Programmer to Programmer ™

# Professional C# 2nd Edition

**Written and tested for final release of .NET v1.0**

Simon Robinson, K. Scott Allen, Ollie Cornes, Jay Glynn, Zach Greenvoss, Burton Harvey, Christian Nagel, Morgan Skinner, Karli Watson

Wrox technical support at: support@wrox.com
Updates and source code at: www.wrox.com
Peer discussion at: p2p.wrox.com

# What you need to use this book

The following list is the recommended operating system requirements for running the C# code in this book:

- Windows 2000 Professional or higher with IIS installed
- Windows XP Professional with IIS installed
- Visual Studio .NET Professional or higher

The book is intended for experienced developers, probably from a VB 6, C++, or Java background. Although previous experience of C# or .NET programming is useful, it is not required.

# Summary of Contents

# Data Access with .NET

In this chapter, we'll be discussing how to get at data from your C# programs using ADO.NET. Over the course of this chapter, we'll be covering the following areas:

- Connecting to the database – how to utilize the new `SqlConnection` and `OleDbConnection` classes to connect to and disconnect from the database. Connections utilize the same form of connection strings as did OLEDB providers (and therefore ADO), and these are briefly discussed. We then go through a set of best practices for utilizing database connections, and show how to ensure that a connection is closed after use, which is one of the sources of poor application performance.

- Executing Commands – ADO.NET has the concept of a command object, which may execute SQL directly, or may issue a stored procedure with return values. The various options on command objects are discussed in depth, with examples to show how commands can be used for each of the options presented by the `Sql` and `OleDB` classes.

- Stored Procedures – How to call stored procedures using command objects, and how the results of those stored procedures may be integrated back into the data cached on the client.

- The ADO.NET object model – this is significantly different from the objects available with ADO, and the `DataSet`, `DataTable`, `DataRow`, and `DataColumn` classes are all discussed. A `DataSet` can also include relationships between tables, and also constraints. These issues are also discussed.

- Using XML and XML Schemas – ADO.NET is built upon an XML framework, so we'll examine how some of the support for XML has been added to the data classes.

We'll also present a guide to the naming conventions that preside in the world of ADO.NET and explain some of the reasoning behind them. First, though, let's take a brief tour of ADO.NET and see what's on offer.

# ADO.NET Overview

Like most of the .NET Framework, ADO.NET is more than just a thin veneer over some existing API. The similarity to ADO is in name only – the classes and method of accessing data are completely different.

ADO (Microsoft's ActiveX Data Objects) was a library of COM components that has had many incarnations over the last few years. Currently at version 2.7, ADO consists primarily of the `Connection`, `Command`, `Recordset`, and `Field` objects. A connection would be opened to the database, some data selected into a recordset, consisting of fields, that data would then be manipulated, updated on the server, and the connection would be closed. ADO also introduced the concept of a disconnected recordset, which was used where keeping the connection open for long periods of time was not desirable.

There were several problems that ADO did not address satisfactorily, most notably the unwieldiness (in physical size) of a disconnected recordset. This support was more necessary than ever with the evolution of "web-centric" computing, so a fresh approach was taken. There are a number of similarities between ADO.NET programming and ADO (not only the name), so upgrading from ADO shouldn't be too difficult. What's more, if you're using SQL Server, there's a fantastic new set of managed classes that are very highly tuned to squeeze maximum performance out of the database. This alone should be reason enough to move.

ADO.NET ships with two database client namespaces – one for SQL Server, the other for databases exposed through an OLE DB interface. If your database of choice has an OLE DB driver, you will be able to easily connect to it from .NET – just use the OLE DB classes and connect through your current database driver.

## Namespaces

All of the examples in this chapter access data in one way or another. The following namespaces expose the classes and interfaces used in .NET data access:

- `System.Data` – All generic data access classes
- `System.Data.Common` – Classes shared (or overridden) by individual data providers
- `System.Data.OleDb` – OLE DB provider classes
- `System.Data.SqlClient` – SQL Server provider classes
- `System.Data.SqlTypes` – SQL Server data types

The main classes in ADO.NET are listed below:

# Shared Classes

ADO.NET contains a number of classes that are used regardless of whether you are using the SQL Server classes or the OLE DB classes.

