single query.

Tables

A DataSet is made up of a collection of DataTable objects. In the example, you added one DataTable. If the application required several queries to be run against a database they could all have been placed in the same DataSet. A DataSet can be serialized and transmitted to a remote client for processing. Sending a single DataSet containing all the necessary tables is more efficient than sending several individual ones.

One great feature of using a DataSet is that the tables can come from several databases. It is possible to assemble all the data a client needs into one DataSet and then send all of the data in a single transfer, isolating the client program from the details of how the tables are mapped to databases.

Relationships

Having multiple tables in a DataSet is useful, but what if you do not know exactly which rows you need at the time you make the query? You could extract all the data you might need (providing it is constrained enough) and process it later, but then you lose the power of SQL to select the data you need. With a little additional logic, you can model the relationships between the tables in the DataSet and use them when you process the data.

Imagine the following scenario. You have a WinForms application that displays the language skills of a development team. By using the application, you can discover the languages a given developer knows or see which developers are proficient in a given language. This is a *many-to-many* relationship typically modeled using a simple link table. Figure 6-10 shows the design view of the developers' table, and Figure 6-11 shows the design view of the link table, called devlanglink.

Figure 6-10. The design view of the developers' table

Figure 6-11. The design view of the link table, `devlanglink`

Notice that `devlanglink` has a composite primary key. To set this while designing the table in Visual Studio .NET, select both rows, right-click to open the context menu, and then select Set Primary Key, as shown in Figure 6-12.

Figure 6-12. Creating the composite primary key for `devlanglink`

You can extract the list of languages for a developer using the following:

```
SELECT l.name from dotnetlanguages as l
JOIN devlanglink as k ON l.id = k.langid
JOIN developers as d ON d.empno = k.empno
WHERE d.name = @devname
```

You can obtain the developers who know a language using this:

```
SELECT d.name from developers as d
JOIN devlanglink as k ON d.empno = k.empno
JOIN dotnetlanguages as l ON l.id = k.langid
WHERE l.name = @langname
```

Unfortunately, this requires a trip to the database each time. Over a slow connection the user interface would respond slowly, and it would be impossible to work offline. If you extract the data into a `DataSet`, you can use relationships between the `DataTables` to obtain the subsets without contacting the database each time.

The following example demonstrates how you can use `DataSet` relationships:

1. Create a new Windows Application project called **RelationExample**. Modify the form so that its `Text` property reads **Relation Example** and its `Size` is **500, 300**.

2. Drag a `Panel` control from the Toolbox onto the form. Set its `Dock` property to **Bottom** and its `Size` to **492, 50**. Drag a `Button` control onto the `Panel`. Set its name to **loadDataButton**, its `Text` property to **Load Data**, and its `Dock` property to **Fill**.

3. Drag a `TabControl` control onto the form and set its `Dock` property to **Left**.

4. Drag a `Splitter` control onto the form. It should dock to the right of the `TabControl` control.

5. Drag a `TextBox` control onto the form and set its `Name` to **listTextBox**. Set its `MultiLine` property to **true**, its `Dock` property to **Fill**, and clear its `Text` property.

6. Select the `TabControl` and open its `TabPages` collection by clicking on the button with the ellipsis (. . .). You will see the usual collection dialog box. Click Add to add a `TabPage` to the collection. Set the page's name to **languageTabPage** and its `Text` property to **language**. Click Add again and this time set the name to **developerTabPage** and the `Text` property to **developer**. Click OK to close the dialog box.

7. Select the Language tab.

8. Drag a `Label` control onto the tab. Position it near the top of the page. Change its `Text` property to **Choose a language**.

9. Drag a `ComboBox` control onto the tab and position it below the `Label`. Change the name of the `ComboBox` to **languageComboBox** and clear its `Text` property.

10. Drag a `Button` control onto the tab. Set its name to **languageButton** and its `Text` property to **Which developers know this language?** Change the

Dock property to **Bottom** and then increase the Size property to **192, 50** so that all the text is visible.

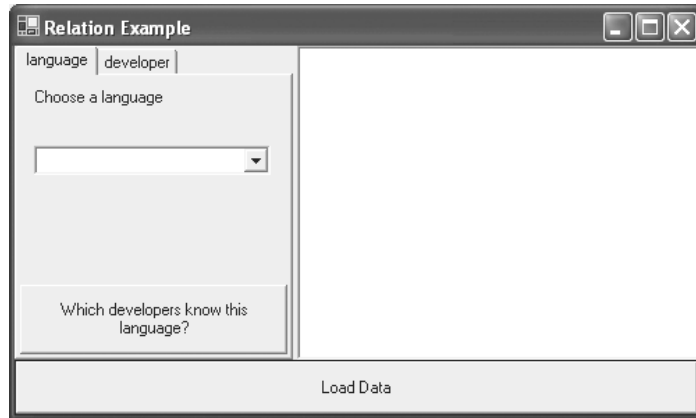11. The form should now look like that shown in Figure 6-13.

