MARUDHAR KESARI JAIN COLLGE FOR WOMEN , VANIYAMBADI

PG DEPARTMENT OF COMPUTER APPLICATIONS

CLASS : II MCA

SUBJECT CODE : GCA43

SUBJECT NAME : WEB APPLICATION USING C#

SYLLABUS

UNIT - V: DATABASE ACCESS WITH COMMAND, ADAPTER AND XML

Teaching Hours: 13

ADO.NET Data Access - About the ADO.NET Example - Obtaining the Sample Database -Simple Data Access - Simple Data Update - Importing the Namespaces - Creating a Connection - The Connection String SQL - Making the Connection - Defining the Select Command - Using a Command with a DataReader - Updating Data - Using Update - Insert - and Delete Commands - Accessing Disconnected Data - Selecting Disconnected Data - Selecting Multiple Tables - Modifying Disconnected Data - Modifying and Deleting Rows - Adding Information - to a DataSet - Updating Disconnected Data - The Command Builder - Updating a DataTable - Controlling Updates - An Update Example – Using XML - XML‘s Hidden Role in .NET - XML Basics - Attributes - Comments - The XML Classes - the XML TextWriter - The XML Text Reader - Working with XML Documents - Reading an XML Document - Searching an XML Document - XML Validation – CreatingXML Schema -XSD Documents - Validating an XML File.

Chapter 13: ADO.NET Data Access

Overview

Chapter 12 introduced ADO.NET and the family of objects that provides its functionality. This family includes objects for modeling data (representing tables, rows, columns, and relations) and objects designed for communicating with a data source. In this chapter, you'll learn how to put these objects to work by creating pages that use ADO.NET to retrieve information from a database and apply changes. You'll also learn how to retrieve and manage disconnected data, and deal with the potential problems that can occur.

About the ADO.NET Examples

One of the goals of ADO.NET is to provide data access universally and with a consistent object model. That means you should be able to access data the same way regardless of what type of database you are using (or even if you are using a data source that doesn't correspond to a database). In this chapter, most examples will use the "lowest common denominator" of data access—the OLE DB data types. These types, found in the System.Data.OleDb namespace, allow you to communicate with just about any database that has a matching OLE DB provider, including SQL Server. However, when you develop a production-level application, you'll probably want to use an authentic .NET provider for best performance. These managed providers will work almost exactly the same, but their Connection and Command objects will have different names and will be located in a different namespace. Currently, the only managed provider included with .NET is SQL Server, but many other varieties are planned, and a managed ODBC driver is already available as a download from Microsoft.

In the examples in this chapter, I make note of any differences between the OLE DB and SQL providers. Remember, the underlying technical details differ, but the objects are almost identical. The only real differences are as follows:

- The names of the Connection, Command, and DataReader classes are different, to help you distinguish them.
- The connection string (the information you use to connect to the database) differs depending on the way you connect and several other factors.
- The OLE DB objects support a wider variety of data types, while the SQL Server objects only support valid SQL Server data types.

Know Your Database

It's best not to worry too much about the differences between OLE DB and SQL providers. They are similar to the differences between different OLE DB data sources, or even between differently configured installations of the same relational database product. When programming with ADO.NET, it always helps to know your database. If you have information on-hand about the data types it uses, the stored procedures it provides, and the user login you require, you'll be able to work more quickly and with less chance of error.

Obtaining the Sample Database

This chapter uses examples drawn from the pubs database, a sample database included with Microsoft SQL Server and designed to help you test database access code. If you aren't using SQL Server, you won't have this database available. Instead, you can use MSDE, the free data engine included with Visual Studio .NET. MSDE is a scaled-down version of SQL Server that's free to distribute. Its only limitations are that it's restricted to five simultaneous users and that it doesn't provide any graphical tools for creating and managing databases. To install the pubs database in MSDE or SQL Server, you can use the Transact-SQL script included with the samples for this chapter. Alternatively, you can use a different relational database engine, and tweak the examples in this section accordingly.

SQL Basics

When you interact with a data source through ADO.NET, you use the SQL language to retrieve, modify, and update information. In some cases, ADO.NET will hide some of the details for you and even generate the required SQL statements automatically. However, to design an efficient database application with the minimum amount of frustration, you need to understand the basic concepts of SQL.

SQL (Structured Query Language) is a standard data access language used to interact with relational databases. Different databases differ in their support of SQL or add additional features, but the core commands used to select, add, and modify data are common. In a database product like Microsoft SQL Server, it's possible to use SQL to create fairly sophisticated SQL scripts for stored procedures and triggers (although they have little of the power of a full object-oriented programming language). When working with ADO.NET, however, you will probably only use a few standard types of SQL statements:

- Use a Select statement to retrieve records.
- Use an Update statement to modify records.
- Use an Insert statement to add a new record.
- Use a Delete statement to delete existing records.

If you already have a good understanding of SQL, you can skip the next few sections. Otherwise, read on for a quick tour of SQL fundamentals.

Experimenting with SQL

If you've never used SQL before, you may want to play around with it and create some sample queries before you start using it in an ASP.NET site. Most database products give you some features for testing queries. If you are using SQL Server, you can try the SQL Query Analyzer, shown in Figure 13-1. This program lets you type in SQL statements, run them, and see the retrieved results, allowing you to test your SQL assumptions and try out new queries.

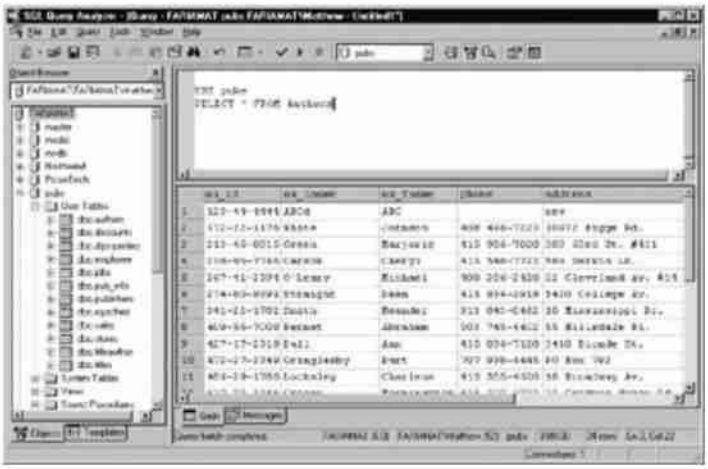

Figure 13-1: The SQL Query Analyzer

More About SQL

You might also want to look at one of the excellent online SQL references available on the Internet. Additionally, if you are working with SQL Server you can use Microsoft's thorough Books Online reference to become a database guru. One topic you might want to investigate is Join queries, which allow you to connect related tables into one set of results.

The SQL Select Statement

To retrieve one or more rows of data, you use a Select statement:

```
SELECT [columns] FROM [tables] WHERE [search_condition]
    ORDER BY [order_expression ASC | DESC]
```

This format really just scratches the surface of SQL. If you want, you can create advanced statements that use sub-grouping, averaging and totaling, and other options (such as setting a maximum number of returned rows). However, though these tasks can be performed with advanced SQL statements, they are often performed manually by your application code or built into a stored procedure in the database.

A Sample Select Statement

A typical (and rather inefficient) Select statement for the pubs database is shown here. It works with the Authors table, which contains a list of authors:

```
SELECT * FROM Authors
```

- The asterisk (*) retrieves all the columns in the table. This isn't the best approach for a large table if you don't need all the information. It increases the amount of data that has to be transferred and can slow down your server.

- The From clause identifies that the Authors table is being used for this statement.

- There is no Where clause. That means that all the records will be retrieved from the database, regardless of whether there are ten or ten million. This is a poor design practice, as it often leads to applications that appear to work fine when they are first deployed, but gradually slow down as the database grows. In general, you should *always* include a Where clause to limit the possible number of rows. Often, queries are limited by a date field (for

example, all orders that were placed in the last three months).

- There is no Order By clause. This is a perfectly acceptable approach, especially if order doesn't matter or you plan to sort the data on your own using the tools provided in ADO.NET.

Try These Examples in Query Analyzer

You can try out all the examples in this section without generating an ASP.NET program. Just use the SQL Query Analyzer or a similar query tool. In the Query Analyzer, you can type in the Select statement, and click the green start button. A table of records with the returned information will appear at the bottom of the screen.

Note that you must specify the database you want to use before using any table names. To do this, type the following Use statement first:

```
Use pubs
```

Improving the Select Statement

Here's another example that retrieves a list of author names:

```
SELECT au_lname, au_fname FROM Authors WHERE State='MI' ORDER BY
au_lname ASC
```

- Only two columns are retrieved (au_lname and au_fname). They correspond to the last and first names of the author.

- A Where clause restricts results to those authors who live in the specified state. Note that the Where clause requires apostrophes around the value you want to match (unless it is a numeric value).

- An Order By clause sorts the information alphabetically by the author's last name.

An Alternative Select Statement

Here's one last example:

```
SELECT TOP 100 au_lname, au_fname FROM Authors ORDER BY au_lname,
au_fname ASC
```

- This example uses the Top clause instead of a Where statement. The database rows will be sorted, and the first 100 matching results will be retrieved. The Top clause is useful for user-defined search operations where you want to ensure that the database is not tied up in a long operation retrieving unneeded rows.

- This example uses a more sophisticated Order By expression, which sorts authors with identical last names into a subgroup by their first names.

The Where Clause

In many respects, the Where clause is the most important part of the Select statement. You can combine multiple conditions with the And keyword, and you can specify greater than and less than comparisons by using the greater than (>) and less than (<) operators.

The following is an example with a different table and a more sophisticated Where statement. The international date format is used to compare date values.

```
SELECT * FROM Sales WHERE ord_date < '2000/01/01' AND ord_date >
'1987/01/01'
```

The SQL Update Statement

The SQL Update statement selects all the records that match a specified search expression and then modifies them all according to an update expression. At its simplest, the Update statement has the following format:

```
UPDATE [table] SET [update_expression] WHERE [search_condition]
```

Typically, you will use an Update statement to modify a single record. The following example adjusts the phone column in a single author record. It uses the unique author ID to find the correct row.

```
UPDATE Authors SET phone='408 496-2222' WHERE au_id='172-32-1176'
```

In the Query Analyzer (see Figure 13-2), this statement will return the number of rows affected, but it will not display the change. To do that, you need to request the row by performing another SQL statement.

Figure 13-2: An Update statement in the Query Analyzer

As with a Select statement, you can use an Update statement to make several changes at once or to search for a record based on several different criteria:

```
UPDATE Authors SET au_lname='Whiteson', au_fname='John'
   WHERE au_lname='White' AND au_fname='Johnson'
```

You can even use the Update statement to update an entire range of matching records. The following example modifies the phone number for every author who lives in California:

```
UPDATE Authors SET phone='408 496-2222' WHERE state='CA'
```

The SQL Insert Statement

The SQL Insert statement adds a new record to a table with the information you specify. It takes the following form:

```
INSERT INTO [table] ([column_list]) VALUES ([value_list])
```

You can provide the information in any order you want, as long as you make sure that the list of column names and the list of values correspond exactly.

```
INSERT INTO Authors (au_id, au_lname, au_fname, zip, contract)
      VALUES ('998-72-3566', 'John', 'Khan', 84152, 0)
```

This example leaves out some information, such as the city and address, in order to provide a simple example. The information shown above is the bare minimum required to create a new record in the Authors table. The new record is featured in Figure 13-3.

Figure 13-3: Inserting a new record

Remember, databases often have special field requirements that may prevent you from adding a record unless you fill in all the values with valid information. Alternatively, some fields may be configured to use a default value if left blank. In the Authors table, some fields are required and a special format is defined for the zip code and author ID.

One feature the Authors table doesn't use is an automatically incrementing identity column. This feature, which is supported in most relational database products, assigns a unique value to a specified column when you perform an insert operation. In this case, you shouldn't specify a value for the identity column when inserting the row. Instead, allow the database to choose one automatically.

Auto-Increment Fields Are Indispensable

If you are designing a database, make sure that you add an auto-incrementing identity field to every table. It's the fastest, easiest, and least error-prone way to assign a unique "identification number" to every record. Without an automatically generated identity field, you will need to go to considerable effort to create and maintain your own unique field. Often, programmers fall into the trap of using a data field for a unique identifier, such as a Social Security number or name. This almost always leads to trouble at some inconvenient time far in the future, when you need to add a person who doesn't have a SSN number (for example, a foreign national) or account for an SSN number or name change (which will cause problems for other related tables, such as a purchase order table that identifies the purchaser by the name or SSN number field). A much better approach is to use a unique identifier, and have the database engine assign an arbitrary unique number to every row automatically.

If you create a table without a unique identification column, you will have trouble when you need to

select that specific row for deletion or updates. Selecting records based on a text field can also lead to problems if the field contains special embedded characters (like apostrophes). You'll also find it extremely awkward to create table relationships.

The SQL Delete Statement

The Delete statement is even easier to use. It specifies criteria for one or more rows that you want to remove. Be careful: once you delete a row, it's gone for good!

```
DELETE FROM [table] WHERE [search_condition]
```

The following example removes a single matching row from the Authors table:

```
DELETE FROM Authors WHERE au_id='172-32-1176'
```

SQL Delete and Update commands return a single piece of information: the number of affected records. You can examine this value and use it to determine whether the operation was successful or executed as expected.

The rest of this chapter shows how you can combine the SQL language with the ADO.NET objects to retrieve and manipulate data in your web applications.

Accessing Data the Easy Way

The "easy way" to access data is to perform all your database operations manually and not worry about maintaining disconnected information. This model is closest to traditional ADO programming, and it allows you to side-step potential concurrency problems, which result when multiple users try to update information at once.

Generally, simple data access is ideal if you only need to read information or if you only need to perform simple update operations, such as adding a record to a log or allowing a user to modify values in a single record (for example, customer information for an e-commerce site). Simple data access is not as useful if you want to provide sophisticated data manipulation features, such as the ability to perform multiple edits on several different records (or several tables).

With simple data access, a disconnected copy of the data is not retained. That means that data selection and data modifications are performed separately. Your program must keep track of the changes that need to be committed to the data source. For example, if a user deletes a record, you need to explicitly specify that record using an SQL Delete statement.