The following are contained in the `System.Data` namespace:

- `DataSet` – This object may contain a set of `DataTables`, can include relationships between these tables, and is designed for disconnected use.
- `DataTable` – A container of data. A `DataTable` consists of one or more `DataColumns`, and when populated will have one or more `DataRows` containing data.
- `DataRow` – A number of values, akin to a row from a database table, or a row from a spreadsheet.
- `DataColumn` – Contains the definition of a column, such as the name and data type.
- `DataRelation` – A link between two `DataTables` within a `DataSet`. Used for foreign key and master/detail relationships.
- `Constraint` – Defines a rule for a `DataColumn` (or set of data columns), such as unique values.

These next two classes are to be found in the `System.Data.Common` namespace:

- `DataColumnMapping` – Maps the name of a column from the database with the name of a column within a `DataTable`.
- `DataTableMapping` – Maps a table name from the database to a `DataTable` within a `DataSet`.

# Database Specific Classes

In addition to the shared classes above, ADO.NET contains a number of database-specific classes shown below. These classes implement a set of standard interfaces defined within the `System.Data` namespace, allowing the classes to be used if required in a generic manner. For example, both the `SqlConnection` and `OleDbConnection` classes implement the `IDbConnection` interface.

- `SqlCommand`, `OleDbCommand` – A wrapper for SQL statements or stored procedure calls.
- `SqlCommandBuilder`, `OleDbCommandBuilder` – A class used to generate SQL commands (such as `INSERT`, `UPDATE`, and `DELETE` statements) from a `SELECT` statement.
- `SqlConnection`, `OleDbConnection` – The connection to the database. Similar to an ADO `Connection`.
- `SqlDataAdapter`, `OleDbDataAdapter` – A class used to hold select, insert, update, and delete commands, which are then used to populate a `DataSet` and update the `Database`.
- `SqlDataReader`, `OleDbDataReader` – A forward only, connected data reader.
- `SqlParameter`, `OleDbParameter` – Defines a parameter to a stored procedure.
- `SqlTransaction`, `OleDbTransaction` – A database transaction, wrapped in an object.

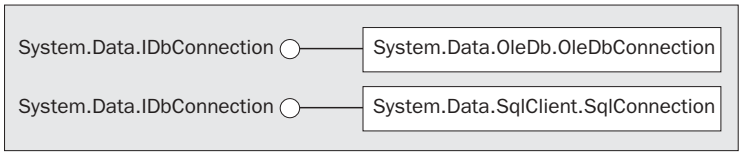
The most important new feature of the ADO.NET classes is that they are designed to work in a disconnected manner, which is important in today's highly web-centric world. It is now common practice to architect a service (such as an online bookshop) to connect to a server, retrieve some data, and then work on that data on the client PC before reconnecting and passing the data back for processing. The disconnected nature of ADO.NET enables this type of behavior.

ADO 2.1 introduced the disconnected recordset, which would permit data to be retrieved from a database, passed to the client for processing, and then reattached to the server. This was often cumbersome to use, as disconnected behavior hadn't been designed in from the start. The ADO.NET classes are different – in all but one case (the `Sql`/`OleDb` `DataReader`) they are designed for use offline from the database.

> *The classes and interfaces used for data access in the .NET Framework will be introduced as the chapter continues. I will mainly concentrate on the `Sql` classes when connecting to the database, because the Framework SDK samples install an MSDE database (SQL Server). In most cases the `OleDb` classes mimic exactly the `Sql` code.*

# Using Database Connections

In order to access the database, you need to provide connection parameters, such as the machine that the database is running on, and possibly your login credentials. Anyone who has worked with ADO will be immediately familiar with the .NET connection classes, `OleDbConnection` and `SqlConnection`:




The following code snippet illustrates how to create, open, and close a connection to the `Northwind` database. In the examples within this chapter I use the `Northwind` database, which is installed with the .NET Framework SDK samples:

```
using System.Data.SqlClient;

string source = "server=(local)\\NetSDK;" +
                "uid=QSUser;pwd=QSPassword;" +
                "database=Northwind";
SqlConnection conn = new SqlConnection(source);
conn.Open();

// Do something useful

conn.Close();
```

The connection string should be very familiar to you if you've ever used ADO or OLE DB before – indeed, you should be able to cut and paste from your old code if you use the `OleDb` provider. In the example connection string, the parameters used are as follows. The parameters are delimited by a semicolon in the connection string.

- `server=(local)\\NetSDK` – This denotes the database server to connect to. SQL Server permits a number of separate database server processes to be running on the same machine, so here we're connecting to the `NetSDK` processes on the local machine.
- `uid=QSUser` – This parameter describes the database user. You can also use `User ID`.
- `pwd=QSPassword` – And this is the password for that user. The .NET SDK comes with a set of sample databases, and this user/password combination is added during the installation of the .NET samples. You can also use `Password`.
- `database=Northwind` – This describes the database instance to connect to – each SQL Server process can expose several database instances.

The example opens a database connection using the defined connection string, and then closes that connection. Once the connection has been opened, you can issue commands against the data source, and when you're finished, the connection can be closed.

SQL Server has another mode of authentication – it can use Windows integrated security, so that the credentials supplied at logon are passed through to SQL Server. This is catered for by removing the `uid` and `pwd` portions of the connection string , and adding in `Integrated Security=SSPI`.

In the download code available for this chapter, you will find a file `Login.cs` that simplifies the examples in this chapter. It is linked to all the example code, and includes database connection information used for the examples; you can alter this to supply your own server name, user, and password as appropriate. This by default uses Windows integrated security; however, you can change the username and password as appropriate.

Now that we know how to open connections, before we move on we should consider some good practices concerning the handling of connections.

# Using Connections Efficiently

In general, when using "scarce" resources in .NET, such as database connections, windows, or graphics objects, it is good practice to ensure that each resource is closed after use. Although the designers of .NET have implemented automatic garbage collection, which will tidy up eventually, it is necessary to actively release resources as early as possible.

This is all too apparent when writing code that accesses a database, as keeping a connection open for slightly longer than necessary can affect other sessions. In extreme circumstances, not closing a connection can lock other users out of an entire set of tables, considerably hurting application performance. Closing database connections should be considered mandatory, so this section shows how to structure your code so as to minimize the risk of leaving a resource open.

There are two main ways to ensure that database connections and the like are released after use.

## Option One – try/catch/finally

The first option to ensure that resources are cleaned up is to utilize `try...catch...finally` blocks, and ensure that you close any open connections within the finally block. Here's a short example:

```
try
{
   // Open the connection
   conn.Open();
   // Do something useful
}
catch ( Exception ex )
{
   // Do something about the exception
}
finally
{
   // Ensure that the connection is freed
   conn.Close ( ) ;
}
```

Within the `finally` block you can release any resources you have used. The only trouble with this method is that you have to ensure that you close the connection – it is all too easy to forget to add in the `finally` clause, so something less prone to vagaries in coding style might be worthwhile.

Also, you may find that you open a number of resources (say two database connections and a file) within a given method, so the cascading of `try`...`catch`...`finally` blocks can sometimes become less easy to read. There is however another way to guarantee resource cleanup – the `using` statement.

## Option Two – The using Block Statement

During development of C#, .NET's method of clearing up objects after they are no longer referenced using nondeterministic destruction became a topic of very heated discussion. In C++, as soon as an object went out of scope, its destructor would be automatically called. This was great news for designers of resource-based classes, as the destructor was the ideal place to close the resource if the user had forgotten to do so. A C++ destructor is called in any and every situation when an object goes out of scope – so for instance if an exception was raised and not caught, all objects with destructors would have them called.