Figure 6-13. The form with the Language tab completed

12. Select the Developer tab.

13. Drag a Label control onto the tab. Position it near the top of the page. Change its Text property to **Choose a developer**.

14. Drag a ComboBox control onto the tab and position it below the Label. Change the name of the ComboBox to **developerComboBox** and clear its Text property.

15. Drag a Button control onto the tab. Set its name to **developerButton** and its Text property to **What languages do they know?** Change the Dock property to **Bottom** and then increase the Size property to **192, 50** so that all the text is visible.

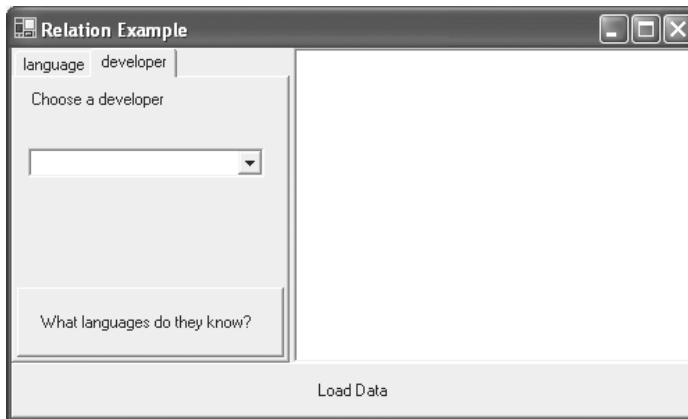16. The form should now look like that shown in Figure 6-14.

Figure 6-14. The form with the Developer tab completed

17. Double-click loadDataButton to create a button click handler. Change the code to that shown in Listing 6-5. Start by populating the DataSet with rows from the dotnetlanguages, developers, and devlanglink tables. To set up the relationships, create DataRelation objects and add them to the Relations property of the DataSet. A DataRelation specifies how columns in one table relate to columns in another table. You specify the columns as DataColumn objects.

Listing 6-5. Loading the Data into the DataSet *and Setting Up the* DataRelation *Objects*

```
const string connstr = @"Server=.\VSdotNET;"
    + @"Database=Chapter6;"
    + @"Integrated Security=SSPI";

private DataSet ds = null;

private void loadDataButton_Click(object sender,
                                  System.EventArgs e)
{
    ds = new DataSet();

    using ( SqlConnection conn = new SqlConnection( connstr ) )
    {
        conn.Open();

        // get the raw data
```

```
            SqlDataAdapter da1
                = new SqlDataAdapter( "select * from dotnetlanguages",
                                        conn );
            da1.Fill( ds, "dotnetlanguages" );

            SqlDataAdapter da2
                = new SqlDataAdapter( "select * from developers",
                                        conn );
            da2.Fill( ds, "developers" );

            SqlDataAdapter da3
                = new SqlDataAdapter( "select * from devlanglink",
                                        conn );
            da3.Fill( ds, "devlanglink" );

            // create the relationships

            ds.Relations.Add( "langlink",
                ds.Tables[ "dotnetlanguages" ].Columns[ "id" ],
                ds.Tables[ "devlanglink" ].Columns[ "langid" ] );

            ds.Relations.Add( "devlink",
                ds.Tables[ "developers" ].Columns[ "empno" ],
                ds.Tables[ "devlanglink" ].Columns[ "empno" ] );

            // establish data binding

            languageComboBox.DataSource = ds.Tables[ "dotnetlanguages" ];
            languageComboBox.DisplayMember = "name";

            developerComboBox.DataSource = ds.Tables[ "developers" ];
            developerComboBox.DisplayMember = "name";
        }
    }
```
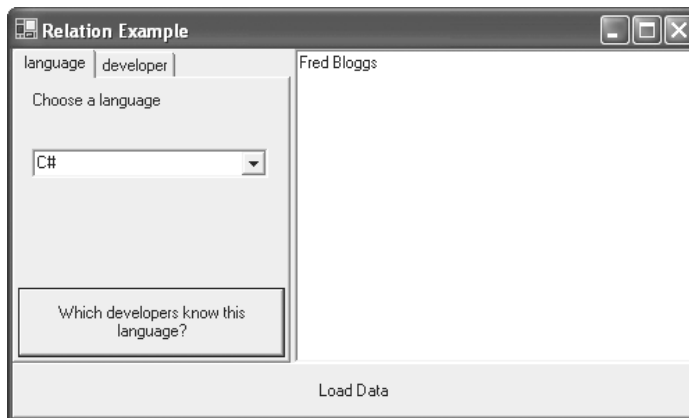
18. Now you can use these relationships to filter the rows in the `DataSet` and extract the desired rows. Double-click the `languageButton` on the Languages tab to create a button click handler. Enter the code as shown in Listing 6-6. Given a row from the `dotnetlanguages` table, you can use the `GetChildRows` method to locate the rows it maps to in the `devlanglink` table. Then iterate those rows and use the `GetParent` method to locate the related developers. Figure 6-15 shows how this appears in the application.