Simple Data Access

To retrieve information with simple data access, follow these steps:

1. Create Connection, Command, and DataReader objects.
2. Use the DataReader to retrieve information from the database and display it in a control on a web form.
3. Close your connection.
4. Send the page to the user. At this point, the information your user sees and the information

in the database no longer have any connection, and all the ADO.NET objects have been destroyed.

Simple Data Updates

To add or update information, follow these steps:

1. Create new Connection and Command objects.

2. Execute the Command (with the appropriate SQL statement).

Importing the Namespaces

Before continuing any further, make sure you import the ADO.NET namespaces, as shown here. Alternatively, you can add these as project-wide imports by modifying your project's properties in Visual Studio .NET.

```
Imports System.Data
Imports System.Data.OleDb  ' Or System.Data.SqlClient
```

Creating a Connection

Before you can retrieve or update data, you need to make a connection to the data source. Generally, connections are limited to some fixed number, and if you exceed that number (either because you run out of licenses or because your server can't accommodate the user load), attempts to create new connections will fail. For that reason, you should try to hold a connection open for as short a time as possible. You should also write your database code inside a Try/Catch error-handling structure, so that you can respond if an error does occur.

To create a connection, you need to specify a value for its ConnectionString property. This ConnectionString defines all the information the computer needs to find the data source, log in, and choose an initial database. Out of all the examples in this chapter, the ConnectionString is the one value you might have to tweak before it works for the database you want to use. Luckily, it's quite straightforward.

```
Dim MyConnection As New OleDbConnection()
MyConnection.ConnectionString = "Provider=SQLOLEDB.1;" & _
  "Data Source=localhost;" & _
  "Initial Catalog=Pubs;User ID=sa"
```

The connection string for the SqlConnection object is quite similar, and just leaves out the Provider setting:

```
Dim MyConnection As New SqlConnection()
MyConnection.ConnectionString = "Data Source=localhost;" & _
 "Initial Catalog=Pubs;User ID=sa"
```

The Connection String

The connection string is actually a series of distinct pieces of information, separated by semicolons (;). In the preceding example, the connection string identifies the following pieces of information:

Provider This is the name of the OLE DB provider, which allows communication between ADO.NET and your database. (SQLOLEDB is the OLE DB provider for SQL.) Other providers include MSDAORA (the OLE DB provider for an Oracle database) and

Microsoft.Jet.OLEDB.4.0 (the OLE DB provider for Access).

Data Source This indicates the name of the server where the data source is located. In this case, the server is on the same computer hosting the ASP.NET site, so localhost is sufficient.

Initial Catalog This is the name of the database that this connection will be accessing. It's only the "initial" database because you can change it later, by running an SQL command or by modifying the Database property.

User ID This is used to access the database. The user ID "sa" corresponds to the system administrator account provided with databases such as SQL Server.

Password By default, the sa account doesn't have a password. Typically, in a production-level site, this account would be modified or replaced. To specify a password for a user, just add the Password settings to the connection string (as in "Password=letmein"). Note that as with all connection string settings, you don't need quotation marks, but you do need to separate the setting with a semicolon.

ConnectionTimeout This determines how long your code will wait, in seconds, before generating an error if it cannot establish a database connection. Our example connection string doesn't set the ConnectionTimeout, so the default of 15 seconds is used. Use 0 to specify no limit, which is a bad idea. That means that, theoretically, the user's web page could be held up indefinitely while this code attempts to find the server.

SQL Server Integrated Authentication

If the example connection string doesn't allow you to connect to an SQL Server database, it could be because the standard SQL Server accounts have been disabled in favor of integrated Windows authentication (in SQL Server 2000, this is the default).

- With **SQL Server authentication**, SQL Server maintains its own user account information.
- With **integrated authentication**, SQL Server automatically uses the Windows account information for the currently logged-in user. In this case, you would use a connection string with the Integrated Security option:

```
MyConnection.ConnectionString = "Provider=SQLOLEDB.1;" & _
 "Data Source=localhost;" & _
 "Initial Catalog=Pubs;Integrated Security=SSPI"
```

For this to work, the currently logged-on Windows user must have the required authorization to access the SQL database. In the case of an ASP.NET application, the "current user" is set based on IIS and web.config options. For more information, refer to Chapter 24, which discusses security.

Other Connection String Values

There are some other, lesser-used options you can set for a connection string, such as parameters that specify how long you'll wait while trying to make a connection before timing out. For more information, refer to the .NET help files (under SqlConnection or OleDbConnection).

Connection String Tips

Typically, all the database code in your application will use the same connection string. For that

reason, it usually makes the most sense to store a connection string in a globally available variable or class member.

```
Dim ConnectionString As String = "Provider=SQLOLEDB ..."
```

You can also create a Connection object with a preassigned connection string by using a special constructor:

```
Dim MyConnection As New OleDbConnection(ConnectionString)
' MyConnection.ConnectionString is now set to ConnectionString.
```

Making the Connection

Before you can use a connection, you have to explicitly open it:

```
MyConnection.Open()
```

To verify that you have successfully connected to the database, you can try displaying some basic connection information. The following example uses a label control called lblInfo (see Figure 13-4).

Figure 13-4: Testing your connection

Here's the code using a Try/Catch error-handling block:

```
' Define the ADO.NET connection object.
Dim MyConnection As New OleDbConnection(ConnectionString)

Try
   ' Try to open the connection.
   MyConnection.Open()
   lblInfo.Text = "<b>Server Version:</b> "
   lblInfo.Text &= MyConnection.ServerVersion
   lblInfo.Text &= "<br><b>Connection Is:</b> "
   lblInfo.Text &= MyConnection.State.ToString()
Catch err As Exception
   ' Handle an error by displaying the information.
   lblInfo.Text = "Error reading the database. "
   lblInfo.Text &= err.Message
Finally
   ' Either way, make sure the connection is properly closed.
   If (Not Is Nothing MyConnection) Then
      MyConnection.Close()
      lblInfo.Text &= "<br><b>Now Connection Is:</b> "
      lblInfo.Text &=  MyConnection.State.ToString()
   End If
End Try
```

Once you use the Open method, you have a live connection to your database. One of the most fundamental principles of data access code is that you should reduce the amount of time you hold a connection open as much as possible. Imagine that as soon as you open the connection, you have a live, ticking time bomb. You need to get in, retrieve your data, and throw the connection away as quickly as possible to make sure your site runs efficiently.

Closing a connection is just as easy:

```
MyConnection.Close()
```

Defining a Select Command

The Connection object provides a few basic properties that give information about the connection, but that's about all. To actually retrieve data, you need a few more ingredients:

- An SQL statement that selects the information you want
- A Command object that executes the SQL statement
- A DataReader or DataSet object to catch the retrieved records

Command objects represent SQL statements. To use a Command, you define it, specify the SQL statement you want to use, specify an available connection, and execute the command.

You can use one of the earlier SQL statements as shown here:

```
Dim MyCommand As New OleDbCommand()
MyCommand.Connection = MyConnection
MyCommand.CommandText = "SELECT * FROM Authors"
```

Or you can use the constructor as a shortcut:

```
Dim MyCommand As New OleDbCommand("SELECT * FROM Authors", _
  MyConnection)
```

The process is identical for an SqlCommand:

```
Dim MyCommand As New SqlCommand("SELECT * FROM Authors", MyConnection)
```

Using a Command with a DataReader

Once you've defined your command, you need to decide how you want to use it. The simplest approach is to use a DataReader, which allows you to quickly retrieve all your results. The DataReader uses a live connection, and should be used quickly and then closed. The DataReader is also extremely simple. It supports fast forward-only, read-only access to your results, which is generally all you need when retrieving information. Because of the DataReader's optimized nature, it provides better performance than the DataSet. It should always be your first choice for simple data access.

Before you can use a DataReader, make sure you have opened the connection:

```
MyConnection.Open()
```

To create a DataReader, you use the ExecuteReader method of the Command object:

```
' You don't need to use the New keyword because
```

```
' the Command will create the DataReader.
Dim MyReader As OleDbDataReader
MyReader = MyCommand.ExecuteReader()
```

The process is identical for the SqlDataReader:

```
Dim MyReader As SqlDataReader
MyReader = MyCommand.ExecuteReader()
```

These two lines of code define a DataReader and create it using your command. (In order for them to work, you'll need all the code we've created so far in the previous sections to define a connection and a command.)

Once you have the reader, you retrieve a row using the Read method:

```
MyReader.Read()   ' The first row in the result set is now available.
```

You can then access values in that row using the corresponding field name. The following example adds an item to a list box with the first name and last name for the current row:

```
lstNames.Items.Add(MyReader("au_lname") & ", " & MyReader("au_fname"))
```

To move to the next row, use the Read method again. If this method returns True, a row of information has been successfully retrieved. If it returns False, you have attempted to read past the end of your result set. There is no way to move backward to a previous row.

As soon as you have finished reading all the results you need, close the DataReader and Connection:

```
MyDataReader.Close()
MyConnection.Close()
```

Putting It All Together

The next example demonstrates how you can use all the ADO.NET ingredients together to create a simple application that retrieves information from the Authors table.

You can select an author record by last name using a drop-down list box, as shown in Figure 13-5. The full record is then retrieved and displayed in a simple label control, as shown in Figure 13-6.

Figure 13-5: Selecting an author

Figure 13-6: Author information

Filling the List Box

To start off, the connection string is defined as a private variable for the page class:

```
Private ConnectionString As String = "Provider=SQLOLEDB.1;" & _
 "Data Source=localhost;Initial Catalog=pubs;Integrated Security=SSPI"
```

The list box is filled after the Page.Load event occurs. Because the list box control is set to persist its viewstate information, this information only needs to be retrieved once, the first time the page is displayed. It will be ignored on all postbacks.

```
Private Sub Page_Load(sender As Object, e As EventArgs) _
  Handles MyBase.Load

    If Me.IsPostBack = False Then
        FillAuthorList()
    End If

End Sub

Private Sub FillAuthorList()

    lstAuthor.Items.Clear()

    ' Define the Select statement.
    ' Three pieces of information are needed: the unique id,
    ' and the first and last name.
    Dim SelectSQL As String
    SelectSQL = "SELECT au_lname, au_fname, au_id FROM Authors"

    ' Define the ADO.NET objects.
    Dim con As New OleDbConnection(ConnectionString)
    Dim cmd As New OleDbCommand(SelectSQL, con)
    Dim reader As OleDbDataReader

    ' Try to open database and read information.
    Try
        con.Open()
        reader = cmd.ExecuteReader()

        ' For each item, add the author name to the displayed
        ' list box text, and store the unique ID in the Value property.
        Do While reader.Read()
            Dim NewItem As New ListItem()
            NewItem.Text = reader("au_lname") & ", " & _
              reader("au_fname")
            NewItem.Value = reader("au_id")
            lstAuthor.Items.Add(NewItem)
```

```
        Loop

        reader.Close()

    Catch err As Exception
        lblResults.Text = "Error reading list of names. "
        lblResults.Text &= err.Message
    Finally
        If (Not con Is Nothing) Then
            con.Close()
        End If
    End Try

End Sub
```

Interesting Facts About this Code

- This example looks more sophisticated than the previous bite-sized snippets in this chapter, but it really doesn't introduce anything new. The standard Connection, Command, and DataAdapter objects are used. The connection is opened inside an error-handling block, so that your page can handle any unexpected errors and provide information. A Finally block makes sure that the connection is properly closed, even if an error occurs.

- The actual code for reading the data uses a loop. With each pass, the Read method is used to get another row of information. When the reader has read all the available information, this method will return False, the While condition will evaluate to False, and the loop will end gracefully.

- The unique ID (the value in the au_id field) is stored in the Value property of the list box for reference later. This is a crucial ingredient that is needed to allow the corresponding record to be queried again. If you tried to build a query using the author's name, you would need to worry about authors with the same name, and invalid characters (such as the apostrophe in O'Leary) that would invalidate your SQL statement.

Retrieving the Record

The record is retrieved as soon as the user changes the list box selection. To accomplish this effect, the AutoPostBack property of the list box is set to True, so its change events are detected automatically.

```
Private Sub lstAuthor_SelectedIndexChanged(sender As Object, _
  e As EventArgs) Handles lstAuthor.SelectedIndexChanged

    ' Create a Select statement that searches for a record
    ' matching the specific author id from the Value property.
    Dim SelectSQL As String
    SelectSQL = "SELECT * FROM Authors "
    SelectSQL &= "WHERE au_id='" & lstAuthor.SelectedItem.Value & "'"

    ' Define the ADO.NET objects.
    Dim con As New OleDbConnection(ConnectionString)
    Dim cmd As New OleDbCommand(SelectSQL, con)
    Dim reader As OleDbDataReader

    ' Try to open database and read information.
    Try
        con.Open()
        reader = cmd.ExecuteReader()
        reader.Read()
        lblResults.Text = "<b>" & reader("au_lname")
        lblResults.Text &= ", " & reader("au_fname") & "</b><br>"
        lblResults.Text &= "Phone: " & reader("phone") & "<br>"
```

```
        lblResults.Text &= "Address: " & reader("address") & "<br>"
        lblResults.Text &= "City: " & reader("city") & "<br>"
        lblResults.Text &= "State: " & reader("state") & "<br>"
        reader.Close()
    Catch err As Exception
        lblResults.Text = "Error getting author. "
        lblResults.Text &= err.Message
    Finally
        If (Not con Is Nothing) Then
            con.Close()
        End If
    End Try

End Sub
```

The process is similar to the procedure used to retrieve the last names. There are only a couple of differences:

- The code dynamically creates an SQL statement based on the selected item in the drop-down list box. It uses the Value property of the selected item, which stores the unique identifier. This is a common (and very useful) technique.

- Only one record is read. The code assumes that only one author has the matching au_id, which is reasonable as this field is unique.

What About More Advanced Controls?

This example shows how ADO.NET works to retrieve a simple result set. Of course, ADO.NET also provides handy controls that go beyond this generic level, and let you provide full-featured grids with sorting and editing. These controls are described in Chapter 15. For now, concentrate on understanding the fundamentals about ADO.NET and how it works with data.

Updating Data

Now that you understand how to retrieve data, it's not much more complicated to perform simple delete and update operations. Once again, you use the Command object, but this time you don't need a DataReader because no results will be retrieved. You also don't need an SQL Select command. Instead, you can use one of three new SQL commands: Update, Insert, or Delete.

