Introducing .NET Data Management

We've looked at the basics of Microsoft's new .NET Framework and ASP.NET in particular. It changes the way you program with ASP, adding a whole range of new techniques that make it easier to create dynamic pages, web services, and web applications. However, there is one fundamental aspect of almost all applications that we've not yet explored. This is how we access and work with data that is stored in other applications or files. In general terms, these sources of information are called *data stores*. This chapter looks at how the .NET Framework provides access to the many different kinds of data store that you may have to interface with.

The .NET Framework includes a series of classes that implement a new data access technology that is specifically designed for use in the .NET world. We'll look at why this has come about, and how it relates to the techniques used in ASP. In fact, the new framework classes provide a whole lot more than just a .NET version of ADO. Like the move from ASP to ASP.NET, they involve fundamental changes in the approach to managing data in external data stores.

While *data management* is often assumed to relate to relational data sources such as databases, we will also explore the other types of data that are increasingly encountered today. There is extended support within .NET for working with *Extensible Markup Language* (*XML*) and its associated technologies. Apart from comprehensive support for the existing XML standards, .NET provides new ways to handle XML. These include integration between XML and traditional relational data access methods.

So, the topics for this chapter are:

- The various types of data storage used today and into the future
- The need for another data access technology
- An overview of the new relational data access techniques in .NET

- An overview of the new techniques for working with XML in .NET
- Choosing an appropriate data access technology and a data format

Let's start with a look at the way data is stored and accessed today.

Data Stores and Data Access

The term *data store* usually meant a database of some kind. Databases were usually file-based, often using fixed-width records written to disk – rather like text files. A database program or data access technology read the files into buffers as tables, and applied rules defined in other files to connect the records from different tables together. As technologies matured, relational databases evolved to provide better storage methods, such as variable-length records and more efficient access techniques.

However, the basic storage medium was still the *database* – a specialist program that managed the data and exposed it to clients. Obvious examples are Oracle, Informix, Sybase, DB2, and Microsoft's own SQL Server. All are enterprise-oriented applications for storing and managing data in a relational way.

At the same time, *desktop* database applications matured and became more powerful. In general, this type of program provides its own interface for working with the data. For example, Microsoft Access can be used to build forms and queries that can access and display data in very powerful ways. They often allow the data to be separated from the interface over the network, so that it can reside on a central server. But, again, we're still talking about relational databases.

Moving to a Distributed Environment

In recent years, the requirements and mode of operation of most businesses have changed. Without consciously realizing it, we've moved away from relying on a central relational database to store all the data that a company produces and needs to access. Now, data is stored in email servers, directory services, Office documents, and other places – as well as the traditional relational database.

The move to a more distributed computing paradigm means that the central data store, running on a huge computer in an air-conditioned IT department, is often only a part of the whole corporate data environment. Modern data access technologies need to be able to work with a whole range of different types of data store, as shown in Figure 8-1.

You can see that the range of storage techniques has become quite quite wide. It's easy to see why the term *database* is no longer appropriate for describing the many different ways that data is often stored today. Distributed computing means that we have to be able to extract data in a suitable format, move it around across a range of different types of network, and change the format of the data to suit many different types of client device.

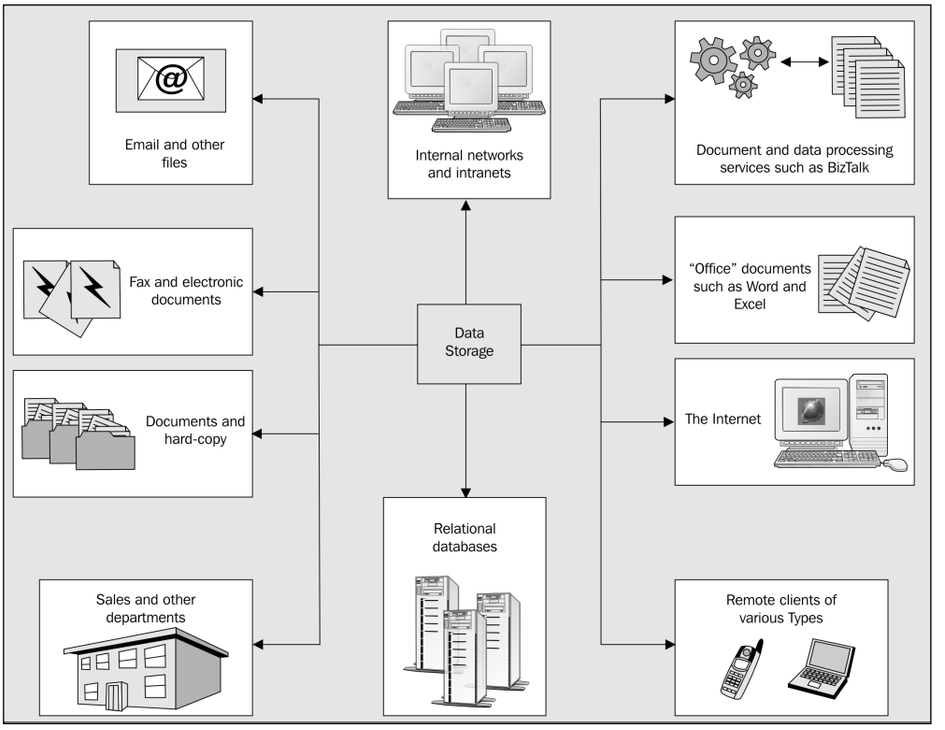

Figure 8-1

The next section explores one of the areas where data storage and management is changing completely – the growth in the use of *XML*.

XML – A Data Format for the Future?

One of the most far-reaching of the new ideas in computing is the evolution of XML. The *World Wide Web Consortium* (*W3C*) issued proposals for XML some three years ago (at the time of writing), and these have matured into standards that are being adopted by almost every sector of the industry.

XML scores when it comes to storing and transferring data – it is an accepted industry standard, and it is just plain text. The former means that we have a way of transferring and exposing information in a format that is independent of platform, operating system, and application. Compare this to, for example, the MIME-encoded recordsets that Internet Explorer's *Remote Data Service* (*RDS*) uses. Instead, XML means that you don't need a specific object to handle the data. Any manufacturer can build one that will work with XML data, and developers can use one that suits their platform, operating system, programming language, or application.

XML is just plain text, and so you no longer have to worry about how to store and transport it. It can be sent as a text file over the Internet using HTTP (which is effectively a 7-bit only transport protocol). You

don't have to encode it into a MIME or UU-encoded form. You can also write it to a disk as a text file, or store it in a database as text. OK, so it often produces a bigger file than the equivalent binary representation, but compression and the availability of large cheap disk drives generally compensate for this.

Applications have already started exposing data as XML in many ways. For example, Microsoft SQL Server 2000 includes features that allow you to extract data directly as XML documents, and update the source data using XML documents. Databases such as Oracle 8i and 9i are designed to manipulate XML directly, and the most recent office applications like Word and Excel will save their data in XML format either automatically or on demand.

XML is already directly ingrained into many applications. ASP.NET uses XML format configuration files, and web services expose their interface and data using an implementation of XML called the *Simple Object Access Protocol* (*SOAP*).

Other XML Technologies

As well as being a standard in itself, XML has also spawned other standards that are designed to interoperate with it. Two common examples are *XML Schemas*, which define the structure and content of XML documents, and the *Extensible Stylesheet Language for Transformation* (*XSLT*), which is used to perform transformations of the data into new formats.

XML schemas also provide a way for data to be expressed in specific XML formats that can be understood globally, or within specific industries such as pharmaceuticals or accountancy applications. There are also several server applications that can transform and communicate XML data between applications that expect different specific formats (or, in fact, other non-XML data formats). In the Microsoft world, this is BizTalk Server, and there are others such as Oasis and Rosetta for other platforms.

Just Another Data Access Technology?

To quote a colleague of mine, "Another year, another Microsoft data access technology". We've just got used to *ActiveX Data Objects* (*ADO*), and it's all-change time again. Is this some fiendish plan on Microsoft's behalf to keep us on our toes, or is there a reason why the technology that seemed to work fine in previous versions of ASP is no longer suitable?

In fact, there are several reasons why we really need to move on from ADO to a new technology. We'll examine these next, then later on take a high-level view of the changes that are involved in moving from ADO to the new .NET Framework data access techniques.

.NET Means Disconnected Data

You've seen a bit about how relational databases have evolved over recent years. However, it's not just the data store that has evolved – it's the whole computing environment. Most of the relational databases still in use today were designed to provide a solid foundation for the client-server world. Here, each client connects to the database server over some kind of permanent network connection, and remains connected for the duration of their session.

For example, with Microsoft Access, the client opens a *Form* window (often defined within their client-side interface program). This form fetches and caches some or all of the data that is required to populate the controls on the form from the server-side database program, and displays it on the client. The user can manipulate the data, and save changes back to the central database over their dedicated connection.

For this to work, the server-side database has to create explicit connections for each client, and maintain these while the client is connected. As long as the database software and the hardware it is running on are powerful enough for the anticipated number of clients, and the network has the bandwidth and stability to cope with the anticipated number of client connections, it all works very well.

But when this is moved to the disconnected world of the Internet, it falls apart very quickly. The concept of a stable and wide-band connection is hard enough to imagine, and the need to keep this connection permanently open can quickly cause problems to appear. It's not so bad if you are operating in a limited-user scenario, but for a public web site, it's obviously not going to work out.

In fact, there are several aspects to being disconnected. The nature of the HTTP protocol that is used on the Web means that connections between client and server are only made during the transfer of data or content. They aren't kept open after a page has been loaded or a recordset has been fetched.

On top of this, there is often a need to use the data extracted from a data store while not even connected to the Internet at all. Maybe while the user is traveling with a laptop computer, or the client is on a dial-up connection and needs to disconnect while working with the data then reconnect again later.

This means that we need to use data access technologies where the client can access, download, and cache the data required, then disconnect from the database server or data store. Once the clients are ready, they then need to be able to reconnect and update the original data store with the changes.

Disconnected Data in N-Tier Applications

Another aspect of working with disconnected data arises when you move from a client-server model into the world of *n*-tier applications. A distributed environment implies that the client and the server are separate, connected by a network. To build applications that work well in this environment, you can use a design strategy that introduces more granular differentiation between the layers, or *tiers*, of an application.

As Figure 8-2 shows, it's usual to create components that perform the data access in an application (the *data tier*), rather than having the ASP code hit the data store directly. There is often a series of rules (usually called *business rules*) that have to be followed, and these can be implemented within components.

They might be part of the components that perform the data access, or they might be separate – forming the *business tier* (or application logic tier). There may also be a separate set of components within the client application (the *presentation tier*) that perform specific tasks for managing, formatting, or presenting the data.

The benefits of designing applications along these lines are many, such as reusability of components, easier testing, and faster development.

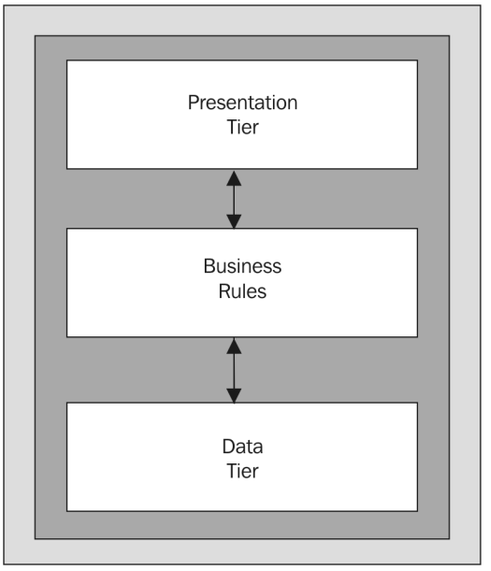

Figure 8-2

Let's take a look at how this influences the process of handling data.Within an *n*-tier application, the data must be passed between the tiers as each client request is processed. So, the data tier connects to the data store to extract the data, perhaps performs some processing upon it, and then passes it to the next tier. At this point, the data tier will usually disconnect from the data store, allowing another instance (another client or a different application) to use the connection.

By disconnecting the retrieved data from the data store at the earliest possible moment, we improve the efficiency of the application and allow it to handle more concurrent users. However, it again demonstrates the need for data access technologies that can handle disconnected data in a useful and easily manageable way – particularly when we need to update the original data in the data store.

The Evolution of ADO

Pre-ADO data access technologies, such as *Data Access Objects* (*DAO*) and *Remote Data Objects* (*RDO*) were designed to provide open data access methods for the client-server world – and are very successful in that environment. For example, if you build Visual Basic applications to access SQL Server over your local network, they work well.

However, with the advent of ASP 1.0, it was obvious that something new was needed. It used only active scripting (such as VBScript and JScript) within the pages, and for these a simplified ActiveX or COM-based technology was required. The answer was ADO 1.0, included with the original ASP installation. ADO allows you to connect to a database to extract recordsets, and perform updates using the database tables, SQL statements, or stored procedures within the database.

However, ADO 1.0 was really only an evolution of the existing technologies, and offered no solution for the disconnected problem. You opened a recordset while you had a connection to the data store, worked

with the recordset (maybe updating it or just displaying the contents), then closed it and destroyed the connection. Once the connection was gone, there was no easy way to reconnect the recordset to the original data.

To some extent, the disconnected issue was addressed in ADO 2.0. A new recordset object allowed you to disconnect it from the data store, work with the contents, then reconnect and flush the changes back to the data store. This worked well with relational databases such as SQL Server, but was not always an ideal solution. It didn't provide the capabilities to store relationships and other details about the data – basically all you stored was the rowset containing the values.

Another technique that came along with ADO 2.0 was the provision of a *Data Source Object* (*DSO*) and *Remote Data Services* (*RDS*) that could be used in a client program such as Internet Explorer to cache data on a client. A recordset can be encoded as a special MIME type and passed over HTTP to the client where it is cached. The client can disconnect and then reconnect later and flush changes back to the data store. However, despite offering several useful features such as client-side data binding, this non-standard technique never really caught on – mainly due to the reliance on specific clients and concerns over security.

To get around all these limitations, the .NET Framework data access classes have been designed from the ground up to provide a reliable and efficient disconnected environment for working with data from a whole range of data stores.

.NET Means XML Data

As you saw earlier in this chapter, the computing world is moving ever more towards the adoption of XML as the fundamental data storage and transfer format. ADO 1.0 and 2.0 had no support for XML – it wasn't around as anything other than vague proposals at that time. In fact, at Microsoft, it was left to the Internet Explorer team to come up with the first tools for working with XML – the MSXML parser that shipped with IE 5 and other applications.

Later, MSXML became part of the ADO team's responsibilities and surfaced in ADO 2.1 and later as an integral part of *Microsoft Data Access Components* (*MDAC*). Along with it, the DSO used for remote data management and caching had XML support added. Methods were also added to the integral ADO objects.

The `Recordset` object gained methods that allowed it to load and save the content as XML. However, it was never anything more than an add-on, and the MSXML parser remained distinct from the core ADO objects.

Now, to bring data access up to date in the growing world of XML data, .NET includes a whole series of objects that are specifically designed to manage and manipulate XML data. This includes native support for XML formatted data within objects like the `Dataset`, as well as a whole range of objects that integrate a new XML parsing engine within the framework as a whole.

.NET Means Managed Code

As mentioned before, the .NET Framework is not a new operating system. It's a series of classes and a managed runtime environment within which code can be executed. The framework looks after all the

complexities of garbage collection, caching, memory management and so on – but only as long as you use managed code. Once you step outside this cozy environment, the efficiency of your applications reduces (the execution has to move across the process boundaries into unmanaged code and back).

The existing ADO libraries are all unmanaged code, and so we need a new technology that runs within the .NET Framework. While Microsoft could just have added managed code wrappers to the existing ADO libraries, this would not have provided an ideal or efficient solution.

Instead, the data access classes within .NET have been designed from the ground up as managed code. They are integral to the framework and so provide maximum efficiency. They also include a series of objects that are specifically designed to work with MS SQL Server, using the native *Tabular Data Stream* (*TDS*) interface for maximum performance. Alternatively, managed code OLEDB, ODBC and Oracle providers are included with the framework to allow connections to all kinds of other data stores.

.NET Means a New Programming Model

One of the main benefits of moving to .NET is the ability to get away from the mish-mash of HTML content and script code that traditional ASP always seems to involve. Instead, there is a whole new structured programming model and approach to follow. You should use server controls (and user controls) to create output that is automatically tailored to each client, and react to events that these controls raise on the server.

Write in *proper* languages, and not script. Instead of VBScript, you can use Visual Basic, C#, as well as a compiled version of the JScript language. And, if you prefer, you can use C++, COBOL, Active Perl, or any one of the myriad other languages that are available or under development for the .NET platform.

This move to a structured programming model with server controls and event handlers provides improvements over existing data handling techniques using traditional ADO. For example, in ADO you need to iterate through a recordset to display the contents. However, the .NET Framework provides extremely useful server controls such as the `DataGrid`, which look after displaying the data themselves – all they need is a data source such as a set of records (a *rowset*).

So, instead of using `Recordset`-specific methods like `MoveNext` to iterate through a rowset, and access each field in turn, you just bind the rowset to the server control. It carries out all the tasks required to present that data, and even makes it available for editing. Yet, if required, you can still access data as a read-only and forward-only rowset using the new `DataReader` object instead. Overall, the .NET data access classes provide a series of objects that are better suited to working with data using server controls, as well as manipulating it directly with code.