With C# and the other managed languages, there is no concept of automatic, deterministic destruction – instead there is the garbage collector, which will dispose of resources at some point in the future. What makes this nondeterministic is that you have little say over when this process actually happens. Forgetting to close a database connection could cause all sorts of problems for a .NET executable. Luckily, help is at hand. The following code demonstrates how to use the `using` clause to ensure that objects that implement the `IDisposable` interface (discussed in Chapter 2) are cleared up immediately the block exits.

```
string source = "server=(local)\\NetSDK;" +
                "uid=QSUser;pwd=QSPassword;" +
                "database=Northwind";

using ( SqlConnection conn = new SqlConnection ( source ) )
```

```
{
   // Open the connection
   conn.Open ( ) ;

   // Do something useful
}
```

The `using` clause was introduced in Chapter 2. The object within the `using` clause must implement the `IDisposable` interface, or a compilation error will be flagged if the object does not support this interface. The `Dispose()` method will automatically be called on exiting the `using` block.

Looking at the IL code for the `Dispose()` method of `SqlConnection` (and `OleDbConnection`), both of these check the current state of the connection object, and if open will call the `Close()` method.

When programming, you should use at least one of these methods, and probably both. Wherever you acquire resources it is good practice to utilize the `using ()` statement, as even though we all mean to write the `Close()` statement, sometimes we forget, and in the face of exceptions the `using` clause does the right thing. There is no substitute for good exception handling either, so in most instances I would suggest you use both methods together as in the following example:

```
try
{
   using (SqlConnection conn = new SqlConnection ( source ))
   {
      // Open the connection
      conn.Open ( ) ;

      // Do something useful

      // Close it myself
      conn.Close ( ) ;
   }
}
catch (Exception e)
{
   // Do something with the exception here...
}
```

Here I have explicitly called `Close()` which isn't strictly necessary as the `using` clause will ensure that this is done anyway; however, you should ensure that any resources such as this are released as soon as possible – you may have more code in the rest of the block and there's no point locking a resource unnecessarily.

In addition, if an exception is raised within the `using` block, the `IDisposable.Dispose` method will be called on the resource guarded by the `using` clause, which in this case will ensure that the database connection is always closed. This produces easier to read code than having to ensure you close a connection within an exception clause.

One last word – if you are writing a class that wraps a resource, whatever that resource may be, always implement the `IDisposable` interface to close the resource. That way anyone coding with your class can utilize the `using()` statement and guarantee that the resource will be cleared up.

# Transactions

Often when there is more than one update to be made to the database, these updates must be performed within the scope of a transaction. A transaction in ADO.NET is begun by calling one of the `BeginTransaction()` methods on the database connection object. These methods return an object that implements the `IDbTransaction` interface, defined within `System.Data`.

The following sequence of code initiates a transaction on a SQL Server connection:

```
string source = "server=(local)\\NetSDK;" +
                "uid=QSUser;pwd=QSPassword;" +
                "database=Northwind";
SqlConnection conn = new SqlConnection(source);
conn.Open();
SqlTransaction tx = conn.BeginTransaction();

// Execute some commands, then commit the transaction

tx.Commit();
conn.Close();
```

When you begin a transaction, you can choose the isolation level for commands executed within that transaction. The level determines how isolated your transaction is from others occurring on the database server. Certain database engines may support fewer than the four presented here. The options are as follows:

| Isolation Level | Description |
|---|---|
| `ReadCommitted` | The default for SQL Server. This level ensures that data written by one transaction will only be accessible in a second transaction after the first commits. |
| `ReadUncommitted` | This permits your transaction to read data within the database, even data that has not yet been committed by another transaction. As an example, if two users were accessing the same database, and the first inserted some data without concluding their transaction (by means of a `Commit` or `Rollback`), then the second user with their isolation level set to `ReadUncommitted` could read the data. |
| `RepeatableRead` | This level, which extends the `ReadCommitted` level, ensures that if the same statement is issued within the transaction, regardless of other potential updates made to the database, the same data will always be returned. This level does require extra locks to be held on the data, which could adversely affect performance.<br><br>This level guarantees that, for each row in the initial query, no changes can be made to that data. It does however permit "phantom" rows to show up – these are completely new rows that another transaction may have inserted while your transaction is running. |

| Isolation Level | Description |
|---|---|
| `Serializable` | This is the most "exclusive" transaction level, which in effect serializes access to data within the database. With this isolation level, phantom rows can never show up, so a SQL statement issued within a serializable transaction will always retrieve the same data.<br><br>The negative performance impact of a `Serializable` transaction should not be underestimated – if you don't absolutely need to use this level of isolation, it is advisable to stay away from it. |