Listing 6-6. Using the Relationships in the DataSet *to List Developers Given a Language*

```
private void languageButton_Click(object sender,
                                  System.EventArgs e)
{
    if ( ds == null )
    {
        MessageBox.Show( this, "You must load the data first" );
    }
    else
    {
        listTextBox.Text = "";
        // languageComboBox combobox is databound to the
        // languages DataTable
        DataRow lang
            = ( (DataRowView)languageComboBox.SelectedValue ).Row;
        foreach( DataRow row in lang.GetChildRows( "langlink" ) )
        {
            DataRow dev = row.GetParentRow( "devlink" );
            listTextBox.Text += dev[ "name" ].ToString() + "\r\n";
        }
    }
}
```
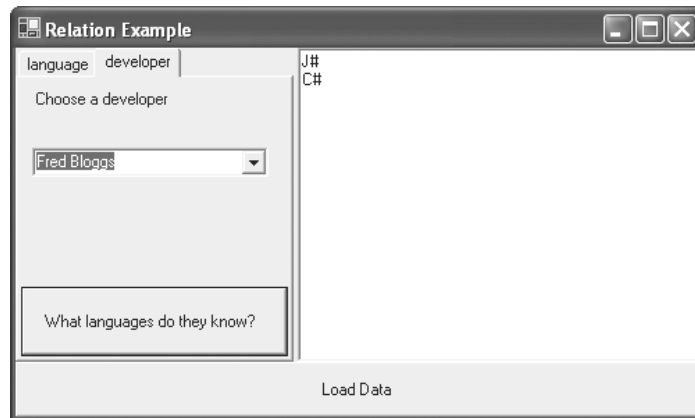

Figure 6-15. Using relationships to list developers who know a given language

19. Similarly, you can use these relationships to locate the languages a given developer knows. Double-click the developerButton on the Developer tab and enter the code shown in Listing 6-7. Figure 6-16 shows the result of implementing this button click handler.

Listing 6-7. Using the Relationships in the DataSet *to List Languages Given a Developer*

```
private void developerButton_Click(object sender,
                        System.EventArgs e)
{
    if ( ds == null )
    {
        MessageBox.Show( this, "You must load the data first" );
    }
    else
    {
        listTextBox.Text = "";
        // developerComboBox combobox is databound to the
        // developers DataTable
        DataRow dev
            = ( (DataRowView)developerComboBox.SelectedValue ).Row;
        foreach( DataRow row in dev.GetChildRows( "devlink" ) )
        {
            DataRow lang = row.GetParentRow( "langlink" );
            listTextBox.Text += lang[ "name" ].ToString() + "\r\n";
        }
    }
}
```


Figure 6-16. Using relationships to list the languages a given developer knows

20. Build and run the application.

Most examples of relationships use the equivalent of a two-way join. I deliberately chose the equivalent of a three-way join to show that it is almost as easy to accomplish.

By setting up the relationships between tables in a DataSet, you can avoid potentially costly roundtrips to the database, and you can even operate in situations where the database is unavailable.

Completing the Roundtrip

So far you have used DataSets to hold the results of queries, but they are not restricted to read-only operations. Like a CachedRowSet, a DataSet can be used to update a database, too.

Updates

When changes are made to a DataSet, they are tracked so that at some point in the future they may be reconciled with the original database. This occurs when you invoke the DataAdapter object's Update method. Just like the Fill method, the table's name is specified to indicate which one to update. The DataAdapter need not be the same instance used to populate the DataSet, and in most cases it will not be.