Introducing Data Management in .NET

Having seen why we need a new data access technology, let's look at what .NET actually provides. In this section, you'll get a high-level overview of all of the .NET data management classes, and see how each of the objects fits with the disconnected and structured programming environment that .NET provides. The remainder of this chapter is divided into two sections; relational data management (techniques such as those you used traditional ADO for) and XML data management (for which, traditionally, you would use an XML parser such as MSXML).

System Namespaces for Data Management

The new relational data management classes are in a series of namespaces based on `System.Data` within the class library. The combination of the classes from the namespaces in the following table is generally referred to as *ADO.NET*:

Namespace	Description
`System.Data`	Contains the basic objects and public classes used for accessing and storing relational data, such as `DataSet`, `DataTable`, and `DataRelation`. Each of these is independent of the type of data source, and independent of the connection type.
`System.Data.Common`	Contains the base classes used by other public classes in the provider-specific namespaces, in particular those used by the classes in the `OleDb`, `Odbc`, `OracleClient`, and `SqlClient` namespaces. In general, this namespace is not specifically imported into applications.
`System.Data.Odbc`	Contains the public classes used to connect to and work with a data source via an ODBC driver, such as `OdbcConnection` and `OdbcCommand`. These objects inherit properties, methods, and events from the base classes in the `Common` namespace.
`System.Data.OleDb`	Contains the public classes used to connect to and work with a data source via an OLE-DB provider, such as `OleDbConnection` and `OleDbCommand`. These objects inherit properties, methods, and events from the base classes in the `Common` namespace.
`System.Data.OracleClient`	Contains the public classes used to connect to and work with an Oracle database, such as `OracleConnection` and `OracleCommand`. These objects inherit properties, methods, and events from the base classes in the `Common` namespace, and add Oracle-specific features as well.
`System.Data.SqlClient`	Contains the public classes used to connect to and work with a data source via the TDS interface of Microsoft SQL Server (only), using classes such as `SqlConnection` and `SqlCommand`. These classes provide better performance by removing some of the intermediate layers required by an OLEDB or ODBC provider. These objects inherit properties, methods, and events from the base classes in the `Common` namespace.
`System.Data.SqlServerCe`	Contains the public classes used to connect to and work with a data source running under Windows CE. These classes are not used or discussed in this book.
`System.Data.SqlTypes`	Contains public classes to implement the data types normally found in relational databases such as SQL Server, and which are different to the standard .NET data types. Examples are `SqlMoney`, `SqlDateTime`, and `SqlBinary`. Using these can improve performance and avoid type conversion errors.

There is also a separate series of namespaces containing the classes used to work with XML rather than relational data. These namespaces are based on `System.Xml`:

Namespace	Description
`System.Xml`	Contains the public classes required to create, read, store, write, and manipulate XML documents in line with W3C recommendations. Includes `XmlDocument` and a series of classes that represent the various types of node in an XML document.
`System.Xml.Schema`	Contains the public classes required to create, store, and manipulate XML schemas, and the nodes that they contain.
`System.Xml.Serialization`	Contains public classes that can be used to convert XML documents to other persistence formats, such as SOAP, for streaming to disk or across the wire.
`System.Xml.XPath`	Contains the public classes required to implement reading, storing, writing, and querying XML documents using a fast custom XPath-based document. Includes `XPathDocument`, `XPathNavigator`, and classes that represent XPath expressions.
`System.Xml.Xsl`	Contains the public classes required to transform XML into other formats using XSL or XSLT stylesheets. The main object is `XslTransform`.

Importing the Required Namespaces

Pages that use objects from the framework's class libraries must import the namespaces containing all the classes that they explicitly create instances of. Many of the common namespaces are imported by default, but this does not include the data management namespaces.

To use any type of data access code, you must import the appropriate namespace.

Importing the System.Data Namespaces

To access relational data, you need at least `System.Data` *and* either `System.Data.OleDb`, `System.Data.SqlClient`, or `System.Data.Odbc` (depending on the way you're connecting to the data source). In ASP.NET, the `Import` page directive is used:

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

Or:

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

In Visual Basic .NET code inside a class or module, use the `Imports` statement:

```
Imports System.Data
Imports System.Data.OleDb
```

In C#, use the `using` statement:

```
using System.Data;
using System.Data.OleDb;
```

At times when you need to specifically import other `System.Data` namespaces. For example, to create a new instance of a `DataTableMapping` class, you need to import the `System.Data.Common` namespace, and to use an SQL-specific data type, you need to import the `System.Data.SqlTypes` namespace.

Importing the System.Xml Namespaces

To access XML data using the objects in the framework class library, you can often get away with importing just the basic `System.Xml` namespace. However, to create an `XPathDocument` instance, you have to import the `System.Xml.XPath` namespace as well. To use the `XslTransform` class to perform server-side transformations of XML documents, you need to import the `System.Xml.Xsl` namespace.

The `System.Xml.Schema` namespace is usually only required when working with collections of schemas. Most XML validation objects are in `System.Xml`, so you can create an `XmlValidatingReader` (for example) without referencing the `System.Xml.Schema` namespace. But to create a new `SchemaCollection` instance, you must import the `System.Xml.Schema` namespace.

Type-Not-Found Compilation Errors

If you forget to import any required namespace, you'll get an error as that shown in Figure 8-3. In this case, it indicates that you have forgotten to import the namespace that contains the class for `OleDbConnection`. To solve this particular error, you just need to import the namespace `System.Data.OleDb`.

To find out which namespace contains a particular class, you can simply look in the .NET SDK Class Library section within the Reference section, or search for the object/class name using the *Index* or *Search* feature of the SDK. Alternatively, use the excellent *WinCV* (Windows Class Viewer) tool that comes with the .NET installation.

> *For help on using the tools that come with .NET, check out the SDK section .NET Framework Tools from within the Tools and Debugger section. The WinCV utility is described in detail in the subsection Windows Forms Class Viewer (`Wincv.exe`).*

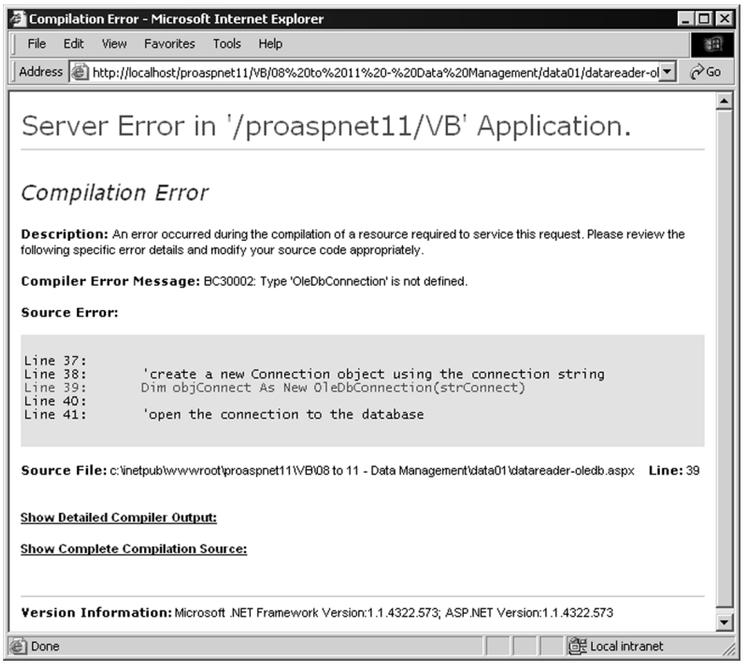

Figure 8-3

The Fundamental ADO.NET Classes

Traditional data access with ADO revolves around one fundamental data storage object – the `Recordset`. The technique used here is to create a connection to a data store using either an OLEDB provider or an ODBC through OLEDB driver (depending on the data store and the availability of the provider) and then execute commands against that connection to return a `Recordset` object containing the appropriate data. This can be done using a `Command` object or directly against the `Connection` object. Alternatively, to insert or update the data, just execute a SQL statement or a stored procedure within the data store using the `Connection` object or `Command` object directly, without returning a `Recordset` object.

Data access in .NET follows a broadly similar principle, but uses a different set of objects. So, switching to .NET does not involve learning a completely different technique. However, the objects used are quite different underneath, providing much better performance with more flexibility and usability.

The .NET data access object model is based around two fundamental objects – the `DataReader` and the `DataSet`. Together, they replace the `Recordset` from traditional ADO, providing many new features that make complex data access techniques much more efficient, while remaining as easy to use as the `Recordset` object. The main differences are that a `DataReader` provides forward-only and read-only access to data (like a *firehose* cursor in ADO), while the `DataSet` object can hold more than one table (in

other words, more than one rowset) from the same data source as well as the relationships between them.

You can create a `DataSet` from existing data in a data store, or fill it directly with data one row at a time using code. It also allows you to manipulate the data held in the `DataSet`'s tables, and build as well as modify the relationships between the tables within it.

Each table within a `DataSet` maintains details of the original values of the data as you work with it, and any changes to the data can be pushed back into the data store at a later date.

The `DataSet` also contains metadata describing the table contents, such as the columns types, rules, and keys. Remember that the whole ethos with a `DataSet` is to be able to work accurately and efficiently in a disconnected environment.

The `DataSet` object can also persist its contents, including more than one data table or rowset, directly as XML, and load data from an XML document that contains structured data in the correct format. In fact, XML is the *only* persistence format for data in .NET – bringing it more into line with the needs of disconnected and remote clients.

Comparison of Techniques in ADO and ADO.NET

As we expect most of our readers to be at least partly familiar with traditional ADO programming techniques, we will start with a quick overview of how the new ADO.NET classes and techniques relate to the traditional approach:

Traditional ADO approach	ADO.NET equivalent
Connected access to data using a `Connection` (and possibly a `Command` as well) to fill a `Recordset` then iterate through the `Recordset`.	Use a `Connection` and a `Command` to connect a `DataReader` object to the data store and read the results iteratively from the data store.
Updating a data store using a `Connection` and `Command` object to execute a SQL statement or stored procedure.	Use a `Connection` and a `Command` to connect to the data store and execute the SQL statement or stored procedure.
Disconnected access to data using a `Connection` (and possibly a `Command` as well) to fill a `Recordset` then remove the connection to the data source.	Use a `Connection` and a `Command` to connect a `DataAdapter` to the data source and then fill a `DataSet` with the results.
Updating a data store from a disconnected `Recordset` by reconnecting and using the `Update` or `UpdateBatch` method.	Use a `Connection` and a `Command` to connect a `DataAdapter` and `DataSet` to the data source and then call the `Update` method of the `DataAdapter`.

The major differences are:

- There is no direct equivalent of a `Recordset` class. Depending on the task you want to achieve, you use a `DataReader` or a `DataSet` instead.
- Client-side and server-side (database) cursors are not used in ADO.NET. The disconnected model means that they are not applicable.
- Database locking is not supported or required. Again, due to the disconnected model, it is not applicable.
- All data persistence is as XML. There are no MIME-encoded or binary representations of rowsets or other data structures.

Let's look at the new ADO.NET classes in more detail.

The Connection Classes

These classes are similar to the ADO `Connection` class, with similar properties. They are used to connect a data store to a `Command` instance.

- The `OleDbConnection` class is used with an OLE-DB provider
- The `SqlConnection` class uses Tabular Data Services (TDS) with MS SQL Server
- The `OdbcConnection` class is used with an ODBC driver
- The `OracleConnection` class is used to connect to an Oracle database

In traditional ADO, it was common to use the `Connection` to directly execute a SQL statement against the data source or to open a `Recordset`. This *cannot* be done with the .NET `Connection` classes. However, they do provide access to transactions that are in progress against a data store.

The Commonly Used Methods of the Connection Classes

The most commonly used methods for the `OleDbConnection`, `OdbcConnection`, `OracleConnection`, and `SqlConnection` classes are shown in the following table:

Method	Description
`Open`	Opens a connection to the data source using the current settings for the properties, such as `ConnectionString` that specifies the connection information to use
`Close`	Closes the connection to the data source
`BeginTransaction`	Starts a data source transaction and returns a `Transaction` instance that can be used to commit or abort the transaction.

An excellent reference to all the properties, methods, and events of the classes discussed here is included within the .NET SDK that is provided with the framework. Simply open the **Class Library** *topic within the* **Reference** *section, or search for the class by name using the* **Index** *or* **Search** *feature of the SDK. Many of the common ones have been demonstrated, including those shown in the preceding table.*

Remember that there are at least two implementations of some of the .NET data access classes, each one being specific to the data store you are connecting to.

Classes prefixed with `OleDb` or `Odbc` are used with a managed code OLEDB provider or ODBC driver against any database that has a suitable provider or driver. The classes prefixed with `Sql` are used only with Microsoft SQL Server (we'll concentrate on just these three types of data store connection).

Other than that, the classes are identical as far as programming with them is concerned. However, you must use the appropriate one depending on which data store you connect to, so your code must be rewritten to use the correct ones if you change from one set of classes to the other.

This is generally only a matter of changing the prefixes in the class declarations. For this reason, you may prefer to avoid including the prefix in your variable and method names, and in comments within your code.

As an aside, it is possible to use the .NET Activator class's `CreateInstance` method to create an instance of a class using a variable to specify the class name. This would allow generic code routines to be created that instantiate the correct class type (`OleDb` or `Sql`) depending on some external condition you specify. The details of this topic can be found in the SDK.

The Command Classes

These classes are similar to the equivalent ADO `Command`, and have similar properties. They are used to connect the `Connection` class to a `DataReader` or a `DataAdapter` instance:

- The `OleDbCommand` class is used with an OLE-DB provider.
- The `SqlCommand` class uses Tabular Data Services with MS SQL Server.
- The `OdbcCommand` class is used with an ODBC driver.
- The `OracleCommand` class is used to access an Oracle database.

The `Command` class allows you to execute a SQL statement or stored procedure against a data source. This includes returning a rowset (in which case you use another class such as a `DataReader` or a `DataAdapter` to access the data), returning a single value (a *singleton*), or returning a count of the number of records affected for queries that do not return a rowset.

The Commonly Used Methods of the Command Classes

The most commonly used methods for the `OleDbCommand`, `OdbcCommand`, `OracleCommand`, and `SqlCommand` classes are shown in the following table:

Method	Description
`ExecuteNonQuery`	Executes the command defined in the `CommandText` property against the connection defined in the `Connection` property for a query that does not return any rows (an `UPDATE`, `DELETE`, or `INSERT`). Returns an `Integer` indicating the number of rows affected by the query.
`ExecuteReader`	Executes the command defined in the `CommandText` property against the connection defined in the `Connection` property. Returns a "reader" instance that is connected to the resulting rowset within the database, allowing the rows to be retrieved. The derivative `ExecuteXmlReader` method can be used with the SQL Server 7.0 SQLXML technology to return an XML document fragment in an `XmlReader` instance. We look at the various "reader" classes later.
`ExecuteScalar`	Executes the command defined in the `CommandText` property against the connection defined in the `Connection` property. Returns only a single value (effectively the first column of the first row of the resulting rowset). Any other returned columns and rows are discarded. Fast and efficient when only a "singleton" value is required.

The DataAdapter Classes

Some classes in the framework connect one or more `Command` instances to a `Dataset`. They provide the pipeline and logic that fetches the data from the data store and populates the tables in the `DataSet`, or pushes the changes in the `DataSet` back into the data store.

- The `OleDbDataAdapter` class is used with an OLE-DB provider.
- The `SqlDataAdapter` class uses Tabular Data Services with MS SQL Server.
- The `OdbcDataAdapter` class is used with an ODBC driver.
- The `OracleDataAdapter` class is used to access an Oracle database.

These classes provide four properties that define the commands used to manipulate the data in a data store: `SelectCommand`, `InsertCommand`, `UpdateCommand`, and `DeleteCommand`.

Each one of these properties is a reference to a `Command` instance (these `Command` instances can all share the same `Connection` instance). Figure 8-4 shows how these classes are related:

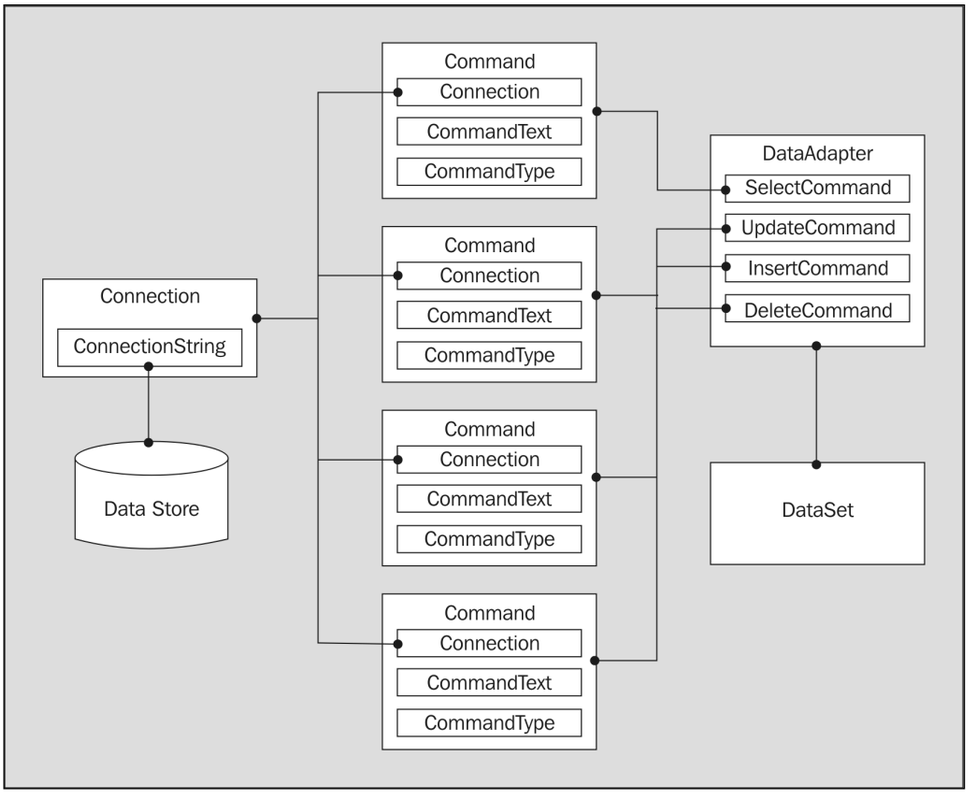

Figure 8-4

The Commonly Used Methods of the DataAdapter Classes

The `OleDbDataAdapter`, `OdbcDataAdapter`, `OracleDataAdapter`, and `SqlDataAdapter` classes provide a series of methods for working with the `DataSet` that they apply to. The three most commonly used methods are shown in the following table:

Method	Description
`Fill`	Executes the `SelectCommand` to fill the `DataSet` with data from the data source. Can also be used to update (refresh) an existing table in a `DataSet` with changes made to the data in the original data source if there is a primary key in the table in the `DataSet`.
`FillSchema`	Uses the `SelectCommand` to extract just the schema for a table from the data source, and creates an empty table in the `DataSet` with all the corresponding constraints.