The SQL Server default isolation level, `ReadCommitted`, is a good compromise between data coherence and data availability, as fewer locks are required on data than in `RepeatableRead` or `Serializable` modes. However, there are situations where the isolation level should be increased, and so within .NET you can simply begin a transaction with a different level from the default. There are no hard and fast rules as to which levels to pick – that comes with experience.

> *One last word on transactions – if you are currently using a database that does not support transactions, it is well worth changing to a database that does!*

# Commands

I briefly touched on the idea of issuing commands against a database in the *Using Database Connections* section. A command is, in its simplest form, a string of text containing SQL statements that is to be issued to the database. A command could also be a stored procedure, or the name of a table that will return all columns and all rows from that table (in other words, a `SELECT *`-style clause).

A command can be constructed by passing the SQL clause as a parameter to the constructor of the `SqlCommand` class, as shown below:

```
string source = "server=(local)\\NetSDK;" +
                "uid=QSUser;pwd=QSPassword;" +
                "database=Northwind";
string select = "SELECT ContactName,CompanyName FROM Customers";
SqlConnection conn = new SqlConnection(source);
conn.Open();
SqlCommand cmd = new SqlCommand(select, conn);
```

The `SqlCommand` and `OleDbCommand` classes have a property called `CommandType`, which is used to define whether the command is a SQL clause, a call to a stored procedure, or a full table statement (which simply selects all columns and rows from a given table). The following table summarizes the `CommandType` enumeration:

| CommandType | Example |
|---|---|
| Text<br>(default) | `String select = "SELECT ContactName FROM Customers";`<br>`SqlCommand cmd = new SqlCommand(select , conn);` |
| StoredProcedure | `SqlCommand cmd = new SqlCommand("CustOrderHist", conn);`<br>`cmd.CommandType = CommandType.StoredProcedure;`<br>`cmd.Parameters.Add("@CustomerID", "QUICK");` |
| TableDirect | `OleDbCommand cmd = new OleDbCommand("Categories", conn);`<br>`cmd.CommandType = CommandType.TableDirect;` |

When executing a stored procedure, it may be necessary to pass parameters to that procedure. The example above sets the `@CustomerID` parameter directly, although there are other ways of setting the parameter value, which we will look at later in the chapter.

**Note: The `TableDirect` command type is only valid for the `OleDb` provider – an exception is thrown by the `Sql` provider if you attempt to use this command type with it.**

# Executing Commands

Once you have the command defined, you need to execute it. There are a number of ways to issue the statement, depending on what you expect to be returned (if anything) from that command. The `SqlCommand` and `OleDbCommand` classes provide the following execute methods:

- `ExecuteNonQuery()` – Execute the command but do not return any output
- `ExecuteReader()` – Execute the command and return a typed `IDataReader`
- `ExecuteScalar()` – Execute the command and return a single value

In addition to the above methods, the `SqlCommand` class also exposes the following method

- `ExecuteXmlReader()` – Execute the command, and return an `XmlReader` object, which can be used to traverse the XML fragment returned from the database.

The example code in this section can be found in the `Chapter 09\01_ExecutingCommands` subdirectory of the code download.

## *ExecuteNonQuery()*

This method is commonly used for `UPDATE`, `INSERT`, or `DELETE` statements, where the only returned value is the number of records affected. This method can, however, return results if you call a stored procedure that has output parameters.