Take your developers' table from the previous example and add a new developer. Loading the data into the DataSet is the same as before:

```
SqlDataAdapter da =
    new SqlDataAdapter( "select * from developers", conn );
da.Fill( ds, "developers" );
```

At this point close the connection to the database.

When you are ready to add a new row, create a DataRow with the same columns as those in the DataTable. Fortunately, the DataTable has a NewRow method that does this for you. Then set the values for the fields and add the row to the Rows collection:

```
DataRow row = ds.Tables[ "developers" ].NewRow();
row[ "empno" ] = ++empno;
row[ "name" ] = developerTextBox.Text;
ds.Tables[ "developers" ].Rows.Add( row );
```

You edit the values of existing rows in the DataSet in a similar way.

Once you are ready to commit the changes to the database, you need to reconnect:

```
using ( SqlConnection conn = new SqlConnection( connstr ) )
{
    conn.Open();

    SqlDataAdapter da =
        new SqlDataAdapter( "select * from developers", conn );
```

So far, you have only supplied a select statement, and you need some combination of insert, update, and delete statements to make the required changes to the database. The SqlDataAdapter has properties that take a SqlCommand for each of these. In this example where you are adding the row, you could do the following:

```
da.InsertCommand = new SqlCommand(
  "INSERT INTO developers( empno, name ) values ( @empno, @name )",
  conn );
da.InsertCommand.Parameters.Add( "@empno",
                                 SqlDbType.Int,
                                 4,
                                 "empno" );
da.InsertCommand.Parameters.Add( "@name",
                                 SqlDbType.NVarChar,
                                 15,
                                 "name" );
```

which creates the insert command and maps the appropriate columns to the parameters.

To process updates and deletes, add the appropriate SqlCommand objects to the UpdateCommand and DeleteCommand properties of the SqlDataAdapter. The other alternative in this situation is to create a SqlCommandBuilder on the SqlDataAdapter that constructs the commands automatically:

```
SqlCommandBuilder cb = new SqlCommandBuilder( da );
```

This only works for tables populated by simple single table selects but eliminates all the parameter mapping code when it is an option.

Whichever method you use to populate the commands, your ultimate goal is to have the changes made to the DataSet reflected in the database. Accomplish this by invoking the DataAdapter object's Update method:

```
da.Update( ds, "developers" );
```

Invoke `Update` for each table changed. If the tables came from separate databases, you need to use a `DataAdapter` connected to the appropriate database for each table.

Conflicts

Running a single copy of the preceding example will not cause any conflicts between the `DataSet` and the database, but if you launch multiple copies you will soon create duplicate employee numbers. This is a classic optimistic locking problem, and the strategies for dealing with it and their shortcomings are well known. It is up to you to detect and handle conflicting updates according to the strategy you feel is most appropriate to your application.

At first glance this optimistic locking stuff looks ugly, but let me put it in perspective. Whenever you display an ASP.NET page to a user that allows them to update information, you do not usually lock the database rows waiting for a response. You must expect that when you get the response the underlying database rows may have changed in a manner that renders the updates made by the user obsolete. Your reaction may simply be to resend the original update page with the new data, but you deal with the situation somehow.

You can control the types of changes that are applied in a given `Update` invocation by extracting the rows with the desired change and only passing them to `Update`. Specify the rows by selecting out the `DataRow` instances with a given `DataViewRowState` such as `Added`, `Deleted`, or `ModifiedCurrent`. This could be important to satisfy constraints on the rows in the database, such as when you delete a row and then add a new one with the same unique key.

Constraints

Just as you can specify foreign key constraints and unique constraints in a database, you can specify them in a `DataSet`. This can help you validate data in a client, but remember that it only applies to the subset of rows in the `DataSet`, so you may still get errors when you invoke `Update` against the complete table.

You apply foreign key constraints by creating an instance of `ForeignKeyConstraint` and adding it to the `Constraints` property of the `DataTable`. Similarly, you apply unique constraints by creating an instance of `UniqueConstraint` and adding it to the same property. In both cases, you may specify multiple columns.

Understanding Pooling

Connection pooling in JDBC is typically provided by specialized class libraries that exploit the façade pattern to provide pooled connection objects. When these connection objects are closed, they do not close the underlying database connection but rather return it to the pool.

The SQL Server managed provider contains its own pooling mechanism, making the specialized class library unnecessary. Should pooling be undesirable in a given application it can be disabled in the connection string by specifying `Pooling='false'`.

The OLE DB and ODBC providers do not provide pooling themselves but rely on existing pooling mechanisms in the OLE DB or ODBC layers.