Table continued on following page

Method	Description
`Update`	Calls the respective `InsertCommand`, `UpdateCommand`, or `DeleteCommand` for each inserted, updated, or deleted row in the `DataSet` so as to update the original data source with the changes made to the content of the `DataSet`. This is a little like the `UpdateBatch` method provided by the ADO `Recordset`, but in the `DataSet` it can be used to update more than one table.

The DataSet Class

The `DataSet` provides the basis for disconnected storage and manipulation of relational data. You can fill it from a data store, work with it while disconnected from that data store, then reconnect and flush changes back to the data store as required. The main differences between a `DataSet` and the ADO `Recordset` are:

- The `DataSet` class can hold more than one table (more than one rowset in other words), as well as the relationships between them.
- The `DataSet` class automatically provides disconnected access to data.

Consider the following schematic:

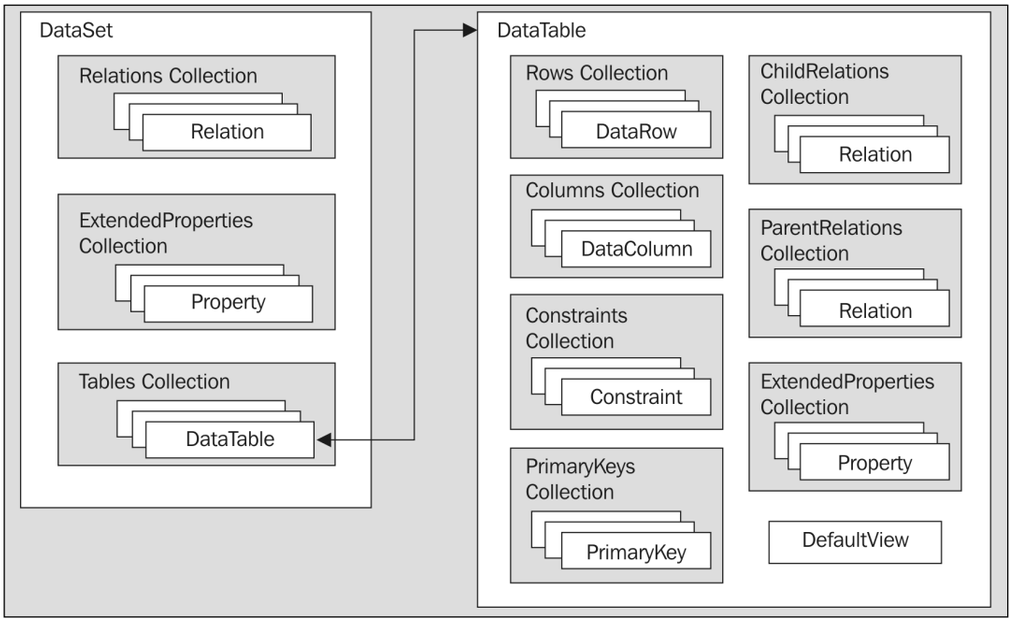

Figure 8-5

Figure 8-5 shows a schematic view of the relationship between all the classes discuss now. Each table in a `DataSet` is a `DataTable` instance within the `Tables` collection. Each `DataTable` contains a collection of `DataRow` instances and a collection of `DataColumn` instances. There are also collections for the primary keys, constraints, and default values used in this table (the `Constraints` collection), and the parent and child relationships between the tables.

There is also a `DefaultView` instance for each table. This is used to create a `DataView` based on the table, so that the data can be searched, filtered or otherwise manipulated – or bound to a control for display (we look at the `DataTable` and `DataView` classes later).

The Commonly Used Methods of the DataSet Class

The `DataSet` class exposes a series of methods that can be used to work with the contents of the tables, or the relationships between them. For example, you can clear the `DataSet`, or merge data from a separate `DataSet` into this one. The following table summarizes the methods available:

Method	Description
`Clear`	Removes all data stored in the `DataSet` by emptying all of the tables it contains. However, it is often more efficient to destroy the instance and create a new one unless you need to hold a reference to the existing one.
`Merge`	Takes the contents of a `DataSet` and merges it with another `DataSet` so that it contains all the data from both of the source `DataSet` instances.

We mentioned earlier that the default persistence format in .NET is XML. The following table shows the methods provided by the `DataSet` class for reading and writing this XML data.

Methods	Description
`ReadXml` *and* `ReadXmlSchema`	Takes an XML document or an XML schema and reads it into the `DataSet`.
`GetXml` *and* `GetXmlSchema`	Returns a `String` containing an XML document or an XML schema that represents the data in the `DataSet`.
`WriteXml` *and* `WriteXmlSchema`	Writes the XML document or XML schema that represents the data in the `DataSet` to a disk file, to a "reader/writer" instance, or to a `Stream`. We look at the "reader/writer" classes later.

The `DataSet` class, together with all the `DataTable` instances it contains, keeps a record of the values for the content when it was originally created and loaded (filled with data). This is a fundamental requirement to allow the changes to be pushed back into the original data store in a multi-user scenario.

There are four methods provided that allow you to control when and how the original values are stored, as shown in the following table:

Method	Description
`AcceptChanges`	Commits all the changes made to the tables or relations within the `DataSet` since it was loaded, or since the last time `AcceptChanges` was executed.
`GetChanges`	Returns a `DataSet` containing some or all of the changes made since it was loaded, or since the last time `AcceptChanges` was executed.
`HasChanges`	Indicates if any changes have been made to the contents of the `DataSet` since it was loaded, or since the last time `AcceptChanges` was executed.
`RejectChanges`	Abandons all the changes made to values in the tables within the `DataSet` since it was loaded, or since the last time `AcceptChanges` was executed. Returns it to the original state and removes all stored changes information.

The DataTable Class

Each of the tables or rowsets stored within a `DataSet` class is exposed through a `DataTable` class instance, as was shown in Figure 8-5. Each `DataTable` has a `Rows` property that references a `DataRowCollection` class instance. This is a collection of `DataRow` class instances.

The Commonly Used Methods of the DataTable Class

The `DataTable` class exposes a series of properties and methods that allow you to interact with each table individually while it is stored in the `DataSet`. The most commonly used methods are `Clear`, `AcceptChanges`, and `RejectChanges`. These are fundamentally the same as the methods just described for the `DataSet` class, but operate only on the specific table to which the `DataTable` class refers.

The following methods allow you to manipulate the contents of the table:

Method	Description
`NewRow`	Creates a new row for the table. The values can then be inserted into it using code, and the new row added to the table.
`Select`	Returns the set of rows that match a filter, in the order specified. Used to create subsets of rows.

The Commonly Used Methods of the DataRowCollection Class

This is a collection of all the rows in a `DataTable`, as referenced by the `Rows` property of the table. It provides methods to add and remove rows, and to find a row based on a value for the primary key (or more than one value for a multiple-column primary key). These methods are summarized in the following table:

Method	Description
`Add`	Adds a new row created with the `NewRow` method of the `DataTable` to the table
`Remove`	Permanently removes the specified `DataRow` class from the table
`RemoveAt`	Permanently removes a row specified by its index position from the table
`Find`	Takes an array of primary key values and returns the matching row as a `DataRow` instance

The Commonly Used Methods of the DataRow Class

This class represents the row itself within the table, and within the `DataRowCollection`. It has the `AcceptChanges` and `RejectChanges` methods, which work the same way as for the `DataTable` class.

The `DataRow` class also has methods that are used to manipulate individual rows in a table, as shown in the following table:

Methods	Description
`BeginEdit`, `EndEdit`, *and* `CancelEdit`	Used to switch the row into "edit mode" and save or abandon the changes made in this mode.
`Delete`	Marks the row as being deleted, though it is not removed from the table until the `Update` or `AcceptChanges` method is executed.
`GetChildRows`	Returns a collection of rows from another table that is related to this row as child rows.
`SetColumnError` *and* `GetColumnsInError`	Used to set and return the error status for this row. In conjunction with the `HasErrors` and `RowError` properties, this allows bulk edit errors to be reported separately afterwards.

The DataView Class

As shown in the earlier schematic, you can retrieve a `DataView` containing the data from a table within a `DataSet`. The `DataView` class exposes a complete table or a subset of the rows from a table. It can be created using the `DefaultView` of the table, or from a `DataTable` instance that selects a subset of rows from a table.

The Commonly Used Methods of the DataView Class

In general, to manipulate the contents of a table within a `DataSet`, it's best to create a `DataView` from the table and use the methods it provides. The most commonly used methods are shown in the following table:

Method	Description
`AddNew`	Adds a new row to the `DataView`. The values can then be inserted into it using code.
`Delete`	Removes the current or specified row from the `DataView`.
`Find`	Takes a single value or an array of values, and returns the index of the row that matches these value(s).
`FindRows`	Takes a single value or an array of values, and returns a collection of `DataRow` instances that match these value(s).

The DataReader Classes

While the `DataSet` provides a comprehensive platform for disconnected data access, there are many occasions when you just want a fast and efficient way to access a data store without actually extracting data that will be *remoted* (disconnected). This might be to extract one or a few records or specific field values, or to execute a simple `INSERT`, `UPDATE`, or `DELETE` SQL statement. Or, it might be where there is too much data to fit into a `DataSet` and to remote sensibly. It's also the ideal solution for server-side data binding in most cases, as mentioned in the previous chapter. For all these tasks you can use a `DataReader` class.

- The `OleDbDataReader` class is used with an OLEDB provider.
- The `SqlDataReader` class uses Tabular Data Services with MS SQL Server.
- The `OdbcDataReader` class is used with an ODBC driver.
- The `OracleDataReader` class is used to access an Oracle database.

As Figure 8-6 suggests, the `DataReader` provides the equivalent of a *firehose* cursor for direct connected access and retrieval of data from a data store. It's somewhat like the way an ADO `Recordset` is used to extract data and then iterate through it.

We execute a SQL statement or stored procedure to get a set of data rows that are referenced by a `DataReader`, and then iterate through them – while all the time remaining connected to the data store.

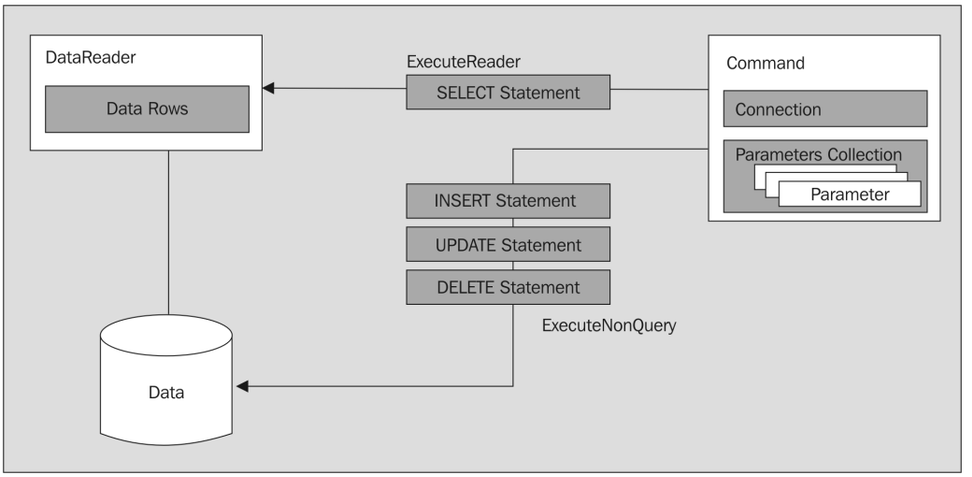

Figure 8-6

The important points to bear in mind with the `DataReader` are:

- It provides a partial equivalent of a cursor against a data store, using a SQL statement or stored procedure to extract a rowset.
- It provides the ability to execute a SQL statement or stored procedure to update the data store content.
- It does not provide disconnected access to data.
- Access to the rowset referenced by a `DataReader` is read-only and forward-only.

You can extract XML formatted data fragments directly from MS SQL Server 2000 using a reader instance (in this case an `XmlReader`) together with the in-built SQL-XML technology.

The Commonly Used Methods and Properties of the DataReader Classes

To use a `DataReader` class, create a `Command` class and then use this to execute your SQL statement or stored procedure and return a `DataReader`. You can then iterate through the rows and columns, using the `DataReader` to extract the results from the data store.

The following table shows the most commonly used methods exposed by the `DataReader` classes:

Method	Description
`Read`	Advances the current row pointer to the next row so that the values of the columns can be accessed using the column name or ordinal position. Returns `False` when there are no more rows to read.
`GetValue`	Returns one value from the current row in its native format (as the native data type in the data source) by specifying the integer column index. The simpler but less efficient alternative to using the column index is to specify the column name directly as: *value* = `DataReader("`*column-name*`")`.
`GetValues`	Gets one or more values from the current row in their native format (as the native data type in the data source) into an array.
`Getxxxxxx`	Returns a value from the current row as the data type specific to each method, by specifying the integer column index. Examples are `GetBoolean`, `GetInt16`, and `GetChars`.
`NextResult`	Moves the current row pointer to the next set of results when the statement is a SQL stored procedure or a batch SQL statement that returns more than one result set. Note that this is not a `MoveNext` operation like that of an ADO `Recordset` – it moves the current row pointer from one *rowset* to the first row in the *next rowset*.
`Close`	Closes the `DataReader` and releases the reference to the rowset.

The `DataReader` classes also expose some useful properties that allow you to discover details about the rowset that it is referencing, as shown in the following table:

Method	Description
`FieldCount`	Returns the number of columns (fields) in the rowset returned by the query or stored procedure.
`HasRows`	Returns a `Boolean` value of `True` if the execution of the query or stored procedure returned any rows, and `False` if there are no rows in the resulting rowset. This method was added in version 1.1.
`IsClosed`	Returns a `Boolean` value that is `True` if the `DataReader` has been closed, or `False` if it is still open following execution of the query or stored procedure.

Method	Description
`RecordsAffected`	Returns an Integer value that is the number of rows in the result set referenced by the `DataReader`. Only valid after all the rows have been read from the `DataReader` by a server control such as a `DataGrid`, or after iterating through until the `Read` method returns `False`.

Should I Use a DataReader or a DataSet?

When you start building applications that access a data store, think about what kind of access you actually need, and how the data will be used. It should be obvious from the descriptions of the classes that the `DataSet` carries a noticeable overhead in terms of complexity when compared to a `DataReader`, with the corresponding negative effect on performance and memory usage.

So, wherever it's possible, aim to use a `DataReader` rather than a `DataSet`. The kinds of occasions that require a `DataSet` are:

- When you need to remote the data (disconnect from the data store and pass the data to another tier in the application) to a client application, store it ready for use in a process, edit the data, or in some similar scenario.
- When you need to store, transport, or access more than one table (more than one `DataTable` instance), and optionally the relationships between these tables.
- When you need to update data in the source database using the in-built methods of the `DataSet` and `DataAdapter` rather than executing individual SQL `UPDATE` statements or stored procedures. The `DataSet` also stores the original (as well as the current) values of each column in each row, so it better manages a situation where multiple users are concurrently updating the data.
- When you need to take advantage of the synchronization between an XML document and the equivalent "relational" rowset. This topic is discussed in Chapter 11.
- In certain data binding scenarios, such as binding the same data to several controls or using automatic record paging in a `DataGrid` control, you cannot use a `DataReader` as the data source. In such cases, it's usual to use a `DataView` created from a table in a `DataSet`.
- If you are iterating through the data rows, and need the freedom to be able to move backwards and forwards in the rowset. You can't use a `DataReader` for this, as it is a forward-only data source.