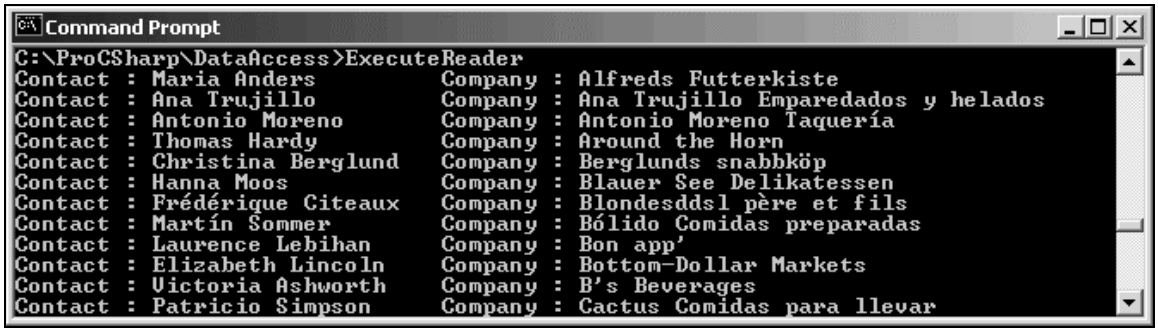
```
using System;
using System.Data.SqlClient;
public class ExecuteNonQueryExample
{
   public static void Main(string[] args)
   {
      string source = "server=(local)\\NetSDK;" +
                      "uid=QSUser;pwd=QSPassword;" +
                      "database=Northwind";
      string select = "UPDATE Customers " +
                      "SET ContactName = 'Bob' " +
                      "WHERE ContactName = 'Bill'";
      SqlConnection  conn = new SqlConnection(source);
      conn.Open();
      SqlCommand cmd = new SqlCommand(select, conn);
      int rowsReturned = cmd.ExecuteNonQuery();
      Console.WriteLine("{0} rows returned.", rowsReturned);
      conn.Close();
   }
}
```

`ExecuteNonQuery()` returns the number of rows affected by the command as an `int`.

## *ExecuteReader()*

This method executes the command and returns a `SqlDataReader` or `OleDbDataReader` object, depending on the provider in use. The object returned can be used to iterate through the record(s) returned, as shown in the following code:

```
using System;
using System.Data.SqlClient;
public class ExecuteReaderExample
{
   public static void Main(string[] args)
   {
      string source = "server=(local)\\NetSDK;" +
                      "uid=QSUser;pwd=QSPassword;" +
                      "database=Northwind";
      string select = "SELECT ContactName,CompanyName FROM Customers";
      SqlConnection conn = new SqlConnection(source);
      conn.Open();
      SqlCommand cmd = new SqlCommand(select, conn);
      SqlDataReader reader = cmd.ExecuteReader();
      while(reader.Read())
      {
         Console.WriteLine("Contact : {0,-20} Company : {1}" ,
                           reader[0] , reader[1]);
      }
   }
}
```




The `SqlDataReader` and `OleDbDataReader` objects will be discussed later in this chapter.

## ExecuteScalar()

On many occasions it is necessary to return a single result from a SQL statement, such as the count of records in a given table, or the current date/time on the server. The `ExecuteScalar` method can be used in such situations:

```
using System;
using System.Data.SqlClient;
public class ExecuteScalarExample
{
   public static void Main(string[] args)
   {
      string source = "server=(local)\\NetSDK;" +
                      "uid=QSUser;pwd=QSPassword;" +
                      "database=Northwind";
      string select = "SELECT COUNT(*) FROM Customers";
      SqlConnection conn = new SqlConnection(source);
      conn.Open();
      SqlCommand cmd = new SqlCommand(select, conn);
      object o = cmd.ExecuteScalar();
      Console.WriteLine ( o ) ;
   }
}
```

The method returns an object, which you can cast into the appropriate type if required.