Implementing Data-Bound Controls

The WinForms and WebForms discussion touched briefly on the subject of data-bound controls using arrays as the data source. Most of the time a data-bound control will take its data from a `DataTable`. These controls save the programmer from having to map the data from the data source into the control explicitly.

Using Simple Data Binding

Simple data binding applies to controls such as combo boxes and list boxes. You must specify both the `DataTable` within the `DataSet` and the column within that `DataTable` to be used as the display value:

```
developersListBox.DataSource = ds.Tables[ "developers" ];
developersListBox.DisplayMember = "name";
```

Using Grid Controls

Both WinForms and WebForms have powerful grid controls that exploit data binding to display data.

WinForms DataGrid

WinForms has the `DataGrid` control to display a `DataTable`. This is the closest equivalent to a Swing `JTable` in WinForms, with the `DataTable` filling the role of

the `TableModel`. Add a `DataGrid` to a `Form` in the usual way by dragging and drop-
ping from the Toolbox. Then make a `DataTable` the `DataSource` for the `DataGrid`:

```
dataGrid1.DataSource = ds.Tables[ "developers" ];
```

The `DataGrid` supports multiple columns, so there is no need to specify
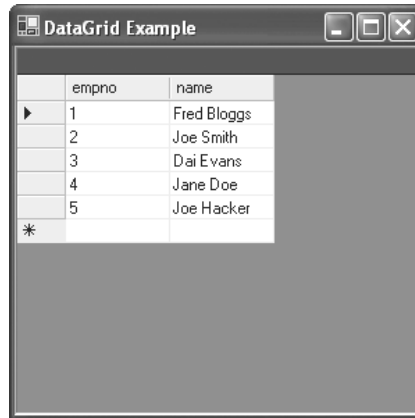a value for `DisplayMember`. Figure 6-17 shows how this would look.

Figure 6-17. The WinForms `DataGrid` *control*

By default a `DataGrid` supports unrestricted editing of the data. All cells are
editable and rows can be added and deleted freely. The `DataTable` is changed by
these operations, but you must provide code to update the database to make the
changes permanent.

If you only want to display the data, without supporting changes, set the
`ReadOnly` property of the `DataGrid` to `true`. To set an individual column to read-
only, you must locate the appropriate `DataGridColumnStyle` instance in the
`TableStyles` property of the `DataGrid` and set its `ReadOnly` property to `true`.

A `DataGrid` can also display a `DataSet`, allowing traversal of any relationships
that have been added between the tables. The initial table is established by set-
ting the `DataMember` property. This is best suited to hierarchical data, where you
want to allow *drill-down* traversal. The `DataSet` used earlier with a many-to-many
relationship does not behave as you might expect.

Imagine you want to view the developers with an option to see their lan-
guage skills. Create the following example:

1. Create a new Windows Application project called
    DataGridRelationExample.

2. Drag a `DataGrid` control from the Toolbox onto the form and set its `Dock`
    property to **Fill**. Set its `ReadOnly` property to **true**.

3. Double-click the form to create a form load handler. Change the code to that shown in Listing 6-8.

Listing 6-8. Using Relationships in a `DataSet` *with a WinForms* `DataGrid`

```
const string connstr = @"Server=.\VSdotNET;"
    + @"Database=Chapter6;"
    + @"Integrated Security=SSPI";

private DataSet ds = null;

private void Form1_Load(object sender, System.EventArgs e)
{
    ds = new DataSet();

    using ( SqlConnection conn = new SqlConnection( connstr ) )
    {
        conn.Open();

        SqlDataAdapter da1
            = new SqlDataAdapter( "select * from developers", conn );
        da1.Fill( ds, "developers" );

        SqlDataAdapter da2
            = new SqlDataAdapter( "SELECT k.empno, l.name " +
                                  "FROM dotnetlanguages l " +
                                  "JOIN devlanglink k " +
                                  "ON l.id = k.langid",
                                  conn );
        da2.Fill( ds, "languages" );

        // create the relationship

        ds.Relations.Add( "language",
                    ds.Tables[ "developers" ].Columns[ "empno" ],
                    ds.Tables[ "languages" ].Columns[ "empno" ] );

        dataGrid1.DataSource = ds;
        dataGrid1.DataMember = "developers";
    }
}
```