Relational Data Providers for .NET

.NET uses managed code data providers to connect to a data store. The following table shows the .NET Data Providers that ship with version 1.1 of the .NET Framework:

Provider Name	Description
SQLOLEDB	OLEDB provider SQL Server
MSDAORA	OLEDB provider for Oracle
Microsoft.Jet.OLEDB.4.0	OLEDB provider for Access and other Jet data sources
SQL Server	ODBC driver for SQL Server
Microsoft ODBC for Oracle	ODBC driver for Oracle
Microsoft Access Driver (`*.mdb`)	ODBC driver for Microsoft Access
Oracle	Microsoft provider for Oracle (requires the Oracle client software version 8.1.7 or later to be installed)

Only the first three of the providers listed were included with.NET Framework version 1.0. A managed provider for ODBC was developed as a beta product during the version 1.0 timeframe, and can still be obtained from the *Microsoft Data* web site at http://www.microsoft.com/data/. More managed providers are planned, such as those for Microsoft Exchange, Active Directory, and other data stores. The existing unmanaged OLEDB providers for these data stores cannot be used in .NET.

> *The beta version of the ODBC driver installs in a different namespace from the driver included in version 1.1 of the .NET Framework. The current namespace is* `System.Data.Odbc`*, whereas the beta version was installed as* `Microsoft.Data.Odbc`*.*

Common Data Access Tasks with .NET

To demonstrate the basics of working with relational data in .NET, we've put together a series of sample pages that show the various objects in action. Figure 8-7 shows the `default.htm` main menu page for the samples:

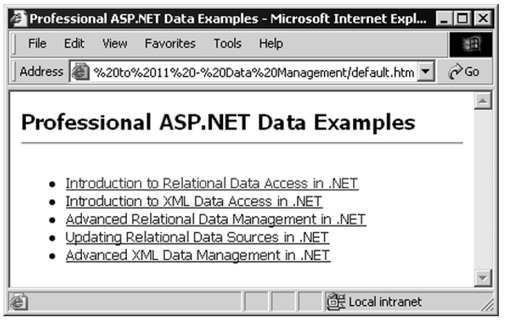

Figure 8-7

You can download the samples to run on your own server at http://www.daveandal.net/books/8900/. You can also run many of them online at the same URL. The samples are available in both VB and C#, and you can choose which to install – or install both sets.

The examples for this chapter are in the Introduction to Relational Data Access in .NET section, and this link displays the `default.htm` page for these sample pages, as shown in Figure 8-8.

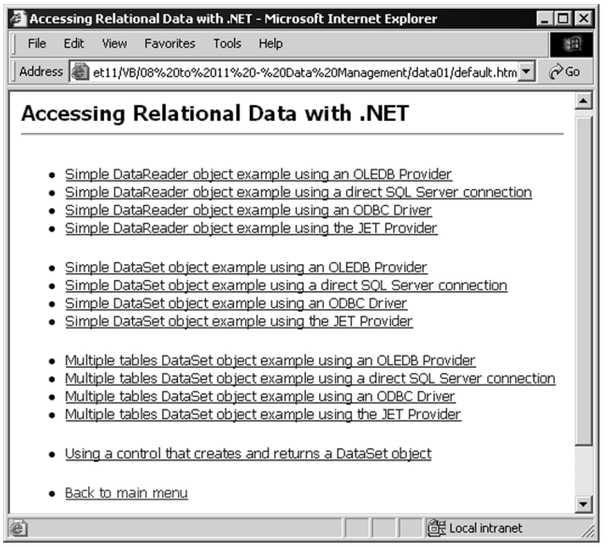

Figure 8-8

The first three groups of links show the three basic techniques for accessing relational data. Each group demonstrates four different connection types: an OLEDB provider for SQL Server, a direct SQL Server TDS connection, a connection through the .NET ODBC driver, and a connection to an Access database file through the Jet provider for Microsoft Access. There is also an example of using a user control that returns a `DataSet`. We'll be examining all these sample pages.

Setting Up the Samples on Your System

The downloadable sample files contain a `WroxBooks.mdb` Access database, which you can use with the Jet examples, and a set of SQL scripts that you can use to create the sample `WroxBooks` database on your own local SQL Server. Instructions for using the scripts are in the `readme.txt` file located within the `database` folder of the samples.

You'll also need to edit the connection strings in the `web.config` file that is installed in the root folder of the examples to suit your setup. The `<appSettings>` section of the `web.config` file contains declarations of the connection strings for all of the examples for this book, but the ones that are relevant to this chapter are highlighted in the following code. Notice that there are four, one for each of the providers/drivers used in the example pages:

```
<configuration>
    ... other settings here ...
  <appSettings>
    <add key="DsnWroxBooksSql"
         value="server=delmonte; database=WroxBooks; user id=sa; password=" />
    <add key="DsnWroxBooksOleDb"
         value="provider=SQLOLEDB.1; data source=delmonte;
                initial catalog=WroxBooks; uid=sa; pw=" />
    <add key="DsnWroxBooksJet"
         value="Provider=Microsoft.Jet.OLEDB.4.0;Data Source=" />
    <add key="DsnWroxBooksOdbc"
         value="DRIVER={SQL Server}; SERVER=delmonte;
                DATABASE=WroxBooks; uid=sa; pw=;" />
    ... other settings here ...
  </appSettings>
</configuration>
```

Any page within the samples can access and use these connection strings by using:

```
strSQLConnect = ConfigurationSettings.AppSettings("DsnWroxBooksSql")
strOLEDBConnect = ConfigurationSettings.AppSettings("DsnWroxBooksOleDb")
strJetConnect = ConfigurationSettings.AppSettings("DsnWroxBooksJet")
strOdbcConnect = ConfigurationSettings.AppSettings("DsnWroxBooksOdbc")
```

Setting Up the Required File Access Permissions

Some of the examples files require write access to the server's `wwwroot` folder and subfolders below this. By default in Windows NT, Windows 2000, and Windows XP, ASP.NET runs under the context of the ASPNET account that is created by the installation and setup of the .NET Framework. This is a relatively unprivileged account that has similar permissions by default as the IUSR_machinename account that is used by Internet Information Services.

To give folders on your test server write access for ASP.NET, right-click on the `wwwroot` folder in Windows Explorer and open the Properties dialog. In the Security tab, select the ASPNET account and give it Write permission or Full Control. Then click Advanced and tick the checkbox at the bottom of this page (Reset permissions on all child objects…).

Alternatively, configure ASP.NET to run under the context of the local System account by editing the `machine.config` file located in the `config` directory of the installation root. By default, this is the `C:\WINNT\Microsoft.NET\Framework\[version]\CONFIG\` directory. Change just the `userName` attribute in the `<processModel>` element within the `<system.web>` section of this file to:

```
<processModel userName="system" password="autogenerate" ... />
```

Do this only while experimenting and only on a development server. For production servers, set up only the minimal permissions required for your applications to run.

ASP.NET with IIS 6.0 and Windows Server 2003

While all the this is true for IIS 4.0 and IIS 5.0, as installed with Windows NT, Windows 2000, and Windows XP, the new version of IIS supplied with Windows Server 2003 (IIS 6.0) works in a slightly different way. Security and account permissions are discussed in Chapter 14. However, to enable the example pages to run on Windows Server 2003 you only need to know the basics here.

By default, in Windows Server 2003, web sites run within Application Pools and the worker processes used for accessing resources run under the context of an account named NETWORK SERVICE. Windows Server 2003 creates an account group called IIS_WPG, of which the IWAM_machinename, LOCAL SERVICE, NETWORK SERVICE and SYSTEM accounts are automatically members.

It means that you can use this group to configure access to resources for ASP.NET running under the default IIS 6.0 configuration. Alternatively, you can just assign the necessary Write permission directly to the NETWORK SERVICE account if you prefer more fine-grained control.

You can also configure IIS 6.0 to run in a special *compatibility* mode called *IIS 5.0 Isolation Mode* (in the Service tab of the Properties dialog for the Web Sites entry in the IIS Manager). In this case, IIS 6.0 runs ASP.NET just like it does under IIS 5.0, and the accounts used and permission settings you make are the same as in IIS 5.0.

So, the ASPNET account is used for ASP.NET resources, and the IWAM_machinename account is used for other resources. And an account named IWAM_machinename is used for out-of-process execution of components in this mode, just as in IIS 5.0.

> *For more information of the IIS and ASP.NET security model in Windows Server 2003, open the Help file from IIS Manager and navigate to the Server Administration Guide | Security | Access Control | Web Site Permissions section.*

Using a DataReader Object

The first group of links in the relational data access menu shows the `DataReader` in action. This is the nearest equivalent to the `Connection/Recordset` data access technique used in traditional ADO. Figure 8-9 shows the result of running the OLEDB example. The others from the same group (the SQL TDS, ODBC and Jet provider examples) provide identical output, but with different connection strings.

The code in the page (`datareader-oledb.aspx`) is placed within the `Page_Load` event handler. So, it runs when the page loads. The code inserts the connection string, SQL `SELECT` statement, and the results into `<div>` elements within page. All the code is fully commented, and we've included elementary error handling to display any errors. However, only the relevant data access code has been shown here. You can examine the entire source code for any of the pages by clicking the [view source] link at the bottom.

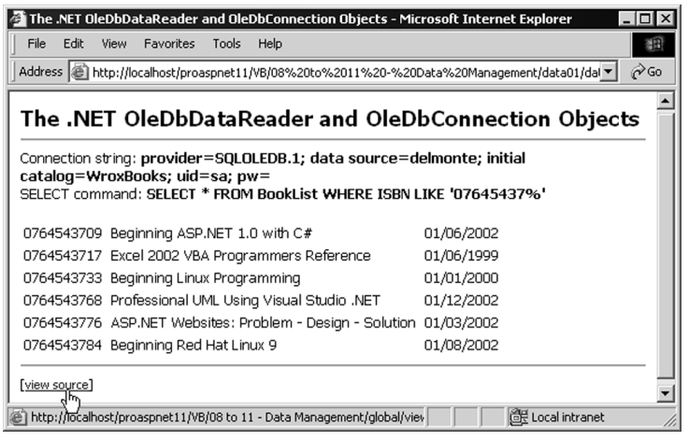

Figure 8-9

The DataReader Example Code

The following code has been used in this example:

```
'get connection string from web.config file and display it
strConnect = ConfigurationSettings.AppSettings("DsnWroxBooksOleDb")
outConnect.InnerText = strConnect

'specify the SELECT statement to extract the data and display it
strSelect = "SELECT * FROM BookList WHERE ISBN LIKE '07645437%'"
outSelect.InnerText = strSelect

'create a new Connection object using the connection string
Dim objConnect As New OleDbConnection(strConnect)

'open the connection to the database
objConnect.Open()

'create a new Command using the connection object and select statement
Dim objCommand As New OleDbCommand(strSelect, objConnect)

'declare a variable to hold a DataReader object
Dim objDataReader As OleDbDataReader

'execute the SQL statement against the command to fill the DataReader
objDataReader = objCommand.ExecuteReader()
```

The first step is to get the connection string from the `web.config` file, and then specify the SQL statement. These are displayed as the code runs in `<div>` elements named `outConnect` and `outSelect` (located within the HTML of the page). Then, we create a new instance of an `OleDbConnection` object, specifying the connection string as the single parameter of its constructor.

After opening the connection by calling the `Open` method, you need an `OleDbCommand` object. This will be used to execute the statement and return a new `OleDbDataReader` object through which you can access the results of the query. The SQL statement and the active `Connection` object are specified as the parameters to the `OleDbCommand` object constructor: You can then call the `ExecuteReader` method of the `OleDbCommand` object. This returns an `OleDbDataReader` object that is connected to the result rowset.

Displaying the Results

A `DataReader` allows you to iterate through the results of a SQL query, much like you do with a traditional ADO `Recordset` object. However, unlike in the ADO `Recordset`, in a `DataReader` you must call the `Read` method first to be able to access the first row of the results. Afterwards, just call the `Read` method repeatedly to get the next row of the results until it returns `False` (which indicates that the end of the results set has been reached).

> *We no longer have a `MoveNext` method. Forgetting to include this statement was found by testers to be the most common reason for problems when working with the `Recordset` object in ADO.*

As was common practice in ASP 3.0 and earlier, you can build up an HTML `<table>` to display the data. However, as you're working with ASP.NET now, this example actually creates the definition of the table as a string and then inserts it into a `<div>` element elsewhere in the page (rather than the ASP-style technique of using `Response.Write` directly). The following code was used to create the output shown in Figure 8-9:

```
Dim strResult As String = "<table>"

'iterate through the records in the DataReader getting field values
'the Read method returns False when there are no more records
Do While objDataReader.Read()
   strResult += "<tr><td>" & objDataReader("ISBN") & "</td><td>  " _
             & objDataReader("Title") & "</td><td>  " _
             & objDataReader("PublicationDate") & "</td><td></tr>"
Loop

'close the DataReader and Connection
objDataReader.Close()
objConnect.Close()

'add closing table tag and display the results
strResult += "</table>"
outResult.InnerHtml = strResult
```

> *You could, of course, simply declare an ASP.NET list control such as a `DataGrid` in the page, and then bind the `DataReader` to the control to display the results. However, the technique used here to display the data demonstrates how we can iterate through the rowset.*

Closing the DataReader and the Connection

You have to explicitly close the `DataReader`. You also have to explicitly close the connection by calling the `Connection` object's `Close` method. Although the garbage collection process will close the `DataReader` when it destroys the object in memory after the page ends, it's good practice to always close *reader* objects connections as soon as you are finished with them.

It's even more important to close the connection after you finish with it. Database connections are a precious resource, and the number available is usually limited. For this reason, as you'll see in the next section, ADO.NET provides a useful method that will close a connection automatically.

The CommandBehavior Enumeration

One useful technique to bear in mind when using a `DataReader` is to take advantage of the optional parameter for the `Command` object's `ExecuteReader` method. It can be used to force the connection to be closed automatically as soon as we call the `Close` method of the `DataReader` object:

```
objDataReader = objCommand.ExecuteReader(CommandBehavior.CloseConnection)
```

This is particularly useful if you pass a reference to the `DataReader` to another routine, say if you return it from a method. By using the `CommandBehavior.CloseConnection` option, you can be sure that the connection will be closed automatically when the routine using the `DataReader` destroys the object reference.

Other values in the `CommandBehavior` enumeration that you can use with the `ExecuteReader` method (multiple values can be used with `And` or `+`) are:

- `SchemaOnly`: The execution of the query will only return the schema (column information) for the results set, and not any data. It can be used, for example, to find the number of columns in the results set.
- `SequentialAccess`: Can be used to allow the `DataReader` to access large volumes of binary data from a column. The data is accessed as a stream rather than as individual rows and columns, and is retrieved using the `GetBytes` or `GetChars` methods of the `DataReader`.
- `SingleResult`: Useful if the query is only expected to return a single value, and can help the database to fine-tune the query execution for maximum efficiency. Alternatively, use the `ExecuteScalar` method of the `Command` object.
- `SingleRow`: Useful if the query is only expected to return one row, and can help the database to fine-tune the query execution for maximum efficiency.

Overall, the techniques used in this example are not that far removed from working with traditional ADO in ASP. However, there are far more opportunities available in .NET for accessing and using relational data. These revolve around the `DataSet` rather than the `DataReader`.

A Simple DataSet Example

A `DataSet` is a disconnected read/write container for holding one or more tables of data, and the relationships between these tables. In this example, we just extract a single table from the database and display the contents.

Figure 8-10 shows what the Simple DataSet object example using an OLEDB Provider (`simple-dataset-oledb.aspx`) sample looks like when it's run:

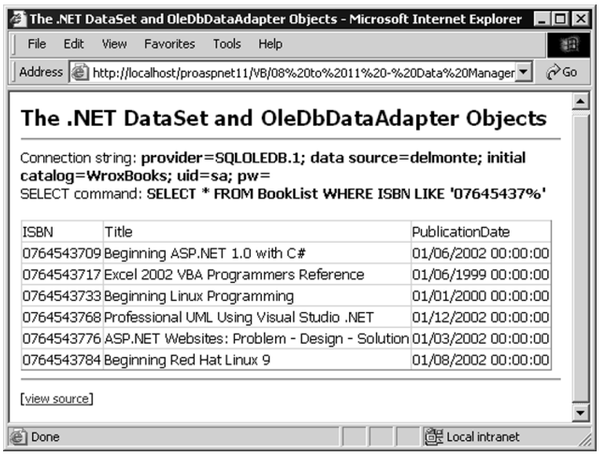

Figure 8-10

The Simple DataSet Example Code

We've used the same connection string and SQL statement as in the `DataReader` example. We also create a new `OleDbConnection` object using this connection string as before:

```
Dim objConnect As New OleDbConnection(strConnect)
```

To execute the SQL statement for the `OleDbDataReader` object in the previous example, we used the `ExecuteReader` method of the `OleDbCommand` object. In this example, to fill a `DataSet` object with data, we use an alternative object to specify the SQL statement – an `OleDbDataAdapter` object. Again, we provide the SQL statement and the active `Connection` object as the parameters to the object constructor:

```
Dim objDataAdapter As New OleDbDataAdapter(strSelect, objConnect)
```

> *This technique still creates and uses a `Command` object. When you create a `DataAdapter`, a suitable `Command` instance is created automatically behind the scenes, and assigned to the `SelectCommand` property of your `DataAdapter`. You could do this yourself, but it would mean writing the extra code, and there is no advantage in doing so.*

Now create an instance of a `DataSet` object and then fill it with data from the data source by calling the `Fill` method of the `DataAdapter` object. Specify as parameters the `DataSet` object and the name of the source table in the database:

```
Dim objDataSet As New DataSet()
objDataAdapter.Fill(objDataSet, "Books")
```

Filling the Schema in a DataSet

The `Fill` method of the `DataAdapter` object that was used here creates the table in the `DataSet`, and then creates the appropriate columns and sets the data type and certain constraints such as the column

width (the number of characters). It doesn't automatically set the primary keys, unique constraints, read-only values, and defaults. However, you can call the `FillSchema` method first (before you call `Fill`) to copy these settings from the data source into the table:

```
objDataAdapter.FillSchema(objDataSet, SchemaType.Mapped)
```

After all this, you've now got a disconnected `DataSet` object that contains the results of the SQL query. The next step is to display that data.