Using Update, Insert, and Delete Commands

To execute an Update, Insert, or Delete statement, you need to create a Command object. You can then execute the command with the ExecuteNonQuery method. This method returns the number of rows that were affected, which allows you to check your assumptions. For example, if you attempt to update or delete a record and are informed that no records were affected, you probably have an error in your Where clause that is preventing any records from being selected. (If, on the other hand, your SQL command has a syntax error or attempts to retrieve information from a non-existent table, an exception will occur.)

To demonstrate how to Update, Insert, and Delete simple information, the previous example has been enhanced. Instead of being displayed in a label, the information from each field is added to a separate text box. Two additional buttons allow you to update the record (Update), or delete it (Delete). You can also insert a new record by clicking Create New, entering the information in the text boxes, and then clicking Insert New, as shown in Figure 13-7.

Figure 13-7: A more advanced author manager

The record selection code is identical from an ADO.NET perspective, but it now uses the individual text box controls.

```
Private Sub lstAuthor_SelectedIndexChanged(sender As Object, _
  e As EventArgs) Handles lstAuthor.SelectedIndexChanged

    ' Define ADO.NET objects.
    Dim SelectSQL As String
    SelectSQL = "SELECT * FROM Authors "
    SelectSQL &= "WHERE au_id='" & lstAuthor.SelectedItem.Value & "'"
    Dim con As New OleDbConnection(ConnectionString)
    Dim cmd As New OleDbCommand(SelectSQL, con)
    Dim reader As OleDbDataReader

    ' Try to open database and read information.
    Try
        con.Open()
        reader = cmd.ExecuteReader()
        reader.Read()

        ' Fill the controls.
        txtID.Text = reader("au_id")
        txtFirstName.Text = reader("au_fname")
        txtLastName.Text = reader("au_lname")
        txtPhone.Text = reader("phone")
        txtAddress.Text = reader("address")
        txtCity.Text = reader("city")
        txtState.Text = reader("state")
        txtZip.Text = reader("zip")
        chkContract.Checked = CType(reader("contract"), Boolean)
        reader.Close()
        lblStatus.Text = ""
    Catch err As Exception
        lblStatus.Text = "Error getting author. "
        lblStatus.Text &= err.Message
    Finally
        If (Not con Is Nothing) Then
            con.Close()
        End If
    End Try

End Sub
```

To see the full code, refer to the included example files. If you play with the example at length, you'll notice that it lacks a few niceties that would be needed in a professional web site. For example, when creating a new record, the name of the last selected user is still visible, and the Update and Delete buttons are still active, which can lead to confusion or errors. A more sophisticated user interface could prevent these problems by disabling inapplicable controls (perhaps by grouping them in a Panel control) or using separate pages. In this case, however, the page is useful as a quick way to test some basic data access code.

Updating a Record

When the user clicks the Update button, the information in the text boxes is applied to the database:

```
Private Sub cmdUpdate_Click(sender As Object, _
  e As EventArgs) Handles cmdUpdate.Click

    ' Define ADO.NET objects.
    Dim UpdateSQL As String
    UpdateSQL = "UPDATE Authors SET "
    UpdateSQL &= "au_id='" & txtID.Text & "', "
    UpdateSQL &= "au_fname='" & txtFirstName.Text & "', "
    UpdateSQL &= "au_lname='" & txtLastName.Text & "', "
    UpdateSQL &= "phone='" & txtPhone.Text & "', "
    UpdateSQL &= "address='" & txtAddress.Text & "', "
    UpdateSQL &= "city='" & txtCity.Text & "', "
    UpdateSQL &= "state='" & txtState.Text & "', "
    UpdateSQL &= "zip='" & txtZip.Text & "', "
    UpdateSQL &= "contract='" & Int(chkContract.Checked) & "' "
    UpdateSQL &= "WHERE au_id='" & lstAuthor.SelectedItem.Value & "'"

    Dim con As New OleDbConnection(ConnectionString)
    Dim cmd As New OleDbCommand(UpdateSQL, con)

    ' Try to open database and execute the update.
    Try
        con.Open()
        Dim Updated As Integer
        Updated = cmd.ExecuteNonQuery
        lblStatus.Text = Updated.ToString() & " records updated."
    Catch err As Exception
        lblStatus.Text = "Error updating author. "
        lblStatus.Text &= err.Message
    Finally
        If (Not con Is Nothing) Then
            con.Close()
        End If
    End Try

End Sub
```

The update code is similar to the record selection code. The main differences are:

- No DataReader is used, because no results are returned.
- A dynamically generated Update command is used for the Command object. It selects the corresponding author records, and changes all the fields to correspond to the values entered in the text boxes.
- The ExecuteNonQuery method returns the number of affected records. This information is displayed in a label to confirm to the user that the operation was successful.

Deleting a Record

When the user clicks the Delete button, the author information is removed from the database. The number of affected records is examined, and if the Delete operation was successful, the FillAuthorList function is called to refresh the page.

```
Private Sub cmdDelete_Click(sender As Object, _
  e As EventArgs) Handles cmdDelete.Click

    ' Define ADO.NET objects.
    Dim DeleteSQL As String
    DeleteSQL = "DELETE FROM Authors "
    DeleteSQL &= "WHERE au_id='" & lstAuthor.SelectedItem.Value & "'"

    Dim con As New OleDbConnection(ConnectionString)
    Dim cmd As New OleDbCommand(DeleteSQL, con)

    ' Try to open database and delete the record.
    Dim Deleted As Integer
    Try
        con.Open()
        Deleted = cmd.ExecuteNonQuery()
    Catch err As Exception
        lblStatus.Text = "Error deleting author. "
        lblStatus.Text &= err.Message
    Finally
        If (Not con Is Nothing) Then
            con.Close()
        End If
    End Try

    ' If the delete succeeded, refresh the author list.
    If Deleted > 0 Then
        FillAuthorList()
    End If

End Sub
```

Interestingly, delete operations rarely succeed with the records in the pubs database, because they have corresponding child records linked in another table of the pubs database. Specifically, each author can have one or more related book titles. Unless the author's records are removed from the TitleAuthor table first, the author cannot be deleted. Because of the careful error-handling used in the previous example, this problem is faithfully reported in your application (see Figure 13-8) and does not cause any real problems.

Figure 13-8: A failed delete attempt

To get around this limitation, you can use the Create New and Insert New buttons to add a new record, and then delete this record. Because it is not linked to any other records, its deletion will be allowed.

Adding a Record

To start adding a new record, click Create New to clear the fields. Technically speaking, this step isn't required, but it simplifies the user's life.

```
Private Sub cmdNew_Click(sender As Object, _
  e As EventArgs) Handles cmdNew.Click

    txtID.Text = ""
    txtFirstName.Text = ""
    txtLastName.Text = ""
    txtPhone.Text = ""
    txtAddress.Text = ""
    txtCity.Text = ""
    txtState.Text = ""
    txtZip.Text = ""
    chkContract.Checked = False

    lblStatus.Text = "Click Insert New to add the completed record."

End Sub
```

The Insert New button performs the actual ADO.NET code to insert the finished record using a dynamically generated Insert statement:

```
Private Sub cmdInsert_Click(sender As Object, _
  e As EventArgs) Handles cmdInsert.Click

    ' Perform user-defined checks.
    ' Alternatively, you could use RequiredFieldValidator controls.
    If txtID.Text = "" Or txtFirstName.Text = "" Or _
      txtLastName.Text = "" Then

        lblStatus.Text = "Records require an ID, first name,"
        lblStatus.Text &= "and last name."
```

```
        Exit Sub
    End If

    ' Define ADO.NET objects.
    Dim strInsert As String
    strInsert = "INSERT INTO Authors ("
    strInsert &= "au_id, au_fname, au_lname, "
    strInsert &= "phone, address, city, state, zip, contract) "
    strInsert &= "VALUES ('"
    strInsert &= txtID.Text & "', '"
    strInsert &= txtFirstName.Text & "', '"
    strInsert &= txtLastName.Text & "', '"
    strInsert &= txtPhone.Text & "', '"
    strInsert &= txtAddress.Text & "', '"
    strInsert &= txtCity.Text & "', '"
    strInsert &= txtState.Text & "', '"
    strInsert &= txtZip.Text & "', '"
    strInsert &= Int(chkContract.Checked) & "')"

    Dim con As New OleDbConnection(ConnectionString)
    Dim cmd As New OleDbCommand(strInsert, con)

    ' Try to open database and execute the update.
    Dim Added As Integer
    Try
        con.Open()
        Added = cmd.ExecuteNonQuery
        lblStatus.Text = Added.ToString() & " records inserted."
    Catch err As Exception
        lblStatus.Text = "Error inserting record. "
        lblStatus.Text &= err.Message
    Finally
        If (Not con Is Nothing) Then
            con.Close()
        End If
    End Try

    ' If the insert succeeded, refresh the author list.
    If Added > 0 Then
        FillAuthorList()
    End If

End Sub
```

If the insert fails, the problem will be reported to the user in a rather unfriendly way (see Figure 13-9). This is typically a result of not specifying valid values. In a more polished application, you would use validators (as discussed in Chapter 9) and provide more useful error messages.

Figure 13-9: A failed insertion

If the insert operation is successful, the page is updated with the new author list.

Accessing Disconnected Data

With disconnected data manipulation, your coding is a little different. In many cases, you have less SQL code to write, and you modify records through the DataSet object. However, you need to carefully watch for problems that can occur when you send changes to the data source. In our simple one-page scenario, disconnected data access won't present much of a problem. If, however, you use disconnected data access to make a number of changes and commit them all at once, you are more likely to run into trouble.

With disconnected data access, a copy of the data is retained, briefly, in memory while your code is running. Changes are tracked automatically using the built-in features of the DataSet object. You fill the DataSet in much the same way that you connect a DataReader. However, while the DataReader held a live connection, information in the DataSet is always disconnected.

Selecting Disconnected Data

The following example shows how you could rewrite the FillAuthorList() subroutine to use a DataSet instead of a DataReader. The changes are highlighted in bold.

```
Private Sub FillAuthorList()

    lstAuthor.Items.Clear()

    ' Define ADO.NET objects.
    Dim SelectSQL As String
    SelectSQL = "SELECT au_lname, au_fname, au_id FROM Authors"
    Dim con As New OleDbConnection(ConnectionString)
    Dim cmd As New OleDbCommand(SelectSQL, con)
    Dim adapter As New OleDbDataAdapter(cmd)
    Dim Pubs As New DataSet()

    ' Try to open database and read information.
    Try
        con.Open()
        ' All the information in transferred with one command.
```

```
        adapter.Fill(Pubs, "Authors")
    Catch err As Exception
        lblStatus.Text = "Error reading list of names. "
        lblStatus.Text &= err.Message
    Finally
        If (Not con Is Nothing) Then
            con.Close()
        End If
    End Try

    Dim row As DataRow
    For Each row In Pubs.Tables("Authors").Rows
        Dim NewItem As New ListItem()
        NewItem.Text = row("au_lname") & ", " & _
          row("au_fname")
        NewItem.Value = row("au_id")
        lstAuthor.Items.Add(NewItem)
    Next

End Sub
```

Every DataAdapter can hold four different commands: SelectCommand, InsertCommand, UpdateCommand, and DeleteCommand. This allows you to use a single DataAdapter object for multiple tasks. The Command object supplied in the constructor is automatically assigned to the DataAdapter.SelectCommand property.

The DataAdapter.Fill method takes a DataSet and inserts one table of information. In this case, the table is named Authors, but any name could be used. That name is used later to access the appropriate table in the DataSet.

To access the individual DataRows, you can loop through the Rows collection of the appropriate table. Each piece of information is accessed using the field name, as it was with the DataReader.

Selecting Multiple Tables

Some of the most impressive features of the DataSet appear when you retrieve multiple different tables. A DataSet can contain as many tables as you need, and you can even link these tables together to better emulate the underlying relational data source. Unfortunately, there's no way to connect tables together automatically based on relationships in the underlying data source. However, you can perform this connection manually, as shown in the next example.

In the pubs database, authors are linked to titles using three tables. This arrangement (called a many-to-many relationship, shown in Figure 13-10) allows several authors to be related to one title, and several titles to be related to one author. Without the intermediate TitleAuthor table, the database would be restricted to a one-to-many relationship, which would only allow a single author for each title.

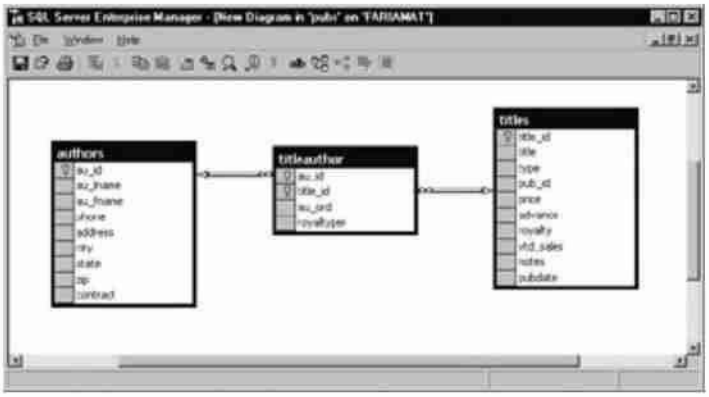

Figure 13-10: A many-to-many relationship

In an application, you rarely need to access these tables individually. Instead, you usually need to combine information from them in some way (for example, to find out what author wrote a given book). On its own, the Titles table indicates only an author ID, but it doesn't provide additional information such as the author's name and address. To link this information together, you can use a special SQL Select statement called a Join query. Alternatively, you can use the features built into ADO.NET.

The next example provides a simple page that lists authors and the titles they have written. The interesting thing about this page is that it is generated using ADO.NET table linking.