4. Build and run the application. The list of developers appears as in Figure 6-18. The `DataGrid` shows a plus sign (+) in the left margin that

you can expand (by clicking it) to follow the relation to the languages for that developer. The languages are shown with the context of the parent, as shown in Figure 6-19.
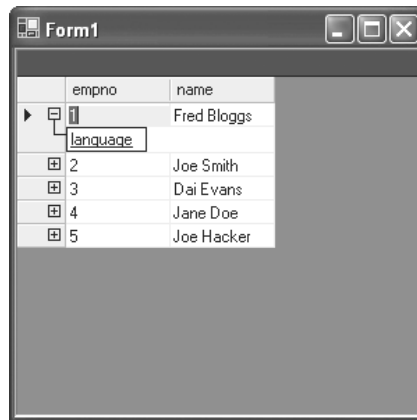

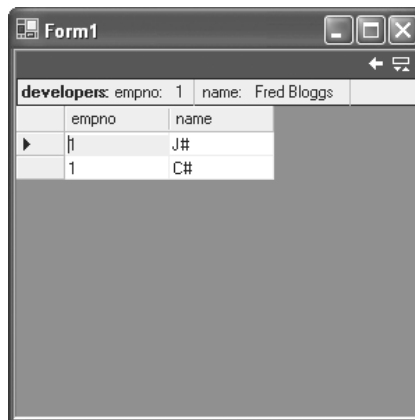*Figure 6-18. A* DataGrid *showing a relationship*

Figure 6-19. Displaying the child rows of a relationship in a WinForms DataGrid

WebForms DataGrid

ASP.NET WebForms also have a DataGrid control. Binding the control to a DataTable is much the same as in WinForms except that once the DataSource and DataMember properties have been set, you must invoke the DataBind method:

```
DataGrid1.DataSource = ds;
DataGrid1.DataMember = "developers";
DataGrid1.DataBind();
```

In contrast to the WinForms version, the WebForms `DataGrid` is read-only by default. It also does not seem to have any built-in support to traverse relationships. You can configure many visual aspects of the DataGrid. Figure 6-20 shows how it looks "out of the box" on a WebForms page.
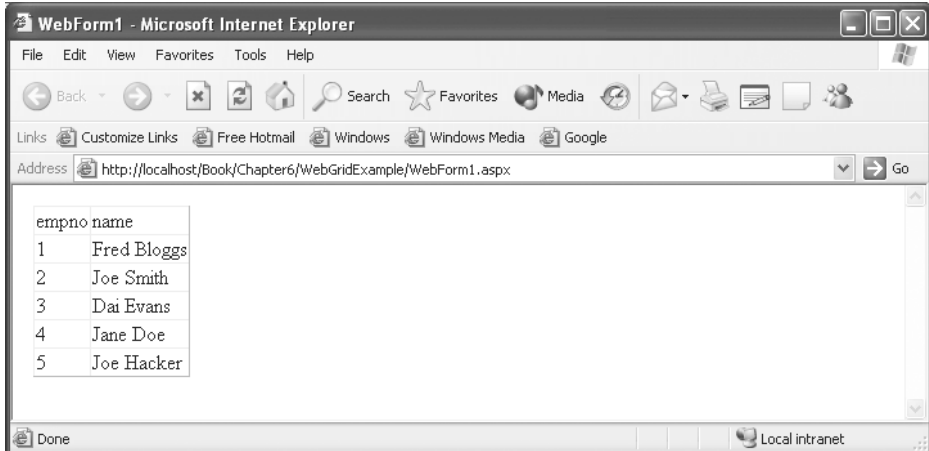

Figure 6-20. The WebForms `DataGrid`

Adding support for editing or even selecting the rows is not a matter of flipping a read-only flag. You must set up handlers for the events and register them with the control. There are no defaults that operate against the `DataSet`, so you must provide all the logic yourself. The following example shows how you achieve this:

1. Create a new ASP.NET Web Application project called
   WebGridEditExample.

2. Drag a `DataGrid` control from the Toolbox onto the Web form.

3. Switch to code view and add the code to populate the `DataGrid`, as
   shown in Listing 6-9. Notice that this includes changes to the `Page_Load`
   method.