Displaying the Results

In this and many of the other examples, we're using an ASP `DataGrid` control to display the data in the `DataSet` object. You saw how the `DataGrid` control works in Chapter 7:

```
<asp:datagrid id="dgrResult" runat="server" />
```

However, you can't simply bind the `DataSet` object directly to a `DataGrid` and have the correct rows displayed, as a `DataSet` can contain multiple tables. One solution is to create a `DataView` based on the table you want to display, and bind the `DataView` object to the `DataGrid`. You get the default `DataView` object for a table by accessing the `Tables` collection of the `DataSet` and specifying the table name:

```
Dim objDataView As New DataView(objDataSet.Tables("Books"))
```

Then, assign the `DataView` to the `DataSource` property of the `DataGrid`, and call the `DataBind` method to display the data:

```
dgrResult.DataSource = objDataView
dgrResult.DataBind()
```

However, it's actually better performance-wise, though not as clear when you read the code, to perform the complete property assignment in one statement:

```
dgrResult.DataSource = objDataSet.Tables("Books").DefaultView
```

There is also a third option, as the ASP.NET Server Controls provide a `DataMember` property that defines which table or other item in the data source will supply the data. So you could use:

```
dgrResult.DataSource = objDataSet
dgrResult.DataMember = "Books"
```

We use a mixture of techniques in our examples.

A Multiple Tables DataSet Example

Having seen how to use a `DataSet` to hold one *results* table, you'll now see how to add multiple tables to a `DataSet` object. The Multiple tables DataSet object example using an OLEDB Provider (`multiple-dataset-oledb.aspx`) example creates a `DataSet` object and fills it with three tables. It also creates relationships between these tables.

As you can see in Figure 8-11, the page displays the connection string and the three SQL statements that extract the data from three tables in the database. Following this are two `DataGrid` controls showing the contents of the `DataSet` object's `Tables` collection and `Relations` collection. Further down the page (not visible here) are two more `DataGrid` controls, which show the related data that is contained in the `Authors` and `Prices` tables within the `DataSet`.

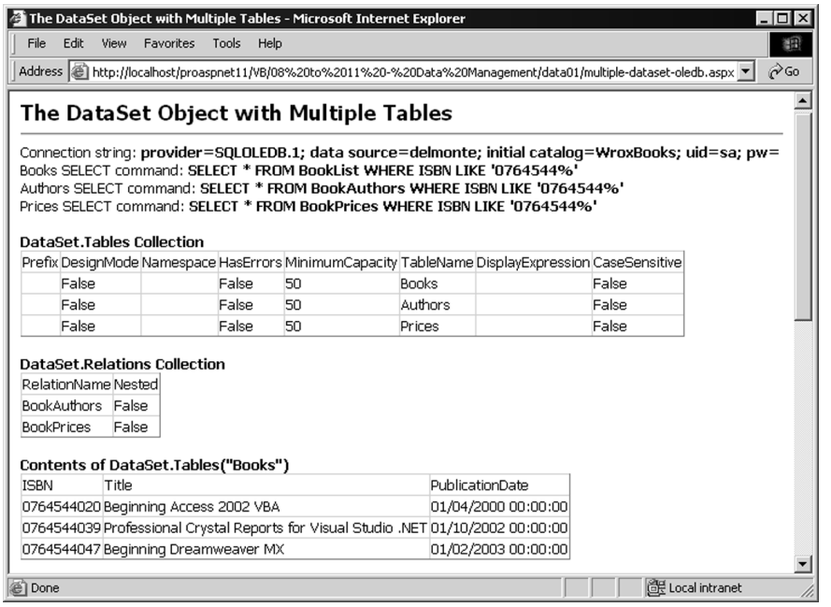

Figure 8-11

The Multiple Tables DataSet Example Code

While the principle for this example is similar to the previous Simple DataSet example, the way we've coded it is subtly different. We've demonstrated another way of using the `Command` and `DataAdapter` objects.

As before, first create a `Connection` object using your connection string, shown in the following code. However, this time create a `Command` object next using the default constructor with no parameters, and then set the properties of the `Command` object in a way similar to that used in *traditional* ADO.

Specify the connection string, the command type (in this case `Text`, as a SQL statement is being used), and the SQL statement itself for the `CommandText` property. By doing it this way, you can change the SQL statement later to get a different set of rows from the database without having to create a new `Command` object.

```
'create a new Connection object using the connection string
Dim objConnect As New OleDbConnection(strConnect)
'create a new Command object
Dim objCommand As New OleDbCommand()
```

```
'set the properties
objCommand.Connection = objConnect
objCommand.CommandType = CommandType.Text
objCommand.CommandText = strSelectBooks
```

Once you have a `Command` object, you can use it within a `DataAdapter`. You need a `DataAdapter` to extract the data from the database and squirt it into your `DataSet` object. After creating the `DataAdapter`, assign the `Command` object to its `SelectCommand` property. This `Command` will then be used when you call the `Fill` method to get the data:

So, you've got a valid `DataAdapter` object, and you can set about filling your `DataSet`. Call the `Fill` method three times, once for each table you want to insert into it. In between, you have to change the `CommandText` property of the active `Command` object to the appropriate SQL statement, as shown in the following code:

```
'create a new DataAdapter object
Dim objDataAdapter As New OleDbDataAdapter()
'and assign the Command object to it
objDataAdapter.SelectCommand = objCommand

'get the data from the "BookList" table in the database and
'put it into a table named "Books" in the DataSet object
objDataAdapter.Fill(objDataSet, "Books")

'change the SELECT statement in the Command object
objCommand.CommandText = strSelectAuthors
'then get data from "BookAuthors" table into the DataSet
objDataAdapter.Fill(objDataSet, "Authors")

'and do the same again to get the "BookPrices" data
objCommand.CommandText = strSelectPrices
objDataAdapter.Fill(objDataSet, "Prices")
```

Opening and Closing Connections with the DataAdapter

In the examples that use a `DataAdapter`, we haven't explicitly opened or closed the connection. This is because the `DataAdapter` looks after this automatically. If the connection is *closed* when the `Fill` method is called, it is opened, the rows are extracted from the data source and pushed into the `DataSet`, and the connection is automatically closed again.

However, if the connection is *open* when the `Fill` method is called, the `DataAdapter` will leave it open after the method has completed. This provides you with a useful opportunity to maximize performance by preventing the connection being opened and closed each time you call `Fill` (if you are loading more than one table in the `DataSet`). Just open the connection explicitly before the first call, and close it again after the last one, as shown by the highlighted lines in the following code:

```
Dim objDataSet As New DataSet()
objCommand.CommandText = strSelectBooks
objConnect.Open()
objDataAdapter.Fill(objDataSet, "Books")
objCommand.CommandText = strSelectAuthors
objDataAdapter.Fill(objDataSet, "Authors")
```

```
objCommand.CommandText = strSelectPrices
objDataAdapter.Fill(objDataSet, "Prices")
objConnect.Close()
```

Adding Relationships to the DataSet

You've got three tables in your `DataSet`, and can now create the relationships between them. Define a variable to hold a `DataRelation` object and create a new `DataRelation` by specifying the name you want for the relation (`BookAuthors`), the name of the primary key field (`ISBN`) in the parent table named `Books`, and the name of the foreign key field (`ISBN`) in the `Authors` child table.

Then add the new relation to the `DataSet` object's `Relations` collection, and do the same to create the relation between the `Books` and `Prices` tables in the `DataSet`. As the relations are added to the `DataSet`, an integrity check is carried out automatically. If, for example, there is a child record that has no matching parent record, an error is raised and the relation is not added to the `DataSet`.

```
'declare a variable to hold a DataRelation object
Dim objRelation As DataRelation

'create a Relation object to link Books and Authors
objRelation = New DataRelation("BookAuthors", _
                  objDataSet.Tables("Books").Columns("ISBN"), _
                  objDataSet.Tables("Authors").Columns("ISBN"))
'and add it to the DataSet object's Relations collection
objDataSet.Relations.Add(objRelation)

'now do the same to link Books and Prices
objRelation = New DataRelation("BookPrices", _
                  objDataSet.Tables("Books").Columns("ISBN"), _
                  objDataSet.Tables("Prices").Columns("ISBN"))
objDataSet.Relations.Add(objRelation)
```

Displaying the Results

Having filled the `DataSet` with three tables and two relations, you can now display the results. You use five `DataGrid` controls to do this, as shown in the following code listing. The `DataSet` object's `Tables` and `Relations` collections are bound directly to the first two `DataGrid` controls, and for the tables within the `DataSet`, we assign the `DataView` returned by the `DefaultView` property of the tables to the remaining three `DataGrid` controls.

```
'bind the collection of Tables to the first DataGrid on the page
dgrTables.DataSource = objDataSet.Tables
dgrTables.DataBind()

'bind the collection of Relations to the second DataGrid on the page
dgrRelations.DataSource = objDataSet.Relations
dgrRelations.DataBind()

'create a DataView object to use with the tables in the DataSet
Dim objDataView As New DataView()

'get the default view of the Books table into the DataView object
objDataView = objDataSet.Tables("Books").DefaultView
```

```
    'and bind it to the third DataGrid on the page
dgrBooksData.DataSource = objDataView
dgrBooksData.DataBind()
    'then do the same for the Authors table
objDataView = objDataSet.Tables("Authors").DefaultView
dgrAuthorsData.DataSource = objDataView
dgrAuthorsData.DataBind()
    'and finally do the same for the Prices table
objDataView = objDataSet.Tables("Prices").DefaultView
dgrPricesData.DataSource = objDataView
dgrPricesData.DataBind()
```

A User Control That Returns a DataSet Object

The preceding code is used in several examples in this and subsequent chapters, and to make it easier we've encapsulated it as a user control that returns a fully populated `DataSet`. Change the page's file extension to `.ascx` and change the `Page` directive to a `Control` directive:

```
<%@Control Language="VB"%>
```

Then, instead of placing the code in the `Page_Load` event handler, place it in a `Public Function` to which you provide the connection string and the `WHERE` clause for the SQL statement as parameters. The function returns a `DataSet` object, as shown in the following code. Note that the parameters passed to this function allow you to select a different set of books by varying the `strWhere` parameter value when you use the control.

```
Public Function BooksDataSet(strConnect As String, _
                             strWhere As String) _
                             As DataSet
   ...
   strSelectBooks = "SELECT * FROM BookList WHERE " & strWhere
   strSelectAuthors = "SELECT * FROM BookAuthors WHERE " & strWhere
   strSelectPrices = "SELECT * FROM BookPrices WHERE " & strWhere
   Dim objDataSet As New DataSet()
   ...
   ... code to fill DataSet as before ...
   ...
   Return objDataSet
End Function
```

The Using a control that creates and returns a DataSet object (`use-dataset-control.aspx`) example page contains the `Register` directive and matching element to insert the user control containing the function just described into the page. Then, to get a `DataSet` from the control, just create a variable of the correct type and set it to the result of the `BooksDataSet` method – specifying the values for the connection string and `WHERE` clause parameters when you make the call.

```
<%@ Register TagPrefix="wrox" TagName="getdataset"
             Src="..\global\get-dataset-control.ascx" %>
...
<wrox:getdataset id="ctlDataSet" runat="server"/>
Dim objDataSet As DataSet
objDataSet = ctlDataSet.BooksDataSet(strConnect, "ISBN LIKE '0764544%'")
```

The investigation of the `DataSet` object will be continued in Chapters 9 and 10. You'll see how to use more complex data sets, and update and edit data using the ADO.NET relational data access classes. We'll also explore the ways that .NET combines the traditional relational database access techniques with the more recent developments in XML-based data storage and management.

An Introduction to XML in .NET

The previous section described the features of .NET that are aimed at accessing relational data, and how they relate to the way you work with data compared to the traditional techniques used in previous versions of ADO. However, there is another technique for working with data within the .NET Framework.

XML is fast becoming the lingua franca of the Web, and is being adopted within many other application areas as well. We discussed the reasons for this earlier, and now look at how XML is supported within .NET. This relates to the .NET support for relational data, as XML is the standard persistence format for data within the .NET data access classes. However, there are also several other techniques for reading, writing, and manipulating XML data and the associated XML-based data formats.

In this book, we're assuming that the reader is familiar with XML as a data storage mechanism, and how it is used through an XML parser and with the associated technologies such as XSLT. Our aim is to show the way that the .NET Framework and ASP.NET can be used with XML data.

> *For a primer and other reference materials covering XML and the associated standards and technologies, check out the Wrox Press list of XML books at http://www.wrox.com/.*

The Fundamental XML Objects

The W3C (at http://www.w3.org/) provides standards that define the structure and interfaces that should be provided by applications used for accessing XML documents. This is referred to as the XML *Document Object Model* (*DOM*), and is supported under .NET by the `XmlDocument` and `XmlDataDocument` objects, as shown in Figure 8-12. They provide full support for the XML DOM Level 2 Core. Within their implementation are the node types and objects that are required for the DOM interfaces, such as the `XmlElement` and `XmlAttribute` objects:

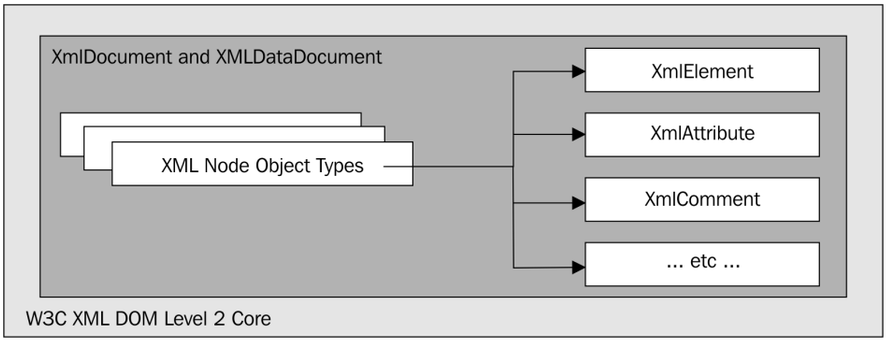

Figure 8-12

However, .NET extends the support for XML to provide much more in the way of techniques for manipulating XML documents, XML Schemas, and stylesheets. Figure 8-13 shows the main classes that are used when working with XML documents within .NET applications, and how they are related by showing the kinds of paths that you can follow when working with XML data:

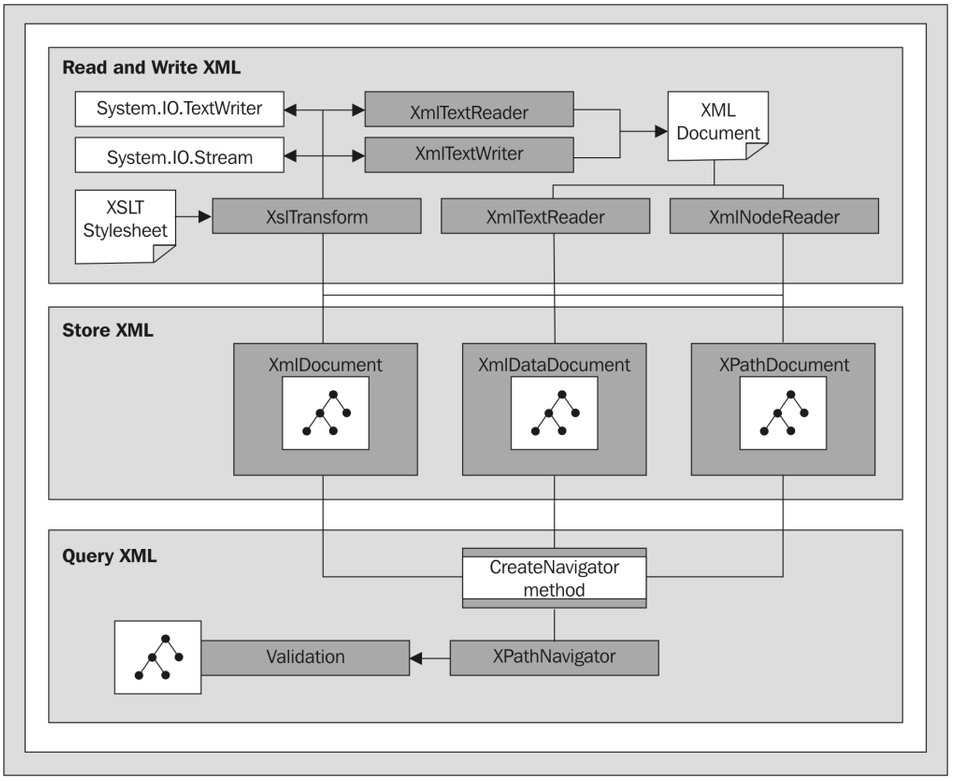

Figure 8-13

Basically, the classes shown in Figure 8-13 fall into three categories:

- **Reading, writing, and transforming XML**: The `XmlTextReader`, `XmlNodeReader`, and `XmlTextWriter` – plus the `XslTransform` object for creating files in a different format to the original XML document.
- **Storing and manipulating XML**: The function of the `XmlDocument`, `XmlDataDocument`, and `XPathDocument` objects.
- **Querying XML**: The function of `XPathNavigator` object.