To start, the standard ADO.NET data access objects are created, including a DataSet. In the online example, all these steps are performed in a custom CreateList subroutine (for clarity), which is called from the Page.Load event handler so that the output is created when the page is first generated.

```
' Define ADO.NET objects.
Dim SelectSQL As String
SelectSQL = "SELECT au_lname, au_fname, au_id FROM Authors"
Dim con As New OleDbConnection(ConnectionString)
Dim cmd As New OleDbCommand(SelectSQL, con)
Dim adapter As New OleDbDataAdapter(cmd)
Dim dsPubs As New DataSet()
```

Next, the information for all three tables is pulled from the database and placed in the DataSet. This task could be accomplished with three separate Command objects, but to make the code a little lighter, this example uses only one, and modifies the CommandText property as needed.

```
Try
    con.Open()
    adapter.Fill(dsPubs, "Authors")

    ' This command is still linked to the data adapter.
    cmd.CommandText = "SELECT au_id, title_id FROM TitleAuthor"
    adapter.Fill(dsPubs, "TitleAuthor")

    ' This command is still linked to the data adapter.
    cmd.CommandText = "SELECT title_id, title FROM Titles"
    adapter.Fill(dsPubs, "Titles")

Catch err As Exception
    lblList.Text = "Error reading list of names. "
    lblList.Text &= err.Message
Finally
    If (Not con Is Nothing) Then
        con.Close()
    End If
End Try
```

Now that all the information is in the database, two DataRelation objects need to be created to make it easier to navigate through the linked information. In this case, these DataRelation objects match the foreign key restrictions placed at the data source level.

To create a DataRelation, you need to specify the linked fields from two different tables, and you need to give your DataRelation a unique name. (In the following example, a blank string is specified for both names, which instructs ADO.NET to assign a default value automatically.) The order of the linked fields is important. The first field is the parent, and the second field is the child. (The idea here is that one parent can have many children, but each child can only have one parent. In other words, the parent-to-child relationship is another way of saying one-to-many.) In our example, each book title can have more than one entry in the TitleAuthor table. Each author can also have more than one entry in the TitleAuthor table.

```
Dim Titles_TitleAuthor As New DataRelation("", _
  dsPubs.Tables("Titles").Columns("title_id"), _
  dsPubs.Tables("TitleAuthor").Columns("title_id"))
Dim Authors_TitleAuthor As New DataRelation("", _
  dsPubs.Tables("Authors").Columns("au_id"), _
  dsPubs.Tables("TitleAuthor").Columns("au_id"))
```

Once these DataRelation objects have been created, they must be added to the DataSet.

```
dsPubs.Relations.Add(Titles_TitleAuthor)
dsPubs.Relations.Add(Authors_TitleAuthor)
```

The remaining code loops through the DataSet. However, unlike the previous example, which moved through one table, this example uses the DataRelation objects to branch to the other linked tables. It works like this:

1. Select the first record from the Author table.

2. Using the Authors_TitleAuthor relationship, find the child records that correspond to this author. This step uses the GetChildRows method of the DataRow.

3. For each matching record in TitleAuthor, look up the corresponding Title record to get the full text title. This step uses the GetParentRows method of the DataRow.

4. Move to the next Author record, and repeat the process.

The code is dense and economical:

```
Dim rowAuthor, rowTitleAuthor, rowTitle As DataRow

For Each rowAuthor In dsPubs.Tables("Authors").Rows

   lblList.Text &= "<br><b>" & rowAuthor("au_fname")
   lblList.Text &= " " & rowAuthor("au_lname") & "</b><br>"
   For Each rowTitleAuthor In _
     rowAuthor.GetChildRows(Authors_TitleAuthor)

      For Each rowTitle In _
        rowTitleAuthor.GetParentRows(Titles_TitleAuthor)

         lblList.Text &= "  " ' Non-breaking spaces.
         lblList.Text &= rowTitle("title") & "<br>"

      Next

   Next

Next
```

The final result is shown in Figure 13-11.

Figure 13-11: Hierarchical information from two tables

If there were a simple one-to-many relationship between authors and titles, you could leave out the inner For Each statement, and use simpler code:

```
For Each rowAuthor In dsPubs.Tables("Authors").Rows
    ' Display Author
    For Each rowTitleAuthor In rowAuthor.GetChildRows(Authors_Titles)
        ' Display Title
    Next
Next
```

However, having seen the more complicated example, you are ready to create and manage multiple DataRelation objects on your own.

DataRelations Restrict and Verify Data

Using a DataRelation implies certain restrictions. For example, if you try to create a child row that refers to a nonexistent parent, ADO.NET will generate an error. Similarly, you can't delete a parent that has linked children records. These restrictions are already enforced by the data source, but by adding them to the DataSet, you ensure that they will be enforced by ADO.NET as well. This technique allows you to catch errors as soon as they occur, rather than waiting until you attempt to commit changes to the data source.

Modifying Disconnected Data

The information in the DataSet can be easily modified. The only complication is that these changes are not committed until you update the data source with a DataAdapter object.

Modifying and Deleting Rows

Modifying and deleting rows are two of the most common changes you'll make to a DataSet. They are also the easiest. The following example modifies one author's last name. You can place this

logic into a function and call it multiple times to swap the last name back and forth.

```
For Each rowAuthor In dsPubs.Tables("Authors").Rows

   If rowAuthor("au_lname") = "Bennet" Then
      rowAuthor("au_lname") = "Samson"
   Else If rowAuthor("au_lname") = "Samson" Then
      rowAuthor("au_lname") = "Bennet"
   End If

Next
```

Deleting a row is just as easy:

```
For Each rowAuthor In dsPubs.Tables("Authors").Rows

   If rowAuthor("au_fname") = "Cheryl" Then
      rowAuthor.Delete
   End If

Next
```

Alternatively, if you know the exact position of a record, you can modify or delete it using the index number rather than enumerating through the collection:

```
dsPubs.Tables("Authors").Rows(3).Delete
```

The DataSet is always disconnected. Any changes you make will appear in your program, but won't affect the original data source unless you take additional steps. In fact, when you use the Delete method, the row is not actually removed, only marked for deletion. (If the row were removed entirely, ADO.NET would be unable to find it and delete it from the original data source when you reconnect later.)

If you use the Delete method, you need to be aware of this fact and take steps to avoid trying to use deleted rows:

```
For Each row In dsPubs.Tables("Authors").Rows
   If row.RowState <> DataRowState.Deleted Then
      ' It's OK to display the row value, modify it, or use it.
   Else
      ' This record is scheduled for deletion.
      ' You should just ignore it.
   End If
Next
```

If you try to read a field of information from a deleted item, an error will occur. As I warned earlier—life with disconnected data isn't always easy.

DataSets Also Provide a Remove Method

You can use the Remove method to delete an item completely. However, if you use this method the record won't be deleted from the data source when you reconnect and update it with your changes. Instead, it will just be eliminated from your DataSet.

Adding Information to a DataSet

You can also add a new row using the Add method of the Rows collection. Before you can add a

row, however, you need to use the NewRow method first to get a blank copy. The following example uses this technique with the original web page for viewing and adding authors:

```
Dim rowNew As DataRow
rowNew = dsPubs.Tables("Authors").NewRow
rowNew("au_id") = txtID.Text
rowNew("au_fname") = txtFirstName.Text
rowNew("au_lname") = txtLastName.Text
rowNew("phone") = txtPhone.Text
rowNew("address") = txtAddress.Text
rowNew("city") = txtCity.Text
rowNew("state") = txtState.Text
rowNew("zip") = txtZip.Text
rowNew("contract") = Int(chkContract.Checked)
dsPubs.Tables("Authors").Rows.Add(rowNew)
```

The full code needed to update the data source with these changes is included a little later in this chapter.

Updating Disconnected Data

Earlier, you saw how the DataAdapter object allows you to retrieve information. Unlike the DataReader, DataAdapter objects can transfer data in both directions.

Updating the data source is a more complicated operation than reading from it. Depending on the changes that have been made, the DataAdapter may need to perform Insert, Update, or Delete operations. Luckily, you don't need to create these Command objects by hand. Instead, you can use ADO.NET's special utility class: the CommandBuilder object (which is provided as SqlCommandBuilder and OleDbCommandBuilder).

The CommandBuilder

The CommandBuilder examines the DataAdapter object you used to create the DataSet, and it adds the additional Command objects for the InsertCommand, DeleteCommand, and UpdateCommand properties. The process works like this:

```
' Create the CommandBuilder.
Dim cb As New OleDbCommandBuilder(adapter)

' Retrieve an updated DataAdapter.
adapter = cb.DataAdapter
```

Updating a DataTable

With the correctly configured DataAdapter, you can update the data source using the Update method. You need to use this method once for each table (just like when retrieving the data).

```
Dim RowsAffected As Integer
RowsAffected = adapter.Update(dsPubs, "Authors")
RowsAffected = adapter.Update(dsPubs, "Titles")
```

The DataSet stores information about the current state of all the rows and their original state. This allows ADO.NET to find the changed rows. It adds every new row (rows with the state DataRowState.Added) with the DataAdapter's Insert command. It removes every deleted row (DataRowState.Deleted) using the DataAdapter's Delete command. It also updates every changed row (DataRowState.Modified) with the Update command. There is no guaranteed order in which these operations will take place. Once the update is successfully completed, all rows will be reset to DataRowState.Unchanged.

Controlling Updates

If you used linked tables, the standard way of updating the data source can cause some problems, particularly if you have deleted or added records. These problems result because changes are usually not committed in the same order that they were made. (To provide this type of tracking, the DataSet object would need to store much more information and waste valuable memory on the server.)

You can control the order in which tables are updated, but that's not always enough to prevent a conflict. For example, if you update the Authors table first, ADO.NET might try to delete an Author record before it deletes the TitleAuthor record that is still using it. It you update the TitleAuthor table first, ADO.NET might try to create a TitleAuthor that references a nonexistent Title. At some point, ADO.NET may be enhanced with the intelligence needed to avoid these problems, provided you use DataRelations. Currently, you need to resort to other techniques for more fine-tuned control.

Using features built into the DataSet, you can pull out the rows that need to be added, modified, or deleted into separate DataSets, and choose an update order that will not place the database into an inconsistent state at any point. Generally, you can safely update a database by performing all the record inserts, followed by all the modifications, and then all the deletions.

By using the DataSet's GetChanges method, you can implement this exact pattern:

```
' Create three DataSets, and fill them from dsPubs.
Dim dsNew As DataSet = dsPubs.GetChanges(DataRowState.Added)
Dim dsModify As DataSet = dsPubs.GetChanges(DataRowState.Deleted)
Dim dsDelete As DataSet = dsPubs.GetChanges(DataRowState.Modified)

' Update these DataSets separately.
' Remember, each DataSet has three tables!
' Also note that the "add" operation for the Authors and Titles tables
' must be carried out before the add operation for the TitleAuthor
' table.
adapter.Update(dsNew, "Authors")
adapter.Update(dsNew, "Titles")
adapter.Update(dsNew, "TitleAuthor")
adapter.Update(dsModify, "Authors")
adapter.Update(dsModify, "Titles")
adapter.Update(dsModify, "TitleAuthor")
adapter.Update(dsDelete, "Authors")
adapter.Update(dsDelete, "Titles")
adapter.Update(dsDelete, "TitleAuthor")
```

This adds a layer of complexity, and a significant amount of extra code. However, in cases where you make many different types of modifications to several different tables at once, this is the only solution. To avoid these problems, you could commit changes earlier (rather than committing an entire batch of changes at once), or use Command objects directly instead of relying on the disconnected data features of the DataSet.

An Update Example

The next example rewrites the code for adding a new author in the update page with equivalent DataSet code. You can find this example in the online samples as AuthorManager2.aspx. (AuthorManager.aspx is the original, command-based version.)

```
Private Sub cmdInsert_Click(sender As Object, e As EventArgs) _
  Handles cmdInsert.Click

    ' Define ADO.NET objects.
    Dim SelectSQL As String
```

```
    SelectSQL = "SELECT * FROM Authors"
    Dim con As New OleDbConnection(ConnectionString)
    Dim cmd As New OleDbCommand(SelectSQL, con)
    Dim adapter As New OleDbDataAdapter(cmd)
    Dim dsPubs As New DataSet()

    ' Get the schema information.
    Try
        con.Open()
        adapter.FillSchema(dsPubs, SchemaType.Mapped, "Authors")
    Catch err As Exception
        lblStatus.Text = "Error reading schema. "
        lblStatus.Text &= err.Message
    Finally
        If (Not con Is Nothing) Then
            con.Close()
        End If
    End Try

    Dim rowNew As DataRow
    rowNew = dsPubs.Tables("Authors").NewRow
    rowNew("au_id") = txtID.Text
    rowNew("au_fname") = txtFirstName.Text
    rowNew("au_lname") = txtLastName.Text
    rowNew("phone") = txtPhone.Text
    rowNew("address") = txtAddress.Text
    rowNew("city") = txtCity.Text
    rowNew("state") = txtState.Text
    rowNew("zip") = txtZip.Text
    rowNew("contract") = Int(chkContract.Checked)
    dsPubs.Tables("Authors").Rows.Add(rowNew)

    ' Insert the new record.
    Dim Added As Integer
    Try
        ' Create the CommandBuilder.
        Dim cb As New OleDbCommandBuilder(adapter)
        ' Retrieve an updated DataAdapter.
        adapter = cb.DataAdapter

        ' Update the database using the DataSet.
        con.Open()
        Added = adapter.Update(dsPubs, "Authors")
    Catch err As Exception
        lblStatus.Text = "Error inserting record. "
        lblStatus.Text &= err.Message
    Finally
        con.Close()
    End Try

    ' If the insert succeeded, refresh the author list.
    If Added > 0 Then
        FillAuthorList()
    End If

End Sub
```

In this case, the example is quite inefficient. In order to add a new record, a DataSet needs to be created, with the required tables and a valid row. To retrieve information about what this row should look like, the FillSchema method of the data adapter is used. The FillSchema method creates a table with no rows, but with other information about the table, such as the name of each field and the requirements for each column (the data type, the maximum length, any restriction against null values, and so on). If you wanted, you could use the FillSchema method followed by the Fill method.

After this step, the information is entered into a new row in the DataSet, the DataAdapter is

updated with the CommandBuilder, and the changes are committed to the database. The whole operation took two database connections and required the use of a DataSet that was then abruptly abandoned. In this scenario, disconnected data is probably an extravagant solution to a problem that would be better solved with ordinary Command objects.

FillSchema Is Useful with Auto-Increment Columns

Many database tables use identity columns that increment automatically. For example, an alternate design of the Authors table might use au_id as an auto-incrementing column. In this case, the database would automatically assign a unique ID to each inserted author record. To allow this to work with ADO.NET, you need to use the FillSchema method for the appropriate table, so that the column is set to be an auto-increment field. When adding new authors, you wouldn't specify any value for the au_id field; instead, this number would be generated when the changes are committed to the database.