Listing 6-9. Populating the `DataSet` *and Binding It to the WebForms* `DataGrid`

```
private const string connstr = @"Server=.\VSdotNET;"
    + @"Database=Chapter6;"
    + @"Integrated Security=SSPI";

private DataSet ds;

private void PopulateAndBind()
{
    ds = new DataSet();

    using ( SqlConnection conn = new SqlConnection( connstr ) )
    {
        conn.Open();

        SqlDataAdapter da1 =
            new SqlDataAdapter( "select * from developers", conn );
        da1.Fill( ds, "developers" );

        DataGrid1.DataSource = ds;
        DataGrid1.DataMember = "developers";
        DataGrid1.DataBind();
    }
}

private void Page_Load(object sender, System.EventArgs e)
{
    if ( !this.IsPostBack )
    {
        PopulateAndBind();
    }
}
```

4. Switch back to the designer and select the DataGrid. Open the Property
    Builder . . . link located on the Properties panel. Select the Columns pane
    and then expand the Button Column node in the Available columns tree-
    view. Select the Edit, Update, Cancel node and press the > button to add
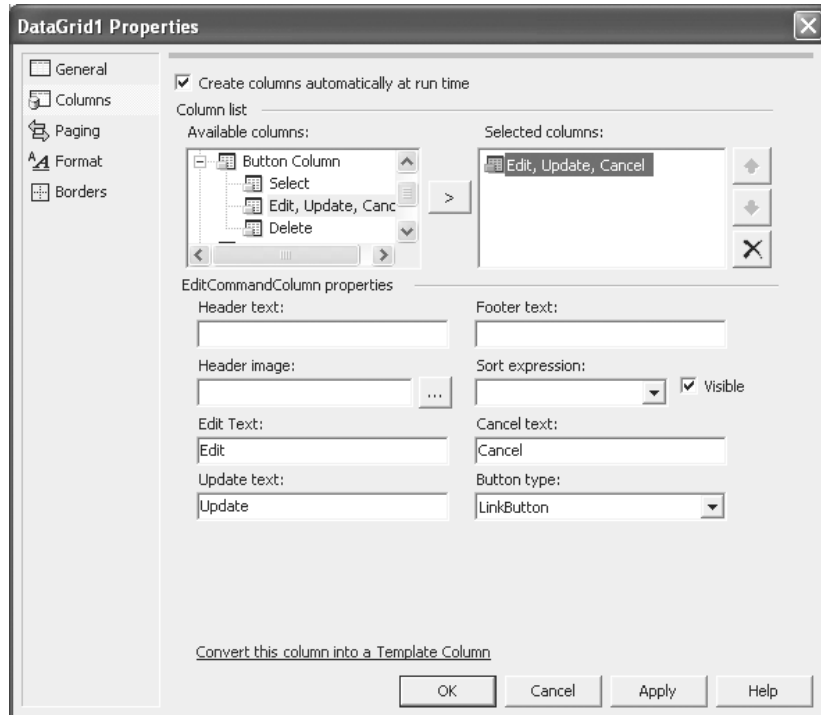   it to the selected columns list as shown in Figure 6-21. Click OK.

Figure 6-21. Adding the update commands column to the Web form

5. Now add the handlers for the Edit, Update, and Cancel events as shown in Listing 6-10. The Edit and Cancel handlers set the DataGrid object's EditItemIndex property to the selected row and –1, respectively. Unless you rebind to the DataSet after setting this value, it does not work correctly. Use the PopulateAndBind convenience function from Listing 6-9 to re-populate and rebind the DataSet. The Update handler must obtain the modified values from the DataGrid object's cell controls.

Listing 6-10. The Edit, Cancel, *and* Update *Handlers for the WebForms* DataGrid

```
public void DataGrid1_Edit(Object sender, DataGridCommandEventArgs E)
{
    DataGrid1.EditItemIndex = (int)E.Item.ItemIndex;
    PopulateAndBind();
}
```

```
public void DataGrid1_Cancel(Object sender, DataGridCommandEventArgs E)
{
     DataGrid1.EditItemIndex = -1;
     PopulateAndBind();
}


public void DataGrid1_Update(Object sender, DataGridCommandEventArgs E)
{
    int row = (int)E.Item.ItemIndex;

    int empno =
        Int32.Parse( ((TextBox)E.Item.Cells[1].Controls[0]).Text );
    String name = ((TextBox)E.Item.Cells[2].Controls[0]).Text;

    ds = new DataSet();

    using ( SqlConnection conn = new SqlConnection( connstr ) )
    {
        conn.Open();

        SqlDataAdapter da1
            = new SqlDataAdapter( "select * from developers",
                                    conn );
        da1.Fill( ds, "developers" );

        SqlCommandBuilder cb = new SqlCommandBuilder( da1 );

        DataRow dr = ds.Tables[ "developers" ].Rows[ row ];
        dr[ "empno" ] = empno;
        dr[ "name" ] = name;

        da1.Update( ds, "developers" );

        DataGrid1.EditItemIndex = -1;
        DataGrid1.DataSource = ds.Tables[ "developers" ];
        DataGrid1.DataBind();
    }
}
```