There is some overlap between these functions, of course. To validate an XML document while reading it, you can use an `XmlValidatingReader`, and there are other objects for creating and editing XML

Schemas that aren't covered in this book. You can also use the `XslTransform` object to perform querying of a document as well as transforming it into different formats.

In this section, we'll briefly overview the objects and their commonly used methods, and then to show some simple examples. We'll come back to XML again in Chapter 11 and see some more advanced techniques.

The Document Objects

There are three implementations of the document object for storing and working with XML:

- The `XmlDocument` object is the .NET implementation of the standard DOM Level 2 `XMLDocument` interface. The properties and methods it exposes include those defined by W3C for manipulating XML documents, plus some extensions to make common operations easier.
- The `XmlDataDocument` object is an extension of the `XmlDocument` object, providing the same set of properties and methods. However, it also acts as a "bridge" between XML and relational data access methods. Once loaded with an XML document, it can expose it as a `DataSet` object. This allows you to use relational data programming techniques to work with the data, as well as the same XML DOM techniques that are used with an `XmlDocument` object.
- The `XPathDocument` object is a fast and compact implementation of an XML storage object that is designed for access via an `XPathNavigator` object, using only XPath queries or navigation element-by-element using the "pull" technique.

The Commonly Used XML Document Methods

The `XPathDocument` object has no really useful public methods other than `CreateNavigator`, as it is designed solely to work with an `XPathNavigator` object. However, the other two document objects expose the full set of properties and methods specified in the W3C XML DOM Level 2 Core. The extensions to these properties and methods include several very useful methods regularly used to work with XML documents.

The following table shows the extensions for creating specific types of node, and accessing existing nodes in the `XmlDocument` and `XmlDataDocument` objects:

Method	Description
`Createxxxxxx`	Creates a node in the XML document depending on the actual method name. Examples are `CreateElement`, `CreateComment`, and `CreateTextNode`.
`CloneNode`	Creates a duplicate of an XML node (for example, a copy of an element).
`GetElementById`	Returns the single node with the specified value for its `ID` attribute.
`GetElementsByTagname`	Returns a collection of nodes that contains all the elements with the specified element name.

The following table shows the series of useful methods that are available for loading and saving XML to and from the `XmlDocument` and `XmlDataDocument` objects:

Method	Description
`Load`	Loads an XML document from a disk file, a `Stream`, or an `XmlTextReader`
`LoadXml`	Loads an XML document from a `String`
`Save`	Saves the entire XML document to a disk file, a `Stream`, or an `XmlTextWriter`
`ReadNode`	Loads a node from an XML document that is referenced by an `XmlTextReader` or `XmlNodeReader`
`WriteTo`	Writes a node to another XML document that is referenced by an `XmlTextWriter`
`WriteContentTo`	Writes a node and all its descendents to another XML document that is referenced by an `XmlTextWriter`

To use an `XPathNavigator` with any of the three types of XML document object, create it using the `CreateNavigator` method as shown in the following table:

Method	Description
`CreateNavigator`	Creates and returns an `XPathNavigator` based on the currently loaded document. Applies to all three document objects. Optionally, for the `XmlDocument` and `XmlDataDocument` only, accepts a parameter that is a reference to a node within the document that will act as the start location for the navigator.

The `XmlDataDocument` adds a single property to those exposed by the `XmlDocument` class, as shown in the following table:

Property	Description
`DataSet`	Returns the contents of the XML document as an ADO.NET `DataSet`.

The `XmlDataDocument` also adds methods that provide extra access to the contents of the document, treating it more like a rowset or data table, as shown in the following table:

Method	Description
`GetRowFromElement`	Returns a `DataRow` representing the element in the document.
`GetElementFromRow`	Returns an `XmlElement` representing a `DataRow` in a table within a `DataSet`.

The XPathNavigator Class

In order to make working with XML documents easier, the `System.Xml` namespace classes include the `XPathNavigator`, which can be used to navigate within an XML document or to query the content of the document using an XPath expression. Note that an `XPathNavigator` can be used with *any* of the XML document objects – not just an `XPathDocument`. You can create an `XPathNavigator` based on an `XmlDocument` or an `XmlDataDocument` as well.

- The `XPathNavigator` provides methods and properties that allow cursor-style navigation through the XML document; for example, by stepping through the nodes (elements and attributes) in order, or by skipping to the next node of a specific type.
- The `XPathNavigator` provides methods that accept an XPath expression, the name of a node or a node type, and return one or more matching nodes. You can then iterate through these nodes.

An `XPathNavigator` can *only* be created from an existing document object. This is done using the `CreateNavigator` method:

```
Dim objNav1 As XPathNavigator = objXMLDoc.CreateNavigator()
Dim objNav2 As XPathNavigator = objXMLDataDoc.CreateNavigator()
Dim objNav3 As XPathNavigator = objXPathDoc.CreateNavigator()
```

The Commonly Used XPathNavigator Methods

The `XPathNavigator` is designed to act as a *pull* model interface for an XML document. It allows you to navigate across a document, and select and access nodes within that document. You can also create two (or more) navigator objects against the same document, and compare their positions.

To edit the XML document(s), you can use the reference to the current node exposed by the navigator, or an `XPathNodeIterator` that contains a collection of nodes, and call the methods of that node or collection. At the same time, the `XPathNavigator` exposes details about the current node directly, so there are two ways to get information about each node.

The table that follows shows methods used to move around within the document, making different nodes current in the navigator, and to create a new navigator:

Method	Description
`MoveToxxxxxx`	Moves the current navigator position. Examples are `MoveToFirst`, `MoveToFirstChild`, `MoveToParent`, `MoveToAttribute`, and `MoveToRoot`.
`Clone`	Creates a new `XPathNavigator` that is automatically located at the same position in the document as the current navigator.
`IsSamePosition`	Indicates if two navigators are at the same position within the document.

The following table shows the methods used to access and select parts of the content of the document:

Method	Description
`GetAttribute`	Returns the value of a specified attribute from the current node in the navigator
`Select`	Returns an `XPathNodeIterator` (a `NodeList`) containing a collection of nodes that match the specified `XPath` expression
`SelectAncestors`	Returns an `XPathNodeIterator` (a `NodeList`) containing a collection of all the ancestor nodes in the document of a specific type or which have a specific name
`SelectDescendants`	Returns an `XPathNodeIterator` (a `NodeList`) containing a collection of all the descendant nodes in the document of a specific type or which have a specific name
`SelectChildren`	Returns an `XPathNodeIterator` (a `NodeList`) containing a collection of all the child nodes in the document of a specific type or which have a specific name

The XmlTextWriter Class

When using an `XmlDocument` to create a new XML document, you must create document fragments and insert them into the document in a specific way – a technique that can be error-prone and complex. The `XmlTextWriter` can be used to create an XML document node by node in serial fashion by simply writing the tags and content to the output stream using the comprehensive range of methods that it provides.

- The `XmlTextWriter` takes as its source either a `TextWriter` that refers to a disk file, the path and name of a disk file, or a `Stream` that will contain the new XML document. It exposes a

series of properties and methods that can be used to create XML nodes and other content, and output them to the disk file or stream directly.

- The `XmlTextWriter` can also be specified as the output device for methods in several other objects, where it automatically streams the content to a disk file, a `TextWriter`, or a `Stream`.

The `TextReader`, `TextWriter`, and `Stream` classes are discussed in Chapter 16.

The Commonly Used XmlTextWriter Methods

The most commonly used methods of the `XmlTextWriter` are listed in the following table:

Method	Description
`WriteStartDocument`	Starts a new document by writing the XML declaration to the output.
`WriteEndDocument`	Ends the document by closing all un-closed elements, and flushing the content to disk.
`WriteStartElement`	Writes an opening tag for the specified element. The equivalent method for creating attributes is `WriteStartAttribute`.
`WriteEndElement`	Writes a closing tag for the current element. The equivalent method for completing an attribute is `WriteEndAttribute`.
`WriteElementString`	Writes a complete element (including opening and closing tags) with the specified string as the value. The equivalent method for writing a complete attribute is `WriteAttributeString`.
`Close`	Closes the stream or disk file and releases any references held.

The XmlReader Classes

You need to be able to read documents from other sources, rather than creating them from scratch. The `XmlReader` class is a base class from which two public classes, `XmlTextReader` and `XmlNodeReader`, inherit.

- The `XmlTextReader` takes as its source either a `TextReader` that refers to an XML disk file, the path and name of an XML disk file, or a `Stream` containing an XML document. The contents of the document can be read one node at a time, and the object provides information about each node and its value as it is read.
- The `XmlNodeReader` takes a reference to an `XmlNode` instance (usually from within an `XmlDocument`) as its source, allowing you to read specific portions of an XML document rather than having to read all of it, if you only want to access a specific node and its children.

- The `XmlTextReader` and `XmlNodeReader` can be used standalone to provide simple and efficient access to XML documents or as the source for another object whereby they automatically read the document and pass it to the parent object.

Like the `XPathNavigator`, the `XmlTextReader` provides a *pull* model for accessing XML documents node-by-node, rather than parsing them into a tree in memory as is done in an XML parser. This allows larger documents to be accessed without impacting on memory usage, and can also make coding easier, depending on the task you need to accomplish.

Furthermore, if you are just searching for a specific value, you won't always have to read the whole document. Taking a broad average, you will reach the specific node you want after reading only half the document. This is considerably faster and more efficient than reading and parsing the whole document every time.

The Commonly Used XmlReader Methods

The `XmlTextReader` and the `XmlNodeReader` objects have almost identical sets of properties and methods. The most commonly used methods are shown in the following table:

Method	Description
`Read`	Reads the next node into the reader object where it can be accessed. Returns `False` if there are no more nodes to read.
`ReadInnerXml`	Reads and returns the complete content of the current node as a string, containing all the markup and text of the child nodes.
`ReadOuterXml`	Reads and returns the markup of the current node and the complete content as a string, containing all the markup and text of the child nodes as well.
`ReadString`	Returns the string value of the current node.
`GetAttribute`	Returns the value of a specified attribute from the current node in the reader.
`GetRemainder`	Reads and returns the remaining XML in the source document as a string. Useful if you are copying XML from one document to another.
`MoveToxxxxxx`	Moves the current reader position. Examples are `MoveToAttribute`, `MoveToContent`, and `MoveToElement`.
`Skip`	Skips the current node in the reader and moves to the next one.
`Close`	Closes the stream or disk file.

The XmlValidatingReader Class

There is another object based on the `XmlReader` base class – the `XmlValidatingReader`. You can think of this as an `XmlTextReader` that does document validation against a schema or DTD. You can create an `XmlValidatingReader` from an existing `XmlReader` (an `XmlTextReader` or `XmlNodeReader`), from a `Stream`, or from a `String` that contains the XML to be validated.

Once the `XmlValidatingReader` is created, it can be used just like any other `XmlReader`. However, it raises an event when a schema validation error occurs, allowing you to ensure that the XML document is valid against one or more specific schemas.

The XslTransform Class

One common requirement when working with XML is the need to transform a document using XML Stylesheet Language (XSL or XSLT). The .NET Framework classes provide the `XslTransform` object, which is specially designed to perform either XSL or XSLT transformations.

The Commonly Used XslTransform Methods

The `XslTransform` class has two methods that are used for working with XML documents and XSL/XSLT stylesheets, as shown in the following table:

Method	Description
`Load`	Loads the specified XSL stylesheet and any stylesheets referenced within it by `xsl:include` elements
`Transform`	Transforms the specified XML data using the currently loaded XSL or XSLT stylesheet, and outputs the results

Let's look at some of the common tasks that need to be carried out using XML documents.

Common XML Tasks in .NET

The default page for the samples contains a link Introduction to XML Data Access in .NET. The menu page that this opens, shown in Figure 8-14, contains links to several examples of the basic .NET Framework XML data access techniques.

The first two pairs of links show how to access XML data stored in a document object in two distinct ways – using the methods and properties provided by the XML DOM, and through the new .NET `XPathNavigator` class. The next pair of links demonstrates use of the `XmlTextWriter` and `XmlTextReader` classes, and the final one shows a simple example of using the `XslTransform` class. We look at all of these classes in Chapter 11.

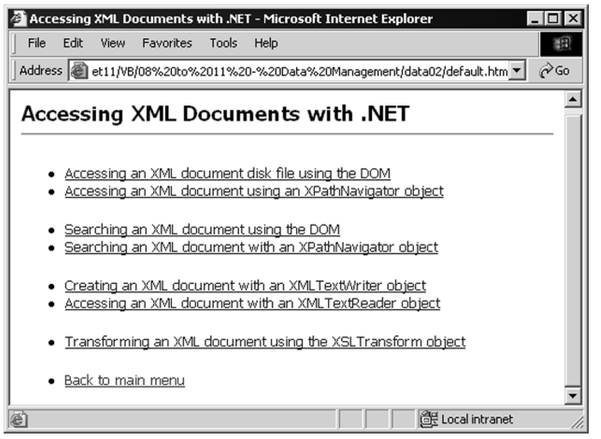

Figure 8-14

XML Document Access via the DOM

The .NET XML classes provide an XML parser object named `XmlDocument` that is W3C DOM-compliant. This is the core object for most XML-based activities carried out in .NET. You can use it to access an XML document using the same kind of code as you would with (say) the MSXML parser object. The first example page, Accessing XML documents using the DOM (`xml-via-dom.aspx`), is shown in Figure 8-15:

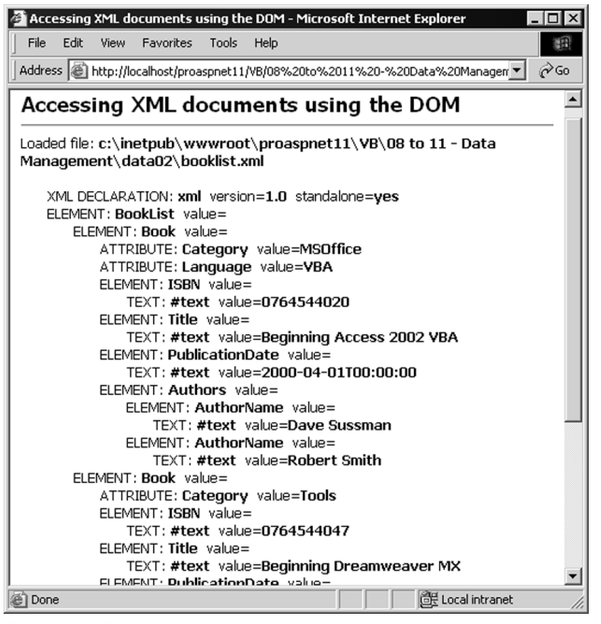

Figure 8-15

This screenshot displays the results of recursively parsing a simple XML document using DOM methods. As with all the examples in this chapter, you can use the [view source] link at the bottom of any of the sample pages to see the entire code

The XML DOM Example Code

This example, like the earlier relational data access examples, uses code in the `Page_Load` event handler to access the data and present the results within `<div>` elements located in the page. It first creates a string containing the path to the XML document, which is located in the same folder as the ASP.NET page, and then creates a new `XmlDocument` object and loads the XML file. The example contains some elementary error-handling code that we've removed here for clarity.

Now you can display the contents of the XML document by calling a custom function that recursively extracts details of each element. A function named `GetChildNodes` used here accepts a parameter an `XmlNodeList` object containing the collection of the child nodes of the current node – in this case, all the children of the document node.

> *An XML document has a single document node that has as its children the XML declaration node (such as `<?xml version="1.0"?>`), the root node of the XML (in this case `<BookList>`) and any comment nodes or processing instructions.*

The function also accepts an integer that indicates the nesting level. This is used to create the indentation of the output to show the structure more clearly. So, by calling this function initially with `objXMLDoc.ChildNodes` and `0` as the parameters, we'll start the process with the XML declaration and the root element of the document:

```
'create physical path to booklist.xml sample file (in same folder as ASPX page)
Dim strCurrentPath As String = Request.PhysicalPath
Dim strXMLPath As String = Left(strCurrentPath, InStrRev(strCurrentPath, "\")) _
                         & "booklist.xml"

'create a new XmlDocument object
Dim objXMLDoc As New XMLDocument()

'load the XML file into the XmlDocument object
objXMLDoc.Load(strXMLPath)
outDocURL.innerHTML = "Loaded file: <b>" & strXMLPath & "</b>"
```

The Custom GetChildNodes Function

The complete listing of the `GetChildNodes` function is shown in the following code. The techniques are standard W3C DOM coding practice. The principle is to iterate through all the nodes in the current `NodeList`, displaying information about each one. There are different properties available for different types of node – check the `NodeType` first, and then access the appropriate properties.