Concurrency Problems

As you discovered earlier, ADO.NET maintains information in the DataSet about the current and the original value of every piece of information in the DataSet. When updating a row, ADO.NET searches for a row that matches every "original" field exactly, and then updates it with the new values. If another user changes even a single field in that record while your program is working with the disconnected data, an exception is thrown. The update operation is then halted, potentially preventing other valid rows from being updated.

There is an easier way to handle these potential problems: using the DataAdapter's RowUpdated event. This event occurs after every individual insert, update, or delete operation, but before an exception is thrown. It gives you the chance to examine the results, note any errors, and prevent an error from occurring.

The first step is to create an appropriate event handler for the DataAdapter.RowUpdated event:

```
Public Sub OnRowUpdated(sender As Object, e As
OleDbRowUpdatedEventArgs)

    ' Check if any records were affected.
    ' If no records were affected, the statement did not execute
    ' as expected.
    If e.RecordsAffected() < 1 Then
        ' Find out the type of failed error.
        Select Case e.StatementType
            Case StatementType.Delete
                lstErrors.Items.Add("Not deleted: " & e.Row("au_id"))
            Case StatementType.Insert
                lstErrors.Items.Add("Not inserted: " & e.Row("au_id"))
            Case StatementType.Update
                lstErrors.Items.Add("Not updated: " & e.Row("au_id"))
        End Select

        ' Using the OledDbRowUpdatedEventArgs class,
        ' you can tell ADO.NET
        ' to ignore the problem and keep updating the other rows.
        e.Status = UpdateStatus.SkipCurrentRow
    End If

End Sub
```

The OleDbRowUpdatedEventArgs object provides this event handler with information about the

Chapter 17: Using XML

Overview

XML features are no longer an optional add-on for ASP applications. XML is woven right into the fabric of .NET, and it powers key parts of ASP.NET technology. In this chapter, you'll learn why XML plays in every ASP.NET web application—whether you realize it or not.

You'll also learn how you can create and read XML documents on your own, by using the classes of the .NET library. Along the way, we'll sort through some of the near-hysteric XML hype and consider what a web application can use XML for in *practical* terms. You may find that ASP.NET's low-level XML support is all you need, and decide that you don't want to manually create and manipulate XML data. On the other hand, you might want to use the XML classes to communicate with other applications or components, manage relational data, or just as a convenient replacement for simple text files. We'll assess all these issues realistically—which is a major departure from most of today's ASP.NET articles, seminars, and books. We'll also start with a whirlwind introduction to XML and why it exists, in case you aren't already familiar with its syntax.

XML's Hidden Role in .NET

The most useful place for XML isn't in your web applications, but in the infrastructure that supports them. Microsoft has taken this philosophy to heart with ASP.NET. Instead of providing separate components that allow you to add in a basic XML parser or similar functionality, ASP.NET uses XML quietly behind the scenes to accomplish a wide range of different tasks. If you don't know much about XML yet, the most important thing you should realize is that you are already using it.

ADO.NET Data Access

Although you interact with data using a combination of .NET objects and properties, the internal format these components use to store the data is actually XML. This has a number of interesting consequences. One change (which you won't see in this book) occurs with distributed programming, which uses multiple components on different computers that communicate over a network. With ADO, these components would exchange data in a proprietary binary format, which would have difficulty crossing firewall boundaries. With ADO.NET, these components exchange pure XML, which can flow over normal channels because it is text-based. This XML data exchange could even be extended to components on another operating system or in a non-Microsoft development language. All they need to do is be able to read XML.

As an ASP.NET developer, you are more likely to see some interesting but less useful features. For example, you can store the results of a database query in an XML file, and retrieve it later in the same page or in another application. You can also manipulate the information in a DataSet by changing an XML string. This technique, which is introduced at the end of the chapter, is not that useful in most scenarios because it effectively bypasses the basic DataSet features for error checking. However, the reverse—manipulating XML documents through ADO.NET objects—can be quite handy.

Configuration Files

ASP.NET stores settings in a human-readable XML format using configuration files such as machine.config and web.config, which were first introduced in Chapter 5. Arguably, a plain text file could be just as efficient, but then the designers of the ASP.NET platform would have to create their own proprietary format, which developers would then need to learn. XML provides an all-purpose language that is designed to let programmers store data in a customized, yet very consistent and standardized way using tags. Anyone who understands XML will immediately

understand how the ASP.NET configuration files work.

Web Services

We haven't explored Web Services yet, but it's worth pointing out that they are one of ASP.NET's most important examples of integrated XML use. In order to create or use a Web Service in a .NET program, you don't actually have to understand anything about XML, because the framework handles all the details for you. However, because Web Services are built on these familiar standards, other programmers can develop clients for your Web Services in completely different programming languages, operating systems, and computer hardware platforms, with just a little more work. In fact, they can even use a competing toolkit to create a Web Service that you can consume in .NET! Cross-platform programming is clearly one of XML's key selling points.

Anywhere Miscellaneous Data Is Stored

Just when you think you've identified everywhere that XML markup is used, you'll find it appearing in another new place. You'll find XML when you write an advertisement file defining the content for the AdRotator control, or if you use .NET serialization to write an object to a file. The developers of the .NET platform have embraced XML in unprecedented ways, partially abandoning Microsoft's traditional philosophy of closed standards and proprietary technologies.

XML Explained

The basic premise of XML is fairly simple, although the possible implementations of it (and the numerous extensions to it) can get quite complex. XML is designed as an all-purpose format for encoding data. In almost every case, when you decide to use XML, you are deciding to store data in a standardized way, rather than creating your own new (and to other developers, unfamiliar) format. The actual location of this data—in memory, in a file, in a network stream—is irrelevant.

For example, consider a simple program that stores product items as a list in a file. When you first create this program, you decide that it will store three pieces of product information (ID, name, and price), and that you will use a simple text file format for easy debugging and testing. The file format you use looks like this:

```
1
Chair
49.33
2
Car
43399.55
3
Fresh Fruit Basket
49.99
```

This is the sort of format that you might create by using .NET classes like the StreamWriter. It's easy to work with—you just write all the information, in order, from top to bottom. Of course, it's a fairly fragile format. If you decide to store an extra piece of information in the file (such as a flag that indicates if an item is available), your old code won't work. Instead you might need to resort to adding a header that indicates the version of the file.

```
SuperProProductList
Version 2.0
1
Chair
49.33
True
2
Car
```

```
43399.55
True
3
Fresh Fruit Basket
49.99
False
```

Now, you could check the file version when you open it and use different file reading code appropriately. Unfortunately, as you add more and more possible versions, this code will become incredibly tangled, and you may accidentally break one of the earlier file formats without realizing it. A better approach would be to create a file format that indicates where every product record starts and stops. Your code would then just set some appropriate defaults if it encounters missing information in an older file format.

```
SuperProProductList
Version 3.0
##Start##
1
Chair
49.33
True
3
##Start##
2
Car
43399.55
True
3
##Start##
3
Fresh Fruit Basket
49.99
False
4
```

All in all, this isn't a bad effort. Unfortunately, you may as well use the binary file format at this point—the text file is becoming hard to read, and it's even harder to guess what piece of information each value represents. On the code side, you'll also need some basic error-checking abilities of your own. For example, you should make your code able to skip over accidentally entered blank lines, detect a missing ##Start## tag, and so on, just to provide a basic level of protection.

The central problem with this homegrown solution is that you're reinventing the wheel. While you are trying to write basic file access code and create a reasonably flexible file format for a simple task, other programmers around the world are creating their own private, ad hoc solutions. Even if your program works fine and you can understand it, other programmers will definitely not find it as easy.

Improving the List with XML

This is where XML comes into the picture. XML is an all-purpose way to identify data using tags. These tags use the same sort of format found in an HTML file, but while HTML tags indicate formatting, XML tags indicate content. (Because an XML file is just about data, there is no standardized way to display it in a browser.)

The SuperProProductList could use the following, clearer XML syntax:

```
<?xml version="1.0"?>
<SuperProProducList>
   <Product>
      <ID>1</ID>
```

```
    <Name>Chair</Name>
    <Price>49.33</Price>
    <Available>True</Available>
    <Status>3</Status>
  </Product>
  <Product>
    <ID>2</ID>
    <Name>Car</Name>
    <Price>43399.55</Price>
    <Available>True</Available>
    <Status>3</Status>
  </Product>
  <Product>
    <ID>3</ID>
    <Name>Fresh Fruit Basket</Name>
    <Price>49.99</Price>
    <Available>False</Available>
    <Status>4</Status>
  </Product>
</SuperProProductList>
```

This format is clearly readable. Every product item is enclosed in a <Product> tag, and every piece of information has its own tag with an appropriate name. Tags are nested several layers deep to show relationships. Essentially, XML provides the basic tag syntax, and you (the programmer) define the tags that you want to use.

Best of all, when you read this XML document in most programming languages (including those in the .NET framework), you can use XML parsers to make your life easier. In other words, you don't need to worry about detecting where a tag starts and stops, collapsing whitespace, and identifying attributes (although you do need to worry about case, as XML is case-sensitive). Instead, you can just read the file into some helpful XML data objects that make navigating through the entire document much easier.

XML versus Databases

XML can do an incredible number of things—perhaps including some that it was never designed for. This book is not intended to teach you XML programming, but good ASP.NET application design. For most ASP.NET programmers, XML is an ideal replacement for custom file access routines, and works best in situations where you need to store a small amount of data for relatively simple tasks.

XML files are not a good substitution for a database, because they have the same limitations as any other type of file access. Database products provide a far richer set of features for managing multi-user concurrency and providing optimized performance. XML documents also can't enforce table relations or provide the same level of error checking.

XML Basics

Part of XML's popularity is the result of its simplicity. When creating your own XML document, there are only a few rules you need to remember. The following two considerations apply to XML and HTML markup:

- Whitespace is automatically collapsed, so you can freely use tabs and hard returns to properly align your information. To add a real space, you need to use the   entity equivalent, as in HTML.

- You can only use valid characters. Special characters, such as the angle brackets (< >) and the ampersand (&), can't be entered as content. Instead, you'll have to use the entity equivalents (such as < and > for angle brackets, and & for the ampersand). These equivalents are the same as in HTML coding, and will be automatically converted back to the original characters when you read them into your program with the appropriate .NET classes.

In addition, XML imposes rules that are not found in ordinary HTML:

- XML tags are case-sensitive, so <ID> and <id> are completely different tags.

- Every start tag must have an end tag, or you must use the special "empty tag" format, which includes a forward slash at the end. For example, you can use <Name>Content</Name> or <Name />, but you cannot use <Name> on its own. This is similar to the syntax for ASP.NET controls.

- All tags must be nested in a root tag. In the SuperProProductList example, the root tag is <SuperProProductList>. As soon as the root tag is closed, the document is finished, and you cannot add any more content. In other words, if you omit the <SuperProProductList> tag and start with a <Product> tag, you will only be able to enter information for one product, because as soon as you add the closing </Product> the document is complete.

- Every tag must be fully enclosed. In other words, when you open a subtag, you need to close it before you can close the parent. <Product><ID></ID></Product> is valid, but <Product><ID></Product></ID> is not. As a general rule, indent when you open a new tag, as this will allow you to see the document's structure and notice if you accidentally close the wrong tag first.

If you meet these requirements, your XML document can be parsed and displayed as a basic tree. This means that your document is *well formed*, but it does not mean that it is valid. For example, you may still have your elements in the wrong order (for example, <ID><Product></Product></ID>), or you may have the wrong type of data in a given field (for example, <ID>Chair</ID><Name>2</Name). There are ways to impose these additional rules on your XML documents, as you'll see shortly.

XML tags are also known as elements. These elements are the primary units for organizing information, but they aren't the only option. You can also use attributes.

Attributes

Attributes are used to add extra information to an element. Instead of putting information into a subtag, you can use an attribute. In the XML community, deciding whether to use subtags or attributes—and what information should go into an attribute—is a matter of great debate, with no clear consensus.

Here's the SuperProProductList example with an ID and Name attribute instead of an ID and Name subtag:

```
<?xml version="1.0"?>
<SuperProProducList>
   <Product ID="1" Name="Chair">
      <Price>49.33</Price>
      <Available>True</Available>
      <Status>3</Status>
   </Product>
   <Product ID="2" Name="Car">
```

```
    <Price>43399.55</Price>
    <Available>True</Available>
    <Status>3</Status>
  </Product>
  <Product ID="3" Name="Fresh Fruit Basket">
    <Price>49.99</Price>
    <Available>False</Available>
    <Status>4</Status>
  </Product>
</SuperProProductList>
```

Of course, you've already seen this sort of syntax with HTML tags and ASP.NET server controls.

```
<asp:DropDownList id="lstBackColor" AutoPostBack="True"
                  runat="server"
                  Width="194px"
                  Height="22px"/>
```

Attributes are also common in the configuration file.

```
<sessionState mode="Inproc "
              cookieless="false"
              timeout="20" />
```

Note that the use of attributes in XML is more stringent than in HTML. In XML, Attributes must always have values, and these values must use quotation marks. For example, <Product Name="Chair" /> is acceptable, but <Product Name=Chair /> or <Product Name /> is not. (ASP.NET control tags do not always follow these rules.)

Comments

You can also add comments to an XML document. Comments go anywhere and are ignored for data processing purposes. Comments are bracketed by the <!— and —> codes.

```
<?xml version="1.0"?>
<SuperProProducList>
<!-- This is a test file. -->
  <Product ID="1" Name="Chair">
    <Price>49.33<!-- Why so expensive? --></Price>
    <Available>True</Available>
    <Status>3</Status>
  </Product>
  <!-- Other products omitted for clarity. -->
</SuperProProductList>
```

The XML Classes

.NET provides a rich set of classes for XML manipulation in several namespaces. One of the most confusing aspects of using XML with .NET is deciding which combination of classes you should use. Many of them provide similar functionality in a slightly different way, optimized for specific scenarios or for compatibility with specific standards.

The majority of the examples we'll explore use the types in this namespace, which allow you to read and write XML files, manipulate XML data in special objects, and even validate XML documents. We'll consider several options for dealing with XML data:

- Dealing with XML as a special type of text file, using XmlTextWriter and XmlTextReader.
- Dealing with XML as a collection of in-memory objects, such as XmlDocument and

XmlNode.