6. Then register the handlers with the control using the Events panel of the Properties window for the DataGrid control. Switch the Properties window to Events by selecting the Events icon (which looks like a lightning flash). Then connect the events to the methods, as shown in Figure 6-22.
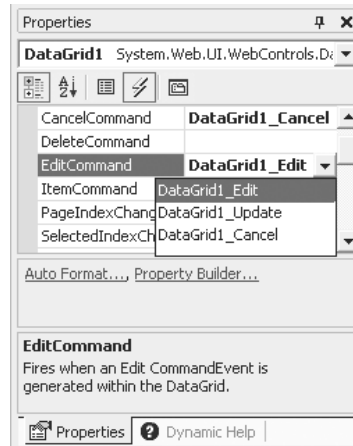
Figure 6-22. The Events panel of the Properties window

7. If you are connecting to a SQL Server database using a user ID and password in your connection string, you can build and run the application. If you need to use Windows authentication, there are a few additional steps.

8. Edit the project's web.config file to add `<identity impersonate="true" />` following the `<authentication>` element. This tells ASP.NET to impersonate the identity of the user connecting to the application so that you can authenticate to SQL Server/MSDE.

9. Open the Internet Information Services administrative tool from the Control Panel. Locate the virtual root for the project and open its Properties panel. Select the Directory Security tab. Uncheck the anonymous access box and ensure that the Windows-integrated authentication box is checked. This makes IIS require user authentication for the application.

10. Now you can build and run the application. Figure 6-23 shows this updatable version of the WebForms `DataGrid` on a Web page.
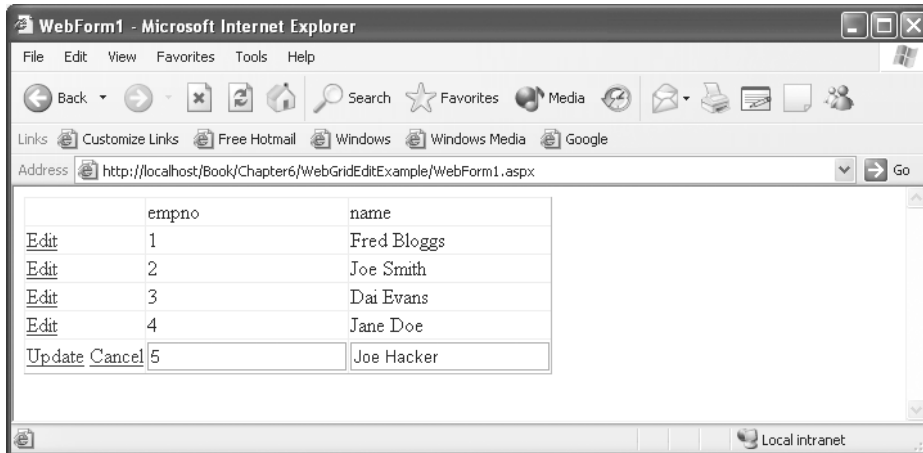
Figure 6-23. The updatable version of the DataGrid

The WebForms version of the DataGrid is not as powerful as the WinForms version, and it requires you to add a large amount of your own code to achieve what is basic functionality in the WinForms DataGrid.

Comparing ADO.NET to the Current ADO

The big difference from ADO is the same as that from JDBC: the disconnected DataSet model. ADO provides the disconnected RecordSet, which is much the same as the CachedRowSet, but neither provides the support for multiple tables that a DataSet does.

An ADO RecordSet is roughly analogous to a DataTable within a DataSet. It is marshaled by COM, which restricts it to Windows clients most of the time. A DataSet is marshaled as Extensible Markup Language (XML), which allows it to be used by non-Windows clients. Until the inner workings of a DataSet become well documented and client-side libraries appear on other platforms, I doubt you will see widespread use on non-Windows platforms.

The other big difference is the move away from the façade pattern to discrete providers with their own classes. ADO uses OLE DB providers and the façade pattern to hide the underlying implementation from developers. It remains to be seen if this is just expediency and if Microsoft will return to this in future incarnations of ADO.NET.

Summary

JDBC users will find it easy to transition to ADO.NET using the command and data reader classes following the patterns to which they have become accustomed. Anyone familiar with `CachedRowSet` will find the `DataSet` concept easy to grasp. Data binding will motivate interface developers to start using `DataSet` objects. Once you become comfortable with the disconnected model, you will start enjoying some of the other capabilities `DataSet` objects provide.