## ExecuteXmlReader() (SqlClient Provider Only)

As its name implies, this method will execute the command and return an `XmlReader` object to the caller. SQL Server permits a SQL `SELECT` statement to be extended with a `FOR XML` clause. This clause can take one of three options:

- `FOR XML AUTO` – builds a tree based on the tables in the `FROM` clause
- `FOR XML RAW` – result set rows are mapped to elements, with columns mapped to attributes
- `FOR XML EXPLICIT` –you must specify the shape of the XML tree to be returned

*Professional SQL Server 2000 XML* (Wrox Press, ISBN 1-861005-46-6) includes a complete description of these options. For this example I shall use `AUTO`:

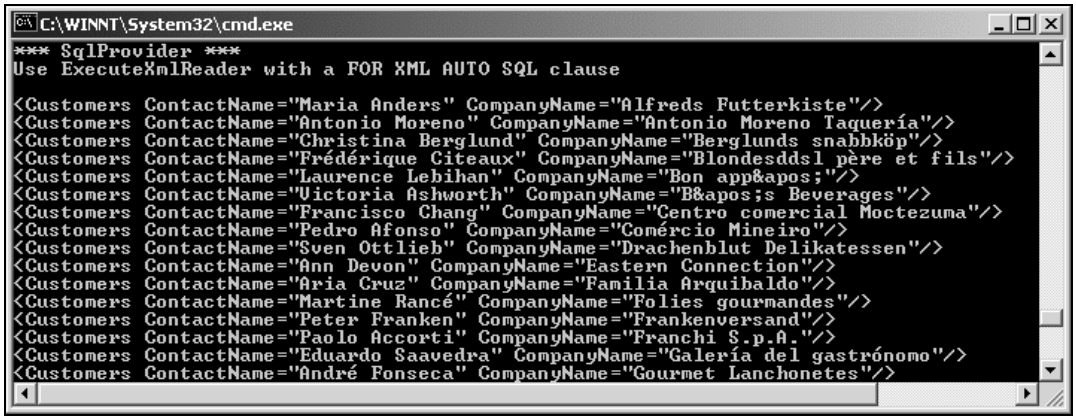
```
using System;
using System.Data.SqlClient;
using System.Xml;
public class ExecuteXmlReaderExample
{
   public static void Main(string[] args)
   {

      string source = "server=(local)\\NetSDK;" +
                      "uid=QSUser;pwd=QSPassword;" +
                      "database=Northwind";

      string select = "SELECT ContactName,CompanyName " +
                      "FROM Customers FOR XML AUTO";
      SqlConnection conn = new SqlConnection(source);
      conn.Open();
      SqlCommand cmd = new SqlCommand(select, conn);
      XmlReader xr = cmd.ExecuteXmlReader();
      while(xr.Read() )
      {
         Console.WriteLine(xr.ReadOuterXml());
      }
      conn.Close();
   }
}
```

Note that we have to import the `System.Xml` namespace in order to output the returned XML. This namespace and further XML capabilities of the .NET Framework are explored in more detail in Chapter 11.

Here we include the `FOR XML AUTO` clause in the SQL statement, then call the `ExecuteXmlReader()` method. A screenshot of the possible output from this code is shown below:



In the SQL clause, we specified `FROM Customers`, so an element of type `Customers` is shown in the output. To this are added attributes, one for each column selected from the database. This builds up an XML fragment for each row selected from the database.

# Calling Stored Procedures

Calling a stored procedure with a command object is just a matter of defining the name of the stored procedure, adding a parameter's definition for each parameter of the procedure, then executing the command with one of the methods presented in the previous section.

In order to make the examples in this section more useful, I have defined a set of stored procedures that can be used to insert, update, and delete records from the `Region` table in the `Northwind` example database. I have chosen this table despite its small size, as it can be used to define examples for each of the types of stored procedures you will commonly write.

## *Calling a Stored Procedure that Returns Nothing*

The simplest example of calling a stored procedure is one that returns nothing to the caller. There are two such procedures defined below, one for updating a pre-existing `Region` record, and the other for deleting a given `Region` record.