Next, if it is an `Element`-type node, iterate through all the attributes adding information about these. Finally, check if this node has any child nodes, and if so, iterate through these recursively calling the same `GetChildNodes` function.

```
Function GetChildNodes(objNodeList As XMLNodeList, intLevel As Integer) _
                                                                  As String
  Dim strNodes As String = ""
  Dim objNode As XmlNode
  Dim objAttr As XmlAttribute

  'iterate through all the child nodes for the current node
  For Each objNode In objNodeList

     'display information about this node
     strNodes = strNodes & GetIndent(intLevel) _
              & GetNodeType(objNode.NodeType) & ": <b>" & objNode.Name

     'if it is an XML Declaration node, display the 'special' properties
     If objNode.NodeType = XMLNodeType.XmlDeclaration Then

        'cast the XMLNode object to an XmlDeclaration object
        Dim objXMLDec = CType(objNode, XmlDeclaration)
        strNodes = strNodes & "</b>  version=<b>" _
                 & objXMLDec.Version & "</b>  standalone=<b>" _
                 & objXMLDec.Standalone & "</b><br />"
     Else

        'just display the generic 'value' property
        strNodes = strNodes & "</b>  value=<b>" _
                 & objNode.Value & "</b><br />"

     End If

     'if it is an Element node, iterate through the Attributes
     'collection displaying information about each attribute
     If objNode.NodeType = XMLNodeType.Element Then

        'display the attribute information for each attribute
        For Each objAttr In objNode.Attributes
           strNodes = strNodes & GetIndent(intLevel + 1) _
                    & GetNodeType(objAttr.NodeType) & ": <b>" _
                    & objAttr.Name & "</b>  value=<b>" _
                    & objAttr.Value & "</b><br />"
        Next

     End If

     'if this node has child nodes, call the same function recursively
     'to display the information for it and each of its child node

     If objNode.HasChildNodes Then
      strNodes = strNodes & GetChildNodes(objNode.childNodes, intLevel + 1)
     End If

  Next  'go to next node

  Return strNodes   'pass the result back to the caller
End Function
```

There are a couple of other minor functions that the preceding code uses. The `GetIndent` function simply takes an integer representing the current indent level and returns a string containing a suitable number of ` ` non-breaking space characters. The `GetNodeType` function looks up the numeric node type value returned from the `NodeType` property of each node, and returns a text description of the node type. Remember that you can view the code for these functions in the sample page using the [view source] link at the bottom of the page.

XML Document Access with an XPathNavigator

The second example, shown in Figure 8-16, demonstrates how you can achieve the same results as the previous example, by using the `XPathNavigator` object. The Accessing XML documents using an XPathNavigator (`xml-via-navigator.aspx`) sample page produces output that is fundamentally similar to the previous example. Notice, however, that now you get the complete content of all the child elements for the `value` of an element (all the `#text` child nodes of all the children concatenated together):

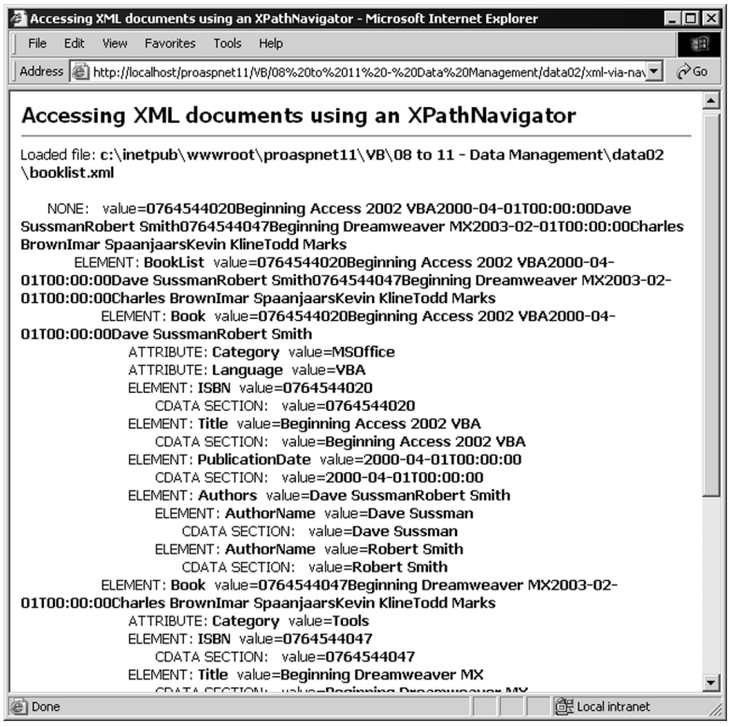

Figure 8-16

The XPathNavigator Example Code

As in the previous example, start out by locating and loading the XML document into an `XmlDocument` object (see the code that follows). If there is no error, you know that the document is well formed and

loaded successfully. However, here the code differs considerably – you create an `XPathNavigator` object based on the `XmlDocument` object (shown highlighted in the code).

To display the output, first move the current position (pointer) of the `XPathNavigator` to the document itself. Then you can call a custom recursive function named `GetXMLDocFragment` that iterates through all the nodes in the document and inserts the result into your `<div>` element elsewhere in the page. Note that this time you are calling your custom function with the new `XPathNavigator` object as the first parameter (the second is the same *indent level* parameter as used in the previous example):

```
'create physical path to booklist.xml sample file (in same folder as ASPX page)
Dim strCurrentPath As String = Request.PhysicalPath
Dim strXMLPath As String = Left(strCurrentPath, _
                         InStrRev(strCurrentPath, "\")) & "booklist.xml"

'create a new XmlDocument object and load the XML file
Dim objXMLDoc As New XmlDocument
objXMLDoc.Load(strXMLPath)
outDocURL.innerHTML = "Loaded file: <b>" & strXMLPath & "</b>"
'now ready to parse the XML document
'it must be well-formed to have loaded without error
'create a new XPathNavigator object using the XMLDocument object
Dim objXPNav As XPathNavigator = objXMLDoc.CreateNavigator()

'move the current position to the root #document node
objXPNav.MoveToRoot()

'call a recursive function to iterate through all the nodes in the
'XPathNavigator, creating a string that is placed in the <div> above
outResults.innerHTML = GetXMLDocFragment(objXPNav, 0)
```

The Custom GetXMLDocFragment Function

The `XPathNavigator` object exposes a series of properties, methods, and collections that make it easy to navigate an XML document. We use a range of these in our custom function, shown in the following code. The first step, after declaring a couple of necessary local variables, is to get the information about the current node. Notice that you use the same `GetNodeType` function as in the previous example to convert the numeric `NodeType` value into a text description of the node type.

```
Function GetXMLDocFragment(objXPNav As XPathNavigator, intLevel As Integer) _
                                                                  As String
  Dim strNodes As String = ""
  Dim intLoop As Integer

  'display information about this node
  strNodes = strNodes & GetIndent(intLevel) _
           & GetNodeType(objXPNav.NodeType) & ": " & objXPNav.Name _
           & "  value=" & objXPNav.Value & "<br />"
```

In the previous XML DOM example, you extracted the value of the node through the `XmlNode` object's `Value` property, which returned just the value of this node. In this example, the content of the XML document is being accessed through an `XPathNavigator`, and not by using the XML DOM methods. For example, to get the value of the node, we are using the `Value` property of our `objXPNav` object – an

`XPathNavigator` that is currently pointing to the node being queried. The `Value` property of a node returned by an `XPathNavigator` is a concatenation of all the child node values.

Reading the Attributes of a Node

Now you can check if this node has any attributes. If it does, iterate through them collecting information about each one. You can see in the following code how this is different from using the DOM methods, where you could iterate through the `Attributes` collection. Using an `XPathNavigator` is predominantly a forward-only *pull* technique. You need to extract the nodes from the document in the order that they appear. So, for a node that does have attributes, we move to the first attribute, process it, move to the next attribute until there are no more to process, and then move back to the previous position using the `MoveToParent` method:

```
'see if this node has any Attributes
If objXPNav.HasAttributes Then
   'move to the first attribute
   objXPNav.MoveToFirstAttribute()
   Do
     'display the information about it
     strNodes = strNodes & GetIndent(intLevel + 1) _
              & GetNodeType(objXPNav.NodeType) & ": " & objXPNav.Name _
              & "  value=" & objXPNav.Value & "<br />"

   Loop While objXPNav.MoveToNextAttribute()

   'then move back to the parent node (that is the element itself)
   objXPNav.MoveToParent()

End If
```

Reading the Child Nodes for a Node

You can see if the current node has any child nodes by checking the `HasChildren` property. If it does, you need to move to the first child node and recursively call the function for that node – incrementing the *level* parameter to get the correct indenting of the results. Then you can move back to the previous position (the parent) and continue, as shown:

```
'see if this node has any child nodes
If objXPNav.HasChildren Then

   'move to the first child node of the current node
   objXPNav.MoveToFirstChild()
   Do
     'recursively call this function to display the child node fragment
     strNodes = strNodes & GetXMLDocFragment(objXPNav, intLevel + 1)

   Loop While objXPNav.MoveToNext()

   'move back to the parent node - the node we started from when we
   'moved to the first child node - could have used Push and Pop instead
   objXPNav.MoveToParent()

End If
```

Reading the Sibling Nodes for a Node

So far you've only processed the current node, its attributes, and its child nodes (if any). You need to repeat the process for all the following sibling (element) nodes as well. This is achieved using the `MoveToNext` method, and by calling the recursive function again for each one, as shown:

```
    Do While objXPNav.MoveToNext()

       'recursively call this function to display this sibling node
       'and its attributes and child nodes
       strNodes = strNodes & GetXMLDocFragment(objXPNav, intLevel)

    Loop

    Return strNodes  'pass the result back to the caller

  End Function
```

Searching an XML Document

The second pair of links in the menu page opens two examples that search for specific element values within an XML document, rather than displaying the entire document. The two examples are Searching an XML document using the DOM (`search-dom.aspx`) and Searching an XML document with an XPathNavigator (`search-navigator.aspx`). The task is to retrieve the values of all the `<AuthorName>` elements within the document. You can run these samples to see the results. Figure 8-17 shows the XML DOM version:

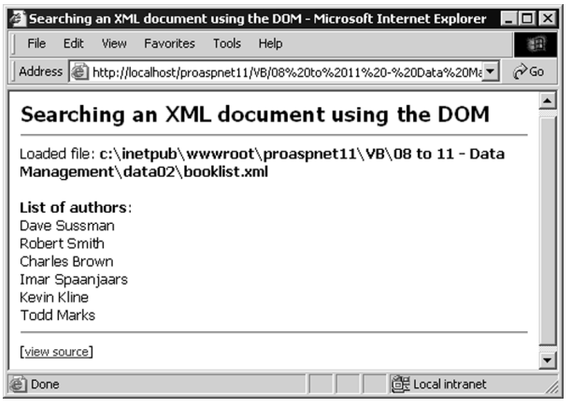

Figure 8-17

Using the DOM Methods

Using the DOM methods, you can take advantage of the very useful `GetElementsByTagname` method that the `XmlDocument` object exposes. This method can be used to create a collection of matching elements as an `XmlNodeList`, as shown in the following code, and then iterate through the collection displaying the values of the `#text` child node for each one.

```
Dim strResults As String = "<b>List of authors</b>:<br />"

'create a NodeList collection of all matching child nodes
Dim colElements As XmlNodeList
colElements = objXMLDoc.GetElementsByTagname("AuthorName")

'iterate through collection getting values of child #text nodes for each one
For Each objNode In colElements
  strResults += objNode.FirstChild().Value & "<br />"
Next
'then display the result
outResults.innerHTML = strResults
```

Remember that an element's value is stored in a `#text`-type child node of the element node – it's not the value of the element node itself. You can clearly see this in the previous examples that displayed all the nodes in the document.

Using an XPathNavigator

You've already seen how to create an `XPathNavigator` for an `XmlDocument` and use it to traverse the document. The `XPathNavigator` also provides the `Select` method, which takes an `XPath` expression and selects all matching nodes or fragments within the document. You can then traverse the set of selected nodes and extract the values you want.

You can also improve performance by using the lighter and faster `XPathDocument` object to hold your XML document rather than the W3C-compliant `XmlDocument` object.

Creating an XPathDocument and XPathNavigator Object

The following code in the `Page_Load` event handler first creates an `XPathDocument` instance and loads the XML document into it. However, in this case, you must use the constructor for the `XPathDocument` to load the XML, because there is no `Load` method for this class. While you can create an `XPathDocument` from a `Stream`, a `TextReader` or an `XmlReader`, the easiest way when you have an XML disk file is to specify the path and name of that file. The code then creates an `XPathNavigator` object for this document.

```
'declare a variable to hold an XPathDocument instance
Dim objXPathDoc As XPathDocument

'create XPathDocument object and load the XML file
objXPathDoc = New XPathDocument(strXMLPath)

'create an XPathNavigator based on this document
Dim objXPNav As XPathNavigator = objXPathDoc.CreateNavigator()
```

Selecting the Nodes and Displaying the Results

Now you can execute the `Select` method of the `XPathNavigator` with an appropriate XPath expression. The result will be an `XPathNodeIterator` object that contains the matching node(s). Then, as shown in the following code, it's simply a matter of iterating through the selected nodes collecting their values. Each *node* in the `XPathNodeIterator` is itself an `XPathNavigator` based on this node

within the document. This new `XPathNavigator` has `Name` and `Value` properties that reflect the values for the current node.

```
Dim strResults As String = "<b>List of authors</b>:<br />"
     'select all AuthorName nodes into XPathNodeIterator object
     'using an appropriate XPath expression
Dim objXPIter As XPathNodeIterator
objXPIter = objXPNav.Select("descendant::AuthorName")

Do While objXPIter.MoveNext()
  'get the value and add to the 'results' string
  strResults += objXPIter.Current.Value & "<br />"
Loop

outResults.innerHTML = strResults   'display the result
```

You need to consider the task you want to achieve quite carefully when deciding whether to use an `XPathNavigator` object or the XML DOM methods. Of course, as you can create an `XPathNavigator` based on an existing `XmlDocument` object (as well as on an `XPathDocument`), you can use both where this is appropriate. Also remember to choose the lighter and faster `XPathDocument` if you don't need to access the XML DOM (in other words when you can perform all the tasks you require using an `XPathNavigator`).

An XML TextWriter Object Example

The Creating an XML document with an XMLTextWriter object (`xml-via-textwriter.aspx`) example demonstrates how to use the `XmlTextWriter` object to quickly create a new XML document as a disk file. It writes to the file a series of elements and attributes that make up the document, and then reads the document back from disk and displays it, as shown in Figure 8-18:

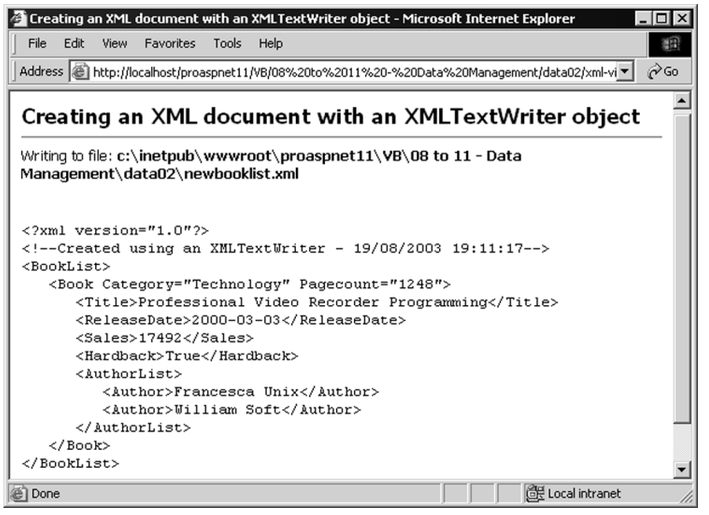

Figure 8-18

The XMLTextWriter Example Code

To create the new XML document, first create a suitable path and filename so that the new file will be placed in the same folder as the ASP page, as shown in the following code. Then create the `XmlTextWriter` object instance. Specify the path to the new file as the first parameter to the constructor, and `Nothing` (`null` in C#)for the second. The second parameter is the encoding required for the file, defined as an `Encoding` object. If you set this parameter to `Nothing`, the default encoding UTF-8 is used.