- Dealing with XML as a special interface to relational data, using the XmlDataDocument class.

The XML TextWriter

One of the simplest ways to create or read any XML document is to use the basic XmlTextWriter and XmlTextReader classes. These classes work like their StreamWriter and StreamReader relatives, except for the fact that they write and read XML documents instead of ordinary text files. That means that you follow the same process that you saw in Chapter 16 for creating a file. First, you create or open the file. Then you write to it or read from it, moving from top to bottom. Finally, you close it, and get to work using the retrieved data in whatever way you'd like.

Before beginning this example, you'll need to import the namespaces for file handling and XML.

```
Imports System.IO
Imports System.Xml
```

Here's an example that creates a simple version of the SuperProProductList document.

```
Dim fs As New FileStream("c:\SuperProProductList.xml", _
                         FileMode.Create)
Dim w As New XmlTextWriter(fs, Nothing)

w.WriteStartDocument()
w.WriteStartElement("SuperProProducList")
w.WriteComment("This file generated by the XmlTextWriter class.")

' Write the first product.
w.WriteStartElement("Product")
w.WriteAttributeString("ID", "", "1")
w.WriteAttributeString("Name", "", "Chair")

w.WriteStartElement("Price")
w.WriteString("49.33")
w.WriteEndElement()

w.WriteEndElement()

' Write the second product.
w.WriteStartElement("Product")
w.WriteAttributeString("ID", "", "2")
w.WriteAttributeString("Name", "", "Car")

w.WriteStartElement("Price")
w.WriteString("43399.55")
w.WriteEndElement()

w.WriteEndElement()

' Write the third product.
w.WriteStartElement("Product")
w.WriteAttributeString("ID", "", "3")
w.WriteAttributeString("Name", "", "Fresh Fruit Basket")

w.WriteStartElement("Price")
w.WriteString("49.99")
w.WriteEndElement()

w.WriteEndElement()

' Close the root element.
```

```
w.WriteEndElement()
w.WriteEndDocument()
w.Close()
```

Formatting Your XML

By default, the XmlTextWriter will create an XML file that has all its tags lumped together in a single line without any helpful carriage returns or indentation. Although additional formatting isn't required, it can make a significant difference if you want to read your XML files in Notepad or another text editor. Fortunately, the XmlTextWriter supports formatting; you just need to enable it, as follows:

```
' Set it to indent output.
w.Formatting = Formatting.Indented

' Set the number of indent spaces.
w.Indentation = 5
```

The code on the previous page is similar to the code used for writing a basic text file. It does have a few advantages, however. You can close elements quickly and accurately, the angle brackets (< >) are included for you automatically, and some errors (such as closing the root element too soon) are caught automatically, ensuring a well-formed XML document as the final result.

To check that your code worked, open the file in Internet Explorer, which automatically provides a collapsible view for XML documents (see Figure 17-1).

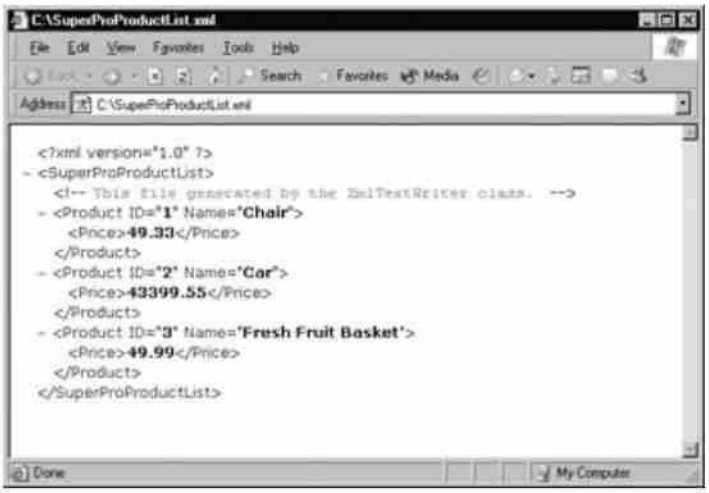

Figure 17-1: SuperProProductList.xml

The XML Text Reader

Reading the XML document in your code is just as easy with the corresponding XmlTextReader class. The XmlTextReader moves through your document from top to bottom, one node at a time. You call the XmlTextReader.Read method to move to the next node. This method returns True if there are more nodes to read, or False once it has read the final node. The current node is provided through the properties of the XmlTextReader class, like NodeType and Name.

A *node* is a designation that includes comments, whitespace, opening tags, closing tags, content, and even the XML declaration at the top of your file. To get a quick understanding of nodes, you can use the XmlTextReader to run through your entire document from start to finish, and display every node it encounters. The code for this task is shown next.

```
Dim fs As New FileStream("c:\SuperProProductList.xml", FileMode.Open)
Dim r As New XmlTextReader(fs)

' Parse the file and read each node.
Do While r.Read()
   lblStatus.Text &= "<b>Type:</b> " & r.NodeType.ToString & "<br>"

   If r.Name <> "" Then
      lblStatus.Text &= "<b>Name:</b> " & r.Name & "<br>"
   End If

   If r.Value <> "" Then
      lblStatus.Text &= "<b>Value:</b> " & r.Value & "<br>"
   End If

   If r.AttributeCount > 0 Then
      lblStatus.Text &= "<b>Attributes:</b> "
      Dim i As Integer
      For i = 0 To r.AttributeCount() - 1
         lblStatus.Text &= "  " & r.GetAttribute(i) & " "
      Next
      lblStatus.Text &= "<br>"
   End If

   lblStatus.Text &= "<br>"
Loop
```

To test this out, try the XmlText.aspx page included with the online samples (as shown in Figure 17-2). The following is a list of all the nodes that are found, shortened to include only one product:

Figure 17-2: Reading XML Structure

Type: XmlDeclaration
Name: xml
Value: version="1.0"
Attributes: 1.0

Type: Element
Name: SuperProProductList

Type: Comment
Value: This file generated by the XmlTextWriter class.

Type: Element

```
Name: Product
Attributes: 1, Chair

Type: Element
Name: Price

Type: Text
Value: 49.33

Type: EndElement
Name: Price

Type: EndElement
Name: Product

Type: EndElement
Name: SuperProProductList
```

In a typical application, you would need to go "fishing" for the elements that interest you. For example, you might read information from an XML file such as SuperProPriceList.xml and use it to create Product objects, based on the Product class shown here:

```
Public Class Product
    Private _ID As Integer
    Private _Name As String
    Private _Price As Decimal

    Public Property ID() As Integer
        Get
            Return _ID
        End Get
        Set(ByVal Value As Integer)
            _ID = Value
        End Set
    End Property

    Public Property Name() As String
        Get
            Return _Name
        End Get
        Set(ByVal Value As String)
            _Name = Value
        End Set
    End Property

    Public Property Price() As Decimal
        Get
            Return _Price
        End Get
        Set(ByVal Value As Decimal)
            _Price = Value
        End Set
    End Property

End Class
```

There's nothing particularly special about this class—all it does is allow you to store three related pieces of information (a price, name, and ID). Note that this class uses property procedures and so is eligible for data binding.

A typical application might read data out of an XML file and place it directly into the corresponding objects. The next example (also a part of the XmlText.aspx page) shows how you can easily create a group of Product objects based on the SuperProProductList.xml file.

```
' Open a stream to the file.
```

```
Dim fs As New FileStream("c:\SuperProProductList.xml", FileMode.Open)
Dim r As New XmlTextReader(fs)

Dim Products As New ArrayList()

' Loop through the products.
Do While r.Read()

    If r.NodeType = XmlNodeType.Element And r.Name = "Product" Then
        Dim NewProduct As New Product()
        NewProduct.ID = Val(r.GetAttribute(0))
        NewProduct.Name = r.GetAttribute(1)

        ' Get the rest of the subtags for this product.
        Do Until r.NodeType = XmlNodeType.EndElement
            r.Read()

            ' Look for Price subtags.
            If r.Name = "Price" Then
                Do Until (r.NodeType = XmlNodeType.EndElement)
                    r.Read()
                    If r.NodeType = XmlNodeType.Text Then
                        NewProduct.Price = Val(r.Value)
                    End If
                Loop
            End If

            ' We could check for other Product nodes
            ' (like Available, Status, etc.) here.
        Loop

        ' Add the product to the list.
        Products.Add(NewProduct)
    End If
Loop

r.Close()

' Display the retrieved document.
dgResults.DataSource = Products
dgResults.DataBind()
```

Interesting Facts About this Code

- This code uses a nested looping structure. The outside loop iterates over all the products, and the inner loop searches through all the product tags (in this case, there is only a possible Price tag). This keeps the code well organized. The EndElement node alerts us when a node is complete and the loop can end. Once all the information is read for a product, the corresponding object is added into the ArrayList collection.

- All the information is retrieved from the XML file as a string. Thus, the Val function is used to extract numbers, although there are other possibilities.

- Data binding is used to display the contents of the collection. A DataGrid set to generate columns automatically creates the table shown in Figure 17-3.

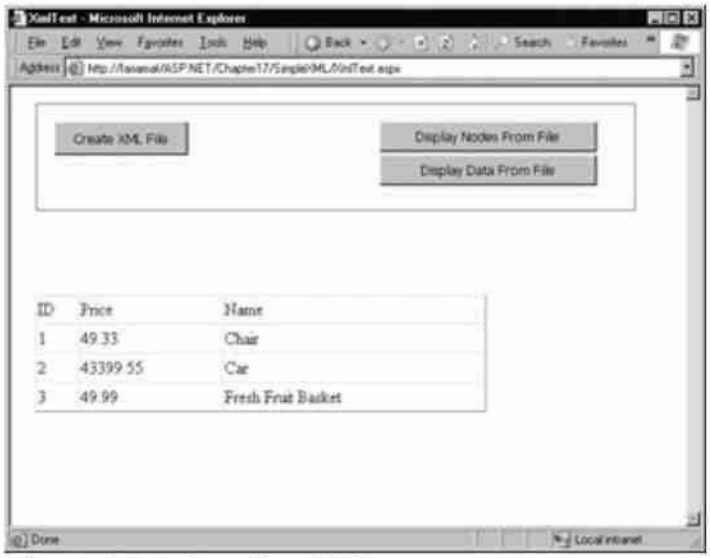

Figure 17-3: Reading XML content

Other XmlTextReader Properties

The XmlTextReader provides many more properties and methods. These additional members don't add new functionality, they allow for increased flexibility. For example, you can read a portion of an XML document into a string using methods such as ReadString, ReadInnerXml, and ReadOuterXml. These members are all documented in the MSDN class library reference. Generally, the most straightforward approach is just to work your way through the nodes as shown in the SuperProProductList example.

Working with XML Documents

The XmlTextReader and XmlTextWriter use XML as a backing store. They are streamlined for getting data into and out of a file quickly. You won't actually work with the XML data in your program. Instead, you open the file, use the data to create the appropriate objects or fill the appropriate controls, and close it soon after. This approach is ideal for storing simple blocks of data. For example, the guest book page in the previous chapter could be slightly modified to store data in an XML format, which would provide greater standardization, but wouldn't change how the application works.

The XmlDocument class provides a different approach to XML data. It provides an in-memory model of an entire XML document. You can then browse through the document, reading, inserting, or removing nodes at any location. The XML document can be saved to or retrieved from a file, but the XmlDocument class does not maintain a direct connection to the file. (The XmlTextWriter and XmlTextReader, on the other hand, are always connected to a stream, which is usually a file.) In this respect, the XmlDocument is analogous to the DataSet in ADO.NET programming: it's always disconnected.

When you use the XmlDocument class, your XML file is created as a series of linked .NET objects in memory. It's similar to the object relationships used with ADO.NET, where every DataSet contains more than one DataTable object, each of which can contain DataRow objects with the information for a single record. The object model is shown in Figure 17-4.

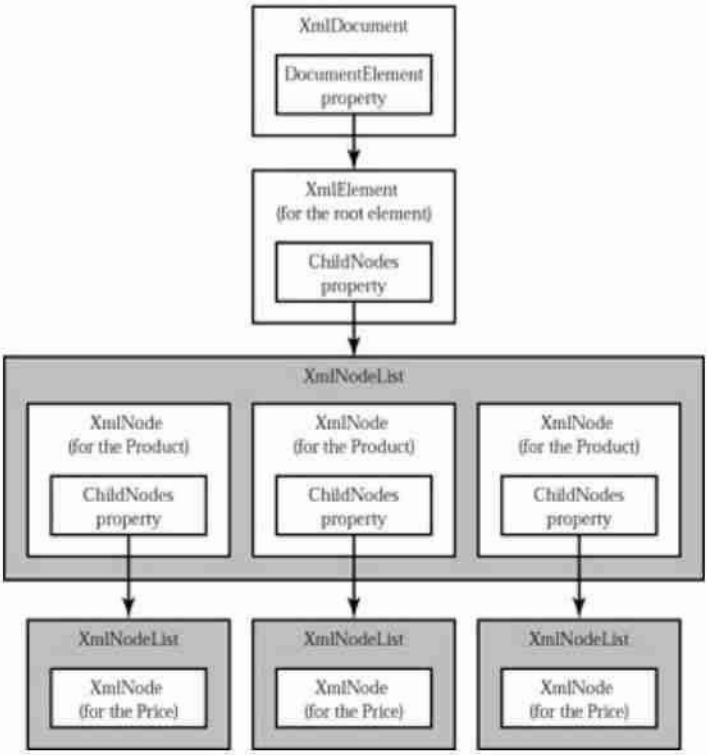

Figure 17-4: An XML document in memory

Next is an example that creates the SuperProProductList in memory, using an XmlDocument class. When it's finished, the XML document is transferred to a file using the XmlDocument.Save method.

```
' Start with a blank in memory document.
Dim doc As New XmlDocument()

' Create some variables that will be useful for
' manipulating XML data.
Dim RootElement, ProductElement, PriceElement As XmlElement
Dim ProductAttribute As XmlAttribute
Dim Comment As XmlComment

' Create the declaration.
Dim Declaration As XmlDeclaration
Declaration = doc.CreateXmlDeclaration("1.0", Nothing, "yes")

' Insert the declaration as the first node.
doc.InsertBefore(Declaration, doc.DocumentElement)

' Add a comment.
Comment = doc.CreateComment("Created with the XmlDocument class.")
doc.InsertAfter(Comment, Declaration)

' Add the root node.
RootElement = doc.CreateElement("SuperProProductList")
doc.InsertAfter(RootElement, Comment)

' Add the first product.
ProductElement = doc.CreateElement("Product")
RootElement.AppendChild(ProductElement)

' Set and add the product attributes.
ProductAttribute = doc.CreateAttribute("ID")
ProductAttribute.Value = "1"
ProductElement.SetAttributeNode(ProductAttribute)
ProductAttribute = doc.CreateAttribute("Name")
ProductAttribute.Value = "Chair"
ProductElement.SetAttributeNode(ProductAttribute)
```

```
' Add the price node.
PriceElement = doc.CreateElement("Price")
PriceElement.InnerText = "49.33"
ProductElement.AppendChild(PriceElement)

' (Code to add two more products omitted.)

' Save the document.
doc.Save("c:\SuperProProductList.xml")
```

One of the best features of the XmlDocument class is that it does not rely on any underlying file. When you use the Save method, the file is created, a stream is opened, the information is written, and the file is closed, all in one line of code. That means that this is probably the only line that you need to put inside a Try/Catch error-handling block.

While you are manipulating data with the XML objects, your text file is not being changed. Once again, this is conceptually similar to the ADO.NET DataSet.

Interesting Facts About this Code

- Every separate part of the XML document is created as an object. Elements (tags) are created as XmlElement objects, comments are created as XmlComment objects, and attributes are represented as XmlAttribute objects.

- To create a new element, comment, or attribute for your XML document, you need to use one of the XmlDocument class methods, such as CreateComment, CreateAttribute, or CreateElement. This ensures that the XML is generated correctly for your document, but it does not actually place any information into the XmlDocument.

- Once you have created the object and entered any additional inner information, you need to add it to your document. You can do so using methods such as InsertBefore or InsertAfter (from the XmlDocument object) to place your root node. To add a child element (such as the Product element inside the SuperProProductList element), you need to find the appropriate parent object and use its AppendChild method. In other words, you need to keep track of some object references; you can't write a child element directly to the document in the same way that you could with the XmlTextWriter.

- You can insert nodes anywhere. While the XmlTextWriter and XmlTextReader forced you to read every node, from start to finish, the XmlDocument is a much more flexible collection of objects.

- The file written by this code is shown in Figure 17-5 (as displayed by Internet Explorer).

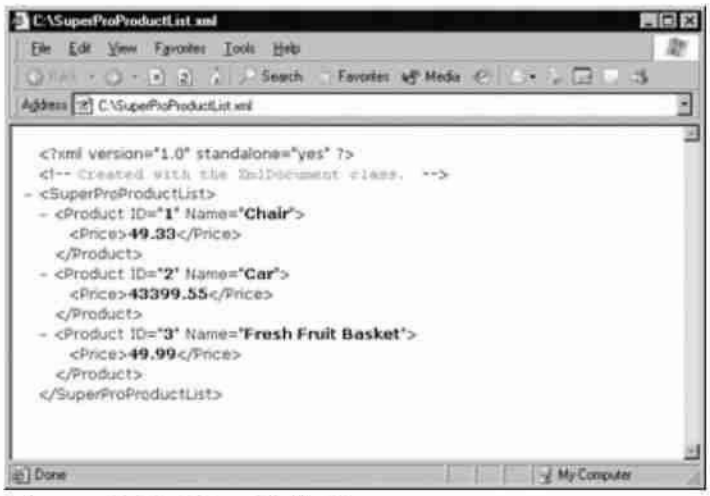

Figure 17-5: The XML file

Reading an XML Document

To read information from your XML file, all you need to do is create an XmlDocument object and use its Load method. Depending on your needs, you may want to keep the data in its XML form, or you can extract by looping through the collection of linked XmlNode objects. This process is similar to the XmlTextReader example, but the code is noticeably cleaner.

```
' Create the document.
Dim doc As New XmlDataDocument()
doc.Load("c:\SuperProProductList.xml")

' Loop through all the nodes, and create the ArrayList..
Dim Element As XmlElement
Dim Products As New ArrayList()

For Each Element In doc.DocumentElement.ChildNodes
   Dim NewProduct As New Product()
   NewProduct.ID = Element.GetAttribute("ID")
   NewProduct.Name = Element.GetAttribute("Name")

   ' If there were more than one child node, you would probably use
   ' another For Each loop here, and move through the
   ' Element.ChildNodes collection.
   NewProduct.Price() = Element.ChildNodes(0).InnerText()

   Products.Add(NewProduct)
Next

' Display the results.
dgResults.DataSource = Products
dgResults.DataBind()
```

You have a variety of other options for manipulating your XmlDocument and extracting or changing pieces of data. Table 17-1 provides an overview.

Table 17-1: XmlNode Manipulation

Technique	Description
Finding a node's relative	Every XmlNode leads to other XmlNode objects. You can use properties such as FirstChild, LastChild, PreviousSibling, NextSibling, and ParentNode to return a reference to a related node. An example is ParentNode = MyNode.ParentNode.
Cloning a portion of an XmlDocument	You can use the CloneNode method with any XmlNode to create a duplicate copy. You need to specify True or False to indicate

	whether you want to clone all children (True) or just the single node (False). An example is NewNode = MyNode.Clone(True).
Removing or adding nodes	Find the parent node, and then use one of its node adding methods. You can use AppendChild (adds the child to the end of the child list) and PrependChild (to add the node to the start of the child list). You can also remove nodes with RemoveChild, ReplaceChild, and RemoveAll (deletes all children and all attributes for the current node). An example is MyNode.RemoveChild(NodeToDelete).
Adding inner content	Find the node, and add a NodeType.Text child node. One possible shortcut is just to set the InnerText property of your node, but that will erase any existing child nodes.
Manipulating attributes	Every node provides an XmlAttributeCollection of all its attributes through the XmlNode.Attributes property. To add an attribute, you must create an XmlAttribute object, and use methods such as Append, Prepend, InsertBefore, or InsertAfter. To remove an attribute, you can use Remove and RemoveAll. An example is MyNode.Attributes.Remove(AttributeToDelete).
Working with content as string data	You can retrieve or set the content inside a node using properties such as InnerText, InnerXml, and OuterXml. Be warned that the inner content of a node includes all child nodes. Thus, setting this property carelessly could wipe out other information, like subtags.

The XmlDocument Class Also Provides Events

The XmlDocument class provides a rich set of events that fire before and after nodes are inserted, removed, and changed. The likelihood of using these events in an ordinary ASP.NET application is fairly slim, but it represents an interesting example of the features .NET puts at your fingertips.

The Difference Between XmlNode and XmlElement

You may have noticed that the XmlDocument is created with specific objects like XmlComment and XmlElement, but read back as a collection of XmlNode objects. The reason is that XmlComment and XmlElement are customized classes that inherit their basic functionality from XmlNode.

The ChildNodes collection allows you to retrieve all the content contained inside any portion of an XML document. Because this content could include comments, elements, and any other types of node, the ChildNodes collection uses the lowest common denominator. Thus, it provides child nodes as a collection of standard XmlNode objects. Each XmlNode has basic properties similar to what you saw with the XmlTextReader, including NodeType, Name, Value, and Attributes. You'll find that you can do all of your XML processing with XmlNode objects.

Searching an XML Document

One of the pleasures of the XmlDocument is its support of searching, which allows you to find nodes when you know they are there—somewhere—but you aren't sure how many matches there are or where the element is.

To search an XmlDocument, all you need to do is use the GetElementById or

GetElementsByTagName method. The following code example puts the GetElementsByTagName method to work, and creates the output shown in Figure 17-6.

Figure 17-6: Searching an XML document

```
Dim doc As New XmlDataDocument()
doc.Load("c:\SuperProProductList.xml")
Dim Results As XmlNodeList
Dim Result As XmlNode

' Find the matches.
 Results = doc.GetElementsByTagName("Price")

' Display the results.
lblStatus.Text = "<b>Found " & Results.Count.ToString() & " Matches "
lblStatus.Text &= " for the Price tag: </b><br><br>"
For Each Result In Results
   lblStatus.Text &= Result.FirstChild.Value & "<br>"
Next
```

This technique works well if you want to find an element based on its name. If you want to use more sophisticated searching, match only part of a name, or examine only part of a document, you'll have to fall back on the traditional standard: looping through all the nodes in the XmlDocument.

You Can Also Use XPath Classes

The search method provided by the XmlDocument class is relatively primitive. For a more advanced tool, you might want to learn the XPath language, which is a W3C recommendation (defined at http://www.w3.org/TR/xpath/) designed for performing queries on XML data. .NET provides XPath support through the classes in the System.Xml.XPath namespace, which include an XPath parser and evaluation engine. Of course, these aren't much use unless you learn the syntax of the XPath language itself.

XML Validation

XML has a rich set of supporting standards, many of which are far beyond the scope of this book. One of the most useful secondary standards is XSD (XML Schema Definition). XSD defines the rules that a specific XML document should conform to. When you are creating an XML file on your

own, you don't need to create a corresponding XSD file—instead, you might just rely on the ability of your code to behave properly. While this is sufficient for tightly controlled environments, if you want to open up your application to other programmers, or allow it to interoperate with other applications, you should create an XSD document.

XSD Documents

An XSD document can define what elements a document should contain and in what relationship (its structure). It can also identify the appropriate data types for all the content. XSD documents are written using an XML syntax with special tag names. Every XSD tag starts with the xs: prefix.

The full XSD specification is out of the scope of this chapter, but you can learn a lot from a simple example. The following is the slightly abbreviated SuperProProductList.xsd file. All it does is define the elements and attributes used in the SuperProProductList.xml file and their data types. It indicates that the file is a list of Product elements, which are a complex type made up of a string (Name), a decimal value (Price), and an integer (ID).

```
<?xml version="1.0"?>
<xs:schema id="SuperProProductList"
   targetNamespace="SuperProProductList" >
 <xs:element name="SuperProProductList">
   <xs:complexType>
     <xs:choice maxOccurs="unbounded">
       <xs:element name="Product">
         <xs:complexType>
           <xs:sequence>
             <xs:element name="Price" type="xs:decimal"
                minOccurs="0" />
           </xs:sequence>
           <xs:attribute name="ID" form="unqualified"
              type="xs:string" />
           <xs:attribute name="Name" form="unqualified"
              type="xs:int" />
         </xs:complexType>
       </xs:element>
     </xs:choice>
   </xs:complexType>
 </xs:element>
</xs:schema>
```

To associate this file with SuperProProductList.xml, you should make sure that they both use the same XML namespace. SuperProProductList.xsd defines a targetNamespace attribute, which is set to SuperProProductList. To specify the namespace for SuperProProductList.xml, modify the SuperProProductList element by adding the xmlns attribute.

```
<SuperProProductList xmlns="SuperProProductList" >
```

ASP.NET allows you to validate your document to make sure that it follows the rules specified in the XSD schema file, using an XmlValidatingReader class. This class enhances an existing XmlTextReader, and throws an exception (or raises an event) to indicate errors as you move through the XML file.

Validating an XML File

Before using XSD schemas, you need to import the namespace System.Xml.Schema namespace, which contains types like XmlSchema and XmlSchemaCollection.

```
Imports System.Xml.Schema
```

The code snippet that follows shows how you can create an XmlValidatingReader that uses the

SuperProProductList.xsd file, and use it to verify that the XML in SuperProProductList.xml is valid.

```
' Open the XML file.
Dim fs As New FileStream(Request.ApplicationPath & _
  "\SuperProProductList.xml", FileMode.Open)
Dim r As New XmlTextReader(fs)

' Create the validating reader.
Dim vr As New XmlValidatingReader(r)
vr.ValidationType = ValidationType.Schema

' Add the XSD file to the validator.
Dim Schemas As New XmlSchemaCollection()
Schemas.Add("SuperProProductList", _
  Request.ApplicationPath & "\SuperProProductList.xsd")
vr.Schemas.Add(Schemas)

' Read through the document.
Do While vr.Read()
    ' Process document here.
    ' If an error is found, an exception will be thrown.
Loop

vr.Close()
```

Using the current file, this code will succeed, and you will be able to access the current node through the XmlValidatingReader object in the same way that you could with the XmlTextReader. However, consider what happens if you make the minor modification shown here.

```
<Product ID="A" Name="Chair">
```

Now when you try to validate the document, an XmlSchemException (from the System.Xml.Schema namespace) will be thrown, alerting you of the invalid data type, as shown in Figure 17-7.

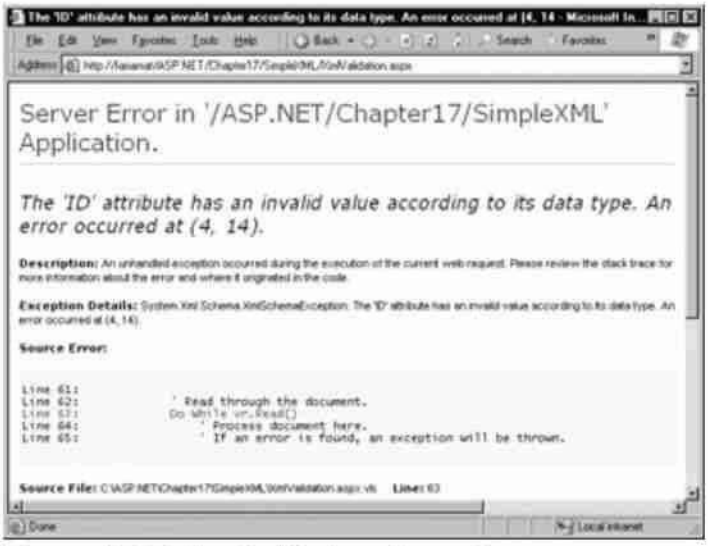

Figure 17-7: An XmlSchemException

Instead of catching errors, you can react to the ValidationEventHandler event. If you react to this event, you will be provided with information about the error, but no exception will be thrown. Typically, you would use the AddHandler statement to connect your event handling procedure to the XmlValidatingReader.ValidationEventHandler event just before you started to read the XML file.

```
AddHandler vr.ValidationEventHandler, AddressOf MyValidateHandler
```

The event handler receives a ValidationEventArgs class, which contains the exception, a message, and a number representing the severity:

```
Public Sub MyValidateHandler(sender As Object, _
  e As ValidationEventArgs)
    lblStatus.Text &= "Error: " & e.Message & "<br>"
End Sub
```

To try out validation, you can use the XmlValidation.aspx page in the online samples. It allows you to validate a valid SuperProProductList, as well as two other versions, one with incorrect data and one with an incorrect tag (see Figure 17-8).

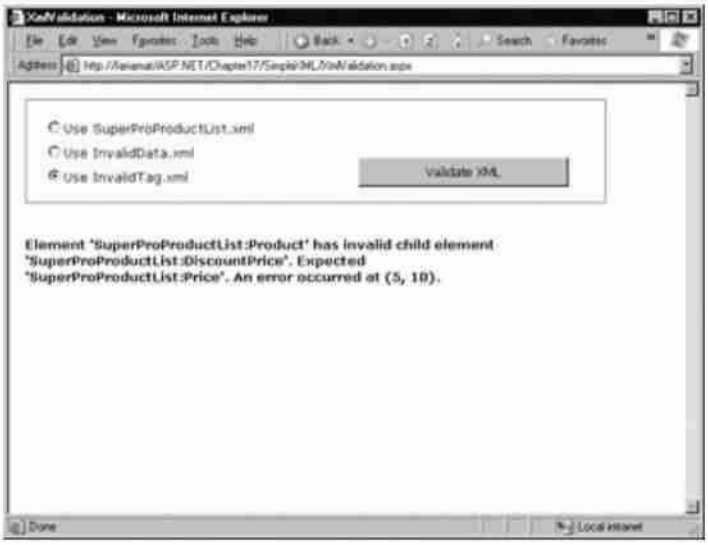

Figure 17-8: The validation test page

Creating XSD Files in Visual Studio .NET

To create a schema for your XML files, you don't have to master the subtleties of XSD syntax. Visual Studio .NET provides an impressive graphical designer that lets you point-and-click your way to success.

Start by adding the appropriate XML file to your project. You can then edit your XML data in one of two ways: in direct text view, which provides a color-coded display of the XML tags, or through a graphical editor that makes your XML content look like one or more database tables (see Figure 17-9). You can add new rows or change existing data with ease.

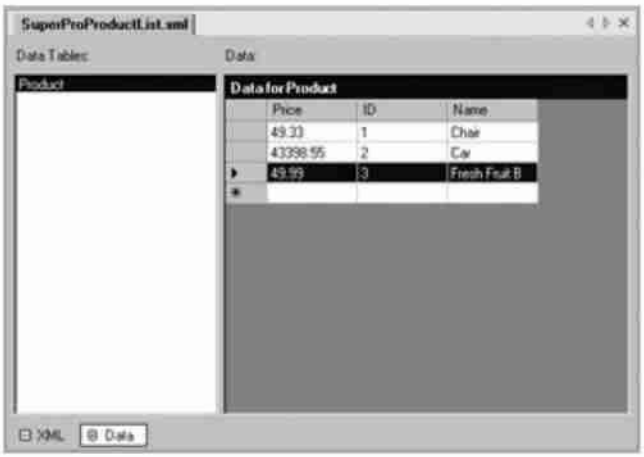

Figure 17-9: The graphical XML designer

To create the schema, just right-click anywhere on either of the XML views, and choose Create Schema from the context menu. An XSD file with the same name as your XML file will be

generated automatically and added to your project.

There are also two views for editing your XSD file: text view and another graphical designer. By default, Visual Studio .NET will assume that all the data in an XML file is made up of strings. You can use the graphical tool to choose better options for each field (see Figure 17-10), and the XSD code will be updated automatically.

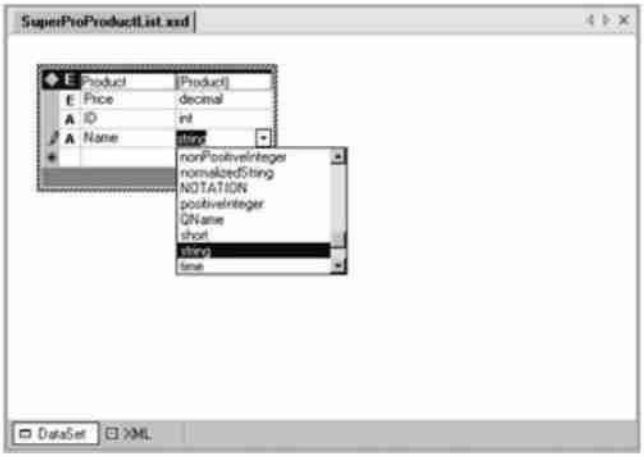

Figure 17-10: The graphical XSD designer

XML Display and Transforms

Another standard associated with XML is XSL, which uses a stylesheet to transform an XML document. XSL can be used to retrieve a portion of an XML document into another XML document—similar to how you might use SQL to retrieve some of the information in a database with a query. An even more popular use of XSL transformations is to convert an XML document into an HTML document that can be displayed in a browser.

XSL is easy to use from the point of view of the .NET class library. All you need to understand is how to create an XmlTransform object (found in the System.Xml.Xsl namespace), use its Load method to specify a stylesheet and its Transform method to output the result to a file or stream.

```
Dim Transformer As New XmlTransform

' Load the XSL stylesheet.
Transformer.Load("SuperProProductList.xslt")

' Create a transformed XML file.
' SuperProProductList.xml is the starting point.
Transformer.Transform("SuperProProductList.xml", "New.xml")
```

However, that doesn't spare you from needing to learn the XSL syntax. Once again, the intricacies of XSL are a bit of a tangent from core ASP.NET programming, and outside the scope of this book. To get started with XSL, however, it helps to review a simple example.

```
<?xml version="1.0" encoding="UTF-8" ?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   version="1.0" >

  <xsl:template match="SuperProProductList">
    <html><body><table border="1">
    <xsl:apply-templates select="Product"/>
    </table></body></html>
  </xsl:template>

  <xsl:template match="Product">
    <tr>
    <td><xsl:value-of select="@ID"/></td>
```

```
    <td><xsl:value-of select="@Name"/></td>
    <td><xsl:value-of select="Price"/></td>
    </tr>
  </xsl:template>

</xsl:stylesheet>
```

Every XSL file has a root xsl:stylesheet element. The stylesheet can contain one or more templates (the sample file SuperProProductList.xslt has two). In this example, the first template searches for the root SuperProProductList element. When it finds it, it outputs the tags necessary to start an HTML table, and then uses the xsl:apply-templates command to branch off and perform processing for any contained Product elements.

```
<xsl:template match="SuperProProductList">
  <html><body><table border="1">
  <xsl:apply-templates select="Product"/>
```

When that process is complete, the HTML tags for the end of the table will be written.

```
</table></body></html>
```

When processing each Product element, information about the ID, Name, and Price is extracted and written to the output using the xsl:value-of command. The @ sign indicates that the value is being extracted from an attribute, not a subtag. Every piece of information is written inside a table row. For more advanced formatting, you could use additional HTML tags to format some text in bold or italics, for example.

```
<xsl:template match="Product">
  <tr>
  <td><xsl:value-of select="@ID"/></td>
  <td><xsl:value-of select="@Name"/></td>
  <td><xsl:value-of select="Price"/></td>
  </tr>
</xsl:template>
```

The final result of this process is the HTML file shown here.

```
<html>
  <body>
    <table border="1">
      <tr>
        <td>1</td>
        <td>Chair</td>
        <td>49.33</td>
      </tr>
      <tr>
        <td>2</td>
        <td>Car</td>
        <td>43398.55</td>
      </tr>
      <tr>
        <td>3</td>
        <td>Fresh Fruit Basket</td>
        <td>49.99</td>
      </tr>
    </table>
  </body>
</html>
```

We'll take a look at how this output appears in an Internet browser in the next section.

Generally speaking, if you aren't sure that you need XSL, you probably don't. The .NET framework provides a rich set of tools for searching and manipulating XML files the easy way, which is the

best approach for small-scale XML use.

The Xml Web Control

Instead of manually writing an HTML file, you might want to display the information in a page. The XmlTransform file just converts XML files—it doesn't provide any way to insert the output into your web page. However, ASP.NET includes a web control called Xml (found in the System.Web.UI.WebControls namespace) that displays XML content. You can specify the XML content for this control in several ways: by assigning an XmlDocument object to the Document property, by assigning a string containing the XML content to the DocumentContent property, or by specifying a string that refers to an XML file using the DocumentSource property.

```
' Display the information from an XML file in the Xml control.
MyXml.DocumentSource = Request.ApplicationPath & _
  "\SuperProProductList.xml"
```

If you assign the SuperProProductList.xml file to the Xml control, you're likely to be disappointed. The result is just a string of the inner text (the price for each product), bunched together without a space (see Figure 17-11).

Figure 17-11: Unformatted XML content

However, you can also apply an XSL transformation, either by assigning an XslTransform object to the Transform property or by using a string that refers to the XSLT file with the TransformSource property.

```
' Specify a XSLT file.
MyXml.TranformSource = Request.ApplicationPath & _
  "\SuperProProductList.xslt"
```

Now the output is automatically formatted according to your stylesheet (see Figure 17-12).

Figure 17-12: Transformed XML content

XML in ADO.NET

The integration between ADO.NET and XML is straightforward and effortless, but it's not likely to have a wide range of usefulness. Probably the most interesting feature is the ability to serialize a DataSet to an XML file (with or without a corresponding XSD schema file).

```
' Save a DataSet.
MyDataSet.WriteXml(Request.ApplicationPath & "datasetfile.xml")

' Save a DataSet schema.
MyDataSet.WriteSchema(Request.ApplicationPath & "datasetfile.xsd")
```

It's then very easy to retrieve the complete DataSet later on.

```
' Retrieve the DataSet with its schema (just to be safe, and make
' sure no data types are corrupted or tags have been changed).
Dim MyDataSet As NewDataSet()
MyDataSet.ReadXmlSchema(Request.ApplicationPath & "datasetfile.xsd")
MyDataSet.ReadXml(Request.ApplicationPath & "datasetfile.xml")
```

This technique could be useful for permanently storing the results of a slow-running query. A web page that needs this information could first check for the presence of the file before trying to connect to the database. This type of system is similar to many homegrown solutions used in traditional ASP programming. It's useful, but it raises additional issues.

For example, every web page that needs the data must check the file age to determine whether it is still valid. Presumably, the XML file will need to be refreshed at periodic intervals, but if more than one executing web page finds that the file needs to be refreshed and tries to create it at the same time, a file access problem will occur. All these problems can be resolved with a little painful coding, although caching provides a more streamlined and elegant solution. It also offers much better performance, because the data is stored in memory. Caching is described in Chapter 23.

Of course, there could be some cases where you need to exchange the results of a query with an application on another platform, or when you need to store the results permanently (caching information will be automatically removed if it is not being used or the server memory is becoming scarce). In these cases, the ability to save a DataSet can come in quite handy. But whatever you do, don't try to work with the retrieved DataSet and commit changes back to the data source. Handling concurrency issues is difficult enough without trying to use stale data from a file!

The XmlDataDocument

Another option provided by the DataSet is the ability to access it through an XML interface. If you wanted to do this with a database, you would need to perform the slightly awkward work of saving the DataSet to a file and then retrieving it into an XmlDataDocument. You would also lose your ability to enforce many relationships, unique constraints, and have a much more difficult time managing data updates.

A more useful way to use the XmlDataDocument is to access an ordinary XML file as though it is a database. This means you don't need to work with the XML syntax we have explored, and you don't need to go to the work of extracting all the information and creating the appropriate objects and collections. Instead, you can use the familiar ADO.NET objects to manipulate the underlying XML document.

Consider these remarkable few lines of code:

```
Dim DataDoc As New XmlDataDocument()

' Set the schema and retrieve the data.
```

```
DataDoc.DataSet.ReadXmlSchema(Request.ApplicationPath & _
  "\SuperProductList.xsd")
DataDoc.Load(Request.ApplicationPath & "\SuperProductList.xml")

' Display the retrieved data.
dgResults.DataSource = DataDoc.DataSet
dgResults.DataBind()
```

In this example, a new XmlDataDocument is created, an XML file is loaded into it, and the information is automatically provided through a special DataSet that is "attached" to the XmlDataDocument. This allows you to use the ADO.NET objects to work with data (like the DataTable and DataRow objects), as well as the standard XmlDocument members (like the GetTagsByElementName method and the ChildNodes property). You could even use both approaches at the same time, because both the XML and the ADO.NET interface access the same underlying data. The retrieved data is shown in Figure 17-13.

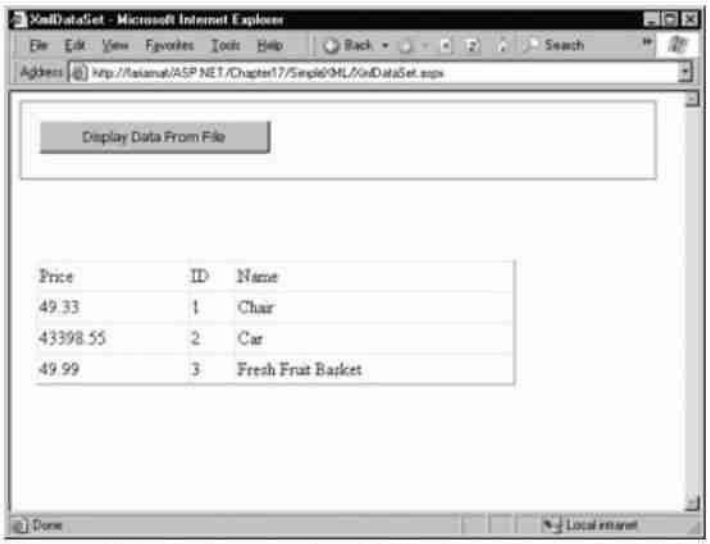

Figure 17-13: Accessing XML data through ADO.NE

The only catch is that your XML file must have a schema in order for ASP.NET to be able to determine the proper structure of your data. Before you retrieve the XML file, you must use the XmlDataDocument.DataSet.ReadXmlSchema method. Otherwise, you won't be able to access the data through the DataSet.

Clearly, if you need to store hierarchical data in an XML file and are willing to create an XSD schema, the XmlDataDocument is a great convenience. It also allows you to bridge the gap between ADO.NET and XML—some clients can look at the data as XML, while others can access it through the database objects without losing any information or requiring an error-prone conversion process.