### *Record Update*

Updating a `Region` record is fairly trivial, as there is only one column that can be modified (assuming primary keys cannot be updated). You can type these examples directly into the SQL Server Query Analyzer, or run the `StoredProcs.sql` file in the `Chapter 09\02_StoredProcs` subdirectory, which will install each of the stored procedures in this section:

```
CREATE PROCEDURE RegionUpdate (@RegionID INTEGER,
                               @RegionDescription NCHAR(50)) AS
   SET NOCOUNT OFF
   UPDATE Region
      SET RegionDescription = @RegionDescription
      WHERE RegionID = @RegionID
GO
```

An update command on a more real-world table might need to re-select and return the updated record in its entirety. This stored procedure takes two input parameters (`@RegionID` and `@RegionDescription`), and issues an `UPDATE` statement against the database.

To run this stored procedure from within .NET code, you need to define a SQL command and execute it:

```
SqlCommand aCommand = new SqlCommand("RegionUpdate", conn);

aCommand.CommandType = CommandType.StoredProcedure;
aCommand.Parameters.Add(new SqlParameter ("@RegionID",
                                          SqlDbType.Int,
                                          0,
                                         "RegionID"));
```

```
aCommand.Parameters.Add(new SqlParameter("@RegionDescription",
                                         SqlDbType.NChar,
                                         50,
                                         "RegionDescription"));
aCommand.UpdatedRowSource = UpdateRowSource.None;
```

This code creates a new `SqlCommand` object named `aCommand`, and defines it as a stored procedure. We then add each parameter in turn, and finally set the expected output from the stored procedure to one of the values in the `UpdateRowSource` enumeration, which is discussed later in this chapter.

The stored procedure takes two parameters: the unique primary key of the `Region` record being updated, and the new description to be given to this record.

Once the command has been created, it can be executed by issuing the following commands:

```
aCommand.Parameters[0].Value = 999;
aCommand.Parameters[1].Value = "South Western England";
aCommand.ExecuteNonQuery();
```

Here we are setting the value of the parameters, then executing the stored procedure. As the procedure returns nothing, `ExecuteNonQuery()` will suffice.

Command parameters may be set by ordinal as shown above, or set by name.

### *Record Deletion*

The next stored procedure required is one that can be used to delete a `Region` record from the database:

```
CREATE PROCEDURE RegionDelete (@RegionID INTEGER) AS
   SET NOCOUNT OFF
   DELETE FROM Region
   WHERE       RegionID = @RegionID
GO
```

This procedure only requires the primary key value of the record. The code uses a `SqlCommand` object to call this stored procedure as follows:

```
SqlCommand aCommand = new SqlCommand("RegionDelete" , conn);
aCommand.CommandType = CommandType.StoredProcedure;
aCommand.Parameters.Add(new SqlParameter("@RegionID" , SqlDbType.Int , 0 ,
                                         "RegionID"));
aCommand.UpdatedRowSource = UpdateRowSource.None;
```

This command only accepts a single parameter as shown in the following code, which will execute the `RegionDelete` stored procedure; here we see an example of setting the parameter by name:

```
aCommand.Parameters["@RegionID"].Value= 999;
aCommand.ExecuteNonQuery();
```

## Calling a Stored Procedure that Returns Output Parameters

Both of the previous examples execute stored procedures that return nothing. If a stored procedure includes output parameters, then these need to be defined within the .NET client so that they can be filled when the procedure returns.

The following example shows how to insert a record into the database, and return the primary key of that record to the caller.

### Record Insertion

The `Region` table only consists of a primary key (`RegionID`) and description field (`RegionDescription`). To insert a record, this numeric primary key needs to be generated, then a new row inserted into the database. I have chosen to simplify the primary key generation in this example by creating one within the stored procedure. The method used is exceedingly crude, which is why I have devoted a section to key generation later in the chapter. For now this primitive example will suffice:

```
CREATE PROCEDURE RegionInsert(@RegionDescription NCHAR(50),
                              @RegionID INTEGER OUTPUT)AS
   SET NOCOUNT OFF
   SELECT @RegionID = MAX(RegionID)+ 1
   FROM Region
   INSERT INTO Region(RegionID, RegionDescription)
   VALUES(