Next, set the properties of the `XmlTextWriter`. In the example, we want the document to be indented (to show the structure more clearly), with each level of indent being three space characters. Then we're ready to start writing the new document. The `WriteStartDocument` method creates the opening XML declaration, and this is followed with a comment indicating the date and time that the document was created:

```
'create physical path for the new file (in same folder as ASPX page)
Dim strCurrentPath As String = Request.PhysicalPath
Dim strXMLPath As String = Left(strCurrentPath, InStrRev(strCurrentPath, "\")) _
                         & "newbooklist.xml"

'declare a variable to hold an XmlTextWriter object
Dim objXMLWriter As XmlTextWriter

'create a new objXMLWriter object for the XML file
objXMLWriter = New XmlTextWriter(strXMLPath, Nothing)
outDocURL.innerHTML = "Writing to file: <b>" & strXMLPath & "</b>"

'now ready to write (or "push") the nodes for the new XML document
'turn on indented formatting and set indent to 3 chararcters
objXMLWriter.Formatting = Formatting.Indented
objXMLWriter.Indentation = 3

'start the document with the XML declaration tag
objXMLWriter.WriteStartDocument()

'write a comment element including the current date/time
objXMLWriter.WriteComment("Created using an XMLTextWriter - " & Now())
```

Writing Elements and Attributes

The next step is to write the opening tag of the `<BookList>` root element. The `WriteStartElement` does this for you; follow it with the opening `<Book>` element tag, as shown in the following code. We also want to add two attributes to the `<Book>` element. For these, use the `WriteAttributeString` method to create an attribute from a text string. Where the value for the attribute is a numeric (or other non-`String`) data type, you must convert it to a string first:

```
objXMLWriter.WriteStartElement("BookList")
objXMLWriter.WriteStartElement("Book")

'add two attributes to this element's opening tag
objXMLWriter.WriteAttributeString("Category", "Technology")
Dim intPageCount As Integer = 1248   'numeric value to convert
objXMLWriter.WriteAttributeString("Pagecount", intPageCount.ToString("G"))
```

The next step is to write the four elements that form the content of the <Book> element that's already opened. Use the WriteElementString method, which writes a complete element (not just the opening tag like the WriteStartElement method we used earlier does). Note that the actual *content* of the element in the final document is always text (XML documents are plain text). Therefore, you have to convert non-String type values to a string first, as shown:

```
'write four elements, using different source data types

objXMLWriter.WriteElementString("Title", _
                          "Professional Video Recorder Programming")
Dim datReleaseDate As DateTime = #02/02/2002#
objXMLWriter.WriteElementString("ReleaseDate", _
                          datReleaseDate.ToString("yyyy-MM-dd"))
Dim intSales As Integer = 17492
objXMLWriter.WriteElementString("Sales", intSales.ToString("G"))
Dim blnHardback As Boolean = True
objXMLWriter.WriteElementString("Hardback", blnHardback.ToString())
```

Next, as shown in the following code, we want to write the <AuthorList> element and its child <Author> elements. You need to open the <AuthorList> element and then write the child elements. Afterwards, you can create the closing </AuthorList> tag simply by calling the WriteEndElement method. This automatically closes the most recently opened element.

```
'write the opening tag for the <AuthorList> child element
objXMLWriter.WriteStartElement("AuthorList")

'add two <Author> elements
objXMLWriter.WriteElementString("Author", "Francesca Unix")
objXMLWriter.WriteElementString("Author", "William Soft")

'close the <AuthorList> element
objXMLWriter.WriteEndElement()
```

To finish the document, just close the <Book> element and the root <BookList> element. Then flush the output to the disk file and close the XmlTextWriter, as shown in the following code. Always remember to call the Close method; otherwise the disk file will remain locked. You don't actually *have* to call the Flush method here, as closing the XmlTextWriter has the same effect, but you can call Flush to force the part-formed document to be written to disk whenever you wish:

```
'close the <Book> element
objXMLWriter.WriteEndElement()

'close the root <BookList> element
objXMLWriter.WriteEndElement()
objXMLWriter.Flush()
objXMLWriter.Close()
```

Displaying the New XML Document

Now that you've got your new XML document written to a disk file, you can read it back and display it. To do this, use a StreamReader object, as shown in the following code. Open the file, read the entire content into a string variable, and close the file. Then you can insert the string into a element

elsewhere on the page to display it. Add <pre> elements (you could use <xmp> instead) to maintain the indentation and line breaks in the document when displayed in the browser.

```
Dim strXMLResult As String
Dim objSR As StreamReader = File.OpenText(strXMLPath)
strXMLResult = objSR.ReadToEnd()
objSR.Close
objSR = Nothing
outResults.innerHTML = "<pre>" & Server.HtmlEncode(strXMLResult) & "</pre>"
```

An XML TextReader Object Example

OK, so you can create an XML document as a disk file with an `XmlTextWriter`. The obvious next step is to read a disk file back using an `XmlTextReader` object. The Accessing an XML document with an XMLTextReader object (`xml-via-textreader.aspx`) example does just that (though with a different XML document).

Figure 8-19 shows a list of the nodes found in the sample `booklist.xml` document. For each node, the page shows the type of node, and the node name and value (if applicable – some types of node have no name and some types have no value):

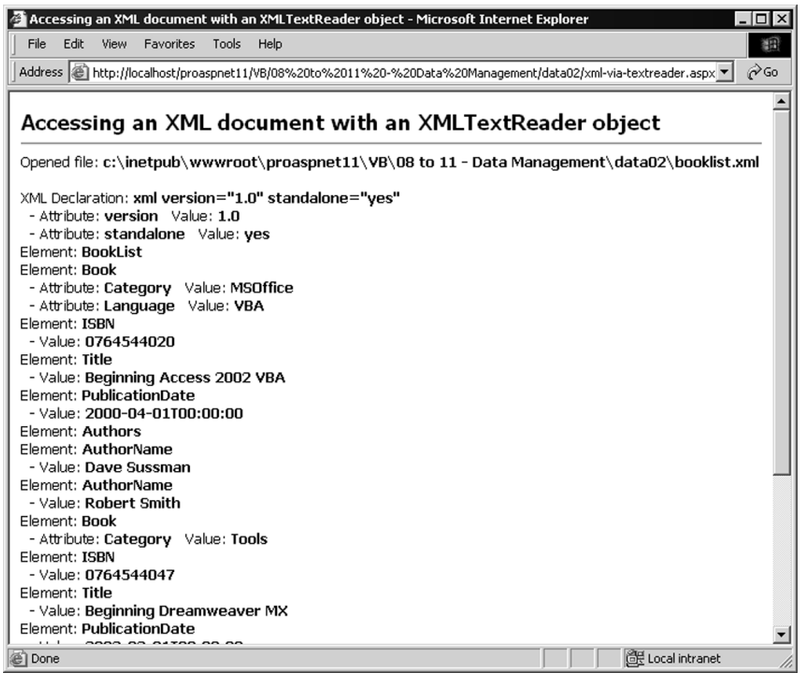

Figure 8-19

The XMLTextReader Example Code

As in the previous example, the first step is to build the path to the file that'll be opened – in this case, `booklist.xml` in the same folder as the ASP page. Next, as shown in the following code, you can declare an `XmlTextReader` object, passing the path to the file that you want to open as the parameter to the constructor.

Reading the XML document is just a matter of calling the `Read` method to return each node. This returns `False` if there are no more nodes to read. For each node you find, examine the `NodeType` property to see what kind of node it is. Depending on the node type, there are different properties available that you can access to build your `results` string.

```
'create physical path to booklist.xml sample file (in same folder as ASPX page)
Dim strCurrentPath As String = Request.PhysicalPath
Dim strXMLPath As String = Left(strCurrentPath, InStrRev(strCurrentPath, "\")) _
                        & "booklist.xml"

'declare a variable to hold an XmlTextReader object
Dim objXMLReader As XmlTextReader

'create a new XmlTextReader object for the XML file
objXMLReader = New XmlTextReader(strXMLPath)
outDocURL.innerHTML = "Opened file: <b>" & strXMLPath & "</b>"

'now ready to read (or "pull") the nodes of the XML document
Dim strNodeResult As String = ""
Dim objNodeType As XmlNodeType

'read each node in turn - returns False if no more nodes to read
Do While objXMLReader.Read()

  'select on the type of the node (these are only some of the types)
  objNodeType = objXMLReader.NodeType

  Select Case objNodeType

    Case XmlNodeType.XmlDeclaration:
      'get the name and value
      strNodeResult += "XML Declaration: <b>" & objXMLReader.Name _
                    & " " & objXMLReader.Value & "</b><br />"

    Case XmlNodeType.Element:
      'just get the name, any value will be in next (#text) node
      strNodeResult += "Element: <b>" & objXMLReader.Name & "</b><br />"

    Case XmlNodeType.Text:
      'just display the value, node name is "#text" in this case
       strNodeResult += "  - Value: <b>" & objXMLReader.Value _
                     & "</b><br />"

  End Select
```

The `XmlTextReader` returns the document node-by-node when you call the `Read` method. However, an element-type node that has attributes is returned as a complete entity during a single `Read` method call,

and so you have to examine each node as you read it to see if it is an element that has attributes. If it does, as shown in the following code, iterate through these by using the `MoveToFirstAttribute` or the `MoveToNextAttribute` methods. After processing the current node, you go back and handle the next one. And after the `Do` loop is complete (in other words, after you've processed all the nodes returned by successive `Read` method calls), close the `XmlTextReader` object and display the results in a `<div>` element elsewhere in the page:

```
    'see if this node has any attributes
    If objXMLReader.AttributeCount > 0 Then
      'iterate through the attributes by moving to the next one
      'could use MoveToFirstAttribute but MoveToNextAttribute does
      'the same when the current node is an element-type node
      Do While objXMLReader.MoveToNextAttribute()
         'get the attribute name and value
         strNodeResult +=  "- Attribute: " & objXMLReader.Name _
                     & " Value: " & objXMLReader.Value & "<br />"
      Loop
    End If

  Loop   'and read the next node
  objXMLReader.Close()   'finished with the reader so close it
  outResults.innerHTML = strNodeResult   'display the results in the page
```

An XSL Transform Object Example

The final example in this chapter shows one other task that is regularly required when working with XML data, and which .NET makes easy. You can use XML stylesheets written in XSL or XSLT to transform an XML document into another format, or to change its structure or content.

The Transforming an XML document using the XSLTransform object (`xsl-transform.aspx`) example page demonstrates a simple transformation using the `booklist.xml` file from the previous example and an XSLT stylesheet named `booklist.xsl`. The result of the transformation is written to disk as `booklist.html`. As shown in Figure 8-20, you can use the links in the page to open the XML document, the stylesheet, and the final HTML page:

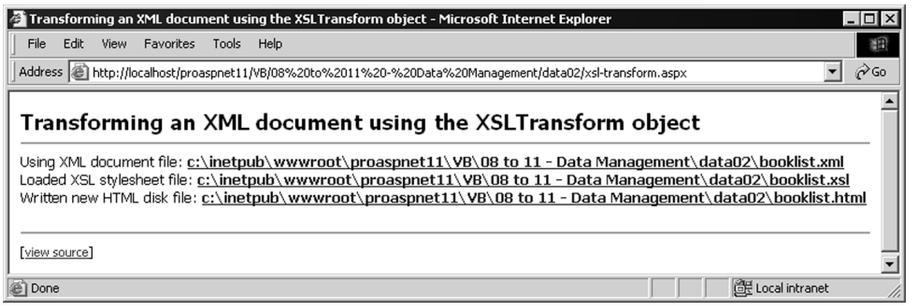

Figure 8-20

You must run this page in a browser running on the same machine as the web server to be able to open the linked files using the absolute physical paths.

The XSLTransform Example Code

There is surprisingly little code required to perform the transformation (you can view the code in the example using the [view source] link at the bottom of the page). First, create an `XslTransform` object and load the XSL stylesheet into it from disk. Then you can perform the transformation directly using the XSL file in the `XslTransform` object and the XML file path held in a variable named `strXMLPath`, as shown in the following code:

```
'create a new XslTransform object
Dim objTransform As New XslTransform()

'load the XSL stylesheet into the XslTransform object
objTransform.Load(strXSLPath)

'perform the transformation
objTransform.Transform(strXMLPath, strHTMLPath)
```

The result is sent to the disk file specified by the variable named `strHTMLPath`. Figure 8-21 shows the resulting HTML page:

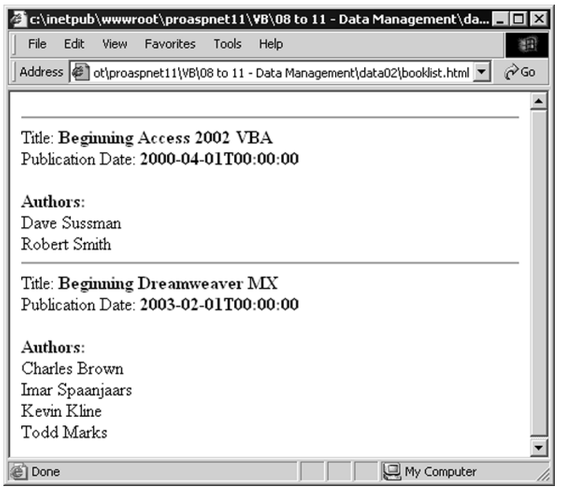

Figure 8-21

This is just one way to use the `XslTransform` object (in fact, the simplest way) and you'll see a more complex example at the end of Chapter 11, where XML data management techniques are discussed.

This section was a basic introduction to working with XML in the .NET environment. The next two chapters look at relational data management, but we'll see how the relational and XML data models are quite thoroughly integrated under .NET. Then, in Chapter 11, we'll come back to XML and look in more depth at some of the other techniques that .NET provides to make even the most complex tasks much easier that ever before. Let's try to make some sense of the whole relational versus XML issue.

Choosing a Data Storage Methodology

Having seen both relational and XML data access in action within the .NET Framework (albeit in a fairly basic way), how do you decide on a data storage methodology? The simple answer is that, with the advent of .NET, you really don't need to worry about this anymore.

Years ago, one of the main directions in data storage and access was the construction of huge data depositaries or data warehouses where all the data your organization required was stored in a massive central database. While this might still suit some situations (such as a government tax office) it has become clear that it is not a generally practical approach in today's distributed and disconnected computing world.

In fact, there has been even less centralization of data over recent years, and the drive now is far more towards the provision of access through common methods to all kinds of remote and non-centralized data. As an example, the Internet contains vast quantities of data in myriad different formats, but we increasingly need to be able to get at this data in a structured and standard way.

Likewise, in an office environment, the promised takeover of thin client computing has not really taken place yet. People like to store information locally as they work, and use it when disconnected from the corporate network. In some cases, such as the traveling salesperson with a laptop computer, this is the prime requirement when working with corporate data.

Access and Manipulation Is the Key

In fact, it's obvious that *where* and (to some extent) *how* we store data is not important. The crux of the matter is how we can *access* and *manipulate* that data – in whatever format it's stored and wherever it resides. As you saw at the start of this chapter, this is what has been the driving force behind the adoption of XML, and the design of the .NET data access libraries.

So, what issues should one consider when implementing a data store, and which data access technique is most appropriate for that data? The answer lies more in the nature of the data, and the way we need to use it. For example, highly structured data, such as stock lists or customer details, is well suited to storage in a relational database such as SQL Server or Oracle, or MS Access on the desktop. However, unstructured data, such as reports, data sheets, email messages, family trees, and other common everyday scenarios, is more suited to storage using the tree-like metaphor of XML.

Likewise, if we regularly need to access parts of the data in specific ways, or all the data on a very regular basis, the relational database is probably the most efficient. It is optimized to provide indexing and other features to give the best performance. But if we usually access the entire data entity in one go, or access it only rarely, an XML-based approach is probably the best choice. And, being basically just text files, XML documents are easy to archive and retrieve.

Of course, in some cases, you don't actually get to choose the data storage format. For example, your email server and your fax server probably have dedicated storage mechanisms that can't be changed. In such cases, you have to make do with what's there, or change to another product.

A New Approach to Querying

Another point to be aware of is that you should not base your data format decision on *current* querying technologies. One of the major issues at the moment is that each data storage format has its own specific techniques for querying and extracting data; for example, SQL for relational data and XSLT for XML data. If you want to perform a query across different types of data, you generally have to convert the all to the same *type* first.

However, this is set to change with the growing realization that a new querying technology, called XML Query Language or XQuery, will be able to integrate different types of data under a universal query mechanism. XQuery has been called *SQL querying for XML data* because it uses a syntax that is similar to the widely accepted SQL standards, and yet can be applied to XML documents.

And as relational data stores such as SQL Server become increasingly XML-capable, and the tools to access and manage XML data inside a relational database improve, XQuery can also be used with suitable relational databases. In future releases of .NET, this scenario will become a core part of the way you query data in mixed environments.

There is a preview of the way Microsoft are approaching XQuery, at least as far as working with XML is concerned, on the special Web site they have set up at http://xqueryservices.com. You can experiment with XQuery online, or download the Microsoft XQuery demo to run on your server.

Transport Protocols Are the Future

Once you've decided on the storage mechanism for your data, the next important decision comes when you consider how you will transport this data from one place to another. Here, there is probably only one good solution that matches the requirements of the future. There's no doubt that we'll face increasing needs to interface with other systems and other organizations as time goes by, and for this, a standard data interchange format will be an absolute necessity.

The only obvious choice today is XML (and the associated standards such as SOAP and other industry-specific implementations of XML). XML is independent of the platform, application, and operating system, and so it provides the best chance for interoperability.

In fact, Microsoft BizTalk Server and similar systems can handle the transmission and guaranteed delivery of data in XML format over almost any kind of network, as well as the conversion to and from other formats. Using the tools available today and in the near future, you can transform an XML document into almost any other document type on demand – and often transform any non-XML document or data into XML as well.

And .NET Is a Great Solution

So, if the transport protocol and transmission format for data are going to be XML-based, and the data storage and manipulation could be through any existing or new technology, what you really need is a solid, reliable, and wide-ranging technique to connect to any kind of data store, and work with any kind of data.

This is where the combination of the relational and XML data access techniques provided by the .NET Framework comes in. As you've seen (and will see), you can use the .NET data access classes to connect

to almost any kind of data store – be it a mail server, a relational database, an office application document, an XML document, or whatever. Then, once you have extracted data, you can convert it between XML and traditional relational rowsets at will – and update the data store or save it to disk in almost any format you need.

Summary

In this chapter, we've started to explore the possibilities for working with data within the .NET Framework, based on ASP.NET, the .NET data access classes, and the extended XML technologies that they provide. We overviewed the two main topic areas, relational and XML data access, then examined in more depth the core objects that are provided within these topic areas.

One of the problems with learning to use the new techniques is the complexity that can arise from the huge number of properties, methods, and events that these new objects expose. Many are rarely used, and so we've tried to make it easier by just concentrating on the commonly used techniques rather than trying to document each one in minute detail.

> *An excellent reference to all the properties, methods, and events of all the .NET Framework objects is included within the SDK that is provided with the framework. Simply open the* Class Library *within the* Reference *section, or search for the object or class name using the Index or Search feature of the SDK.*

What you should have gained by now is an understanding of the core objects and the basic techniques we use when working with them. We'll continue this in the next three chapters as well.

The next chapter looks specifically at relational data access within .NET, and how to use more advanced techniques – in particular working with relational data sets and tables, editing them, and displaying the data they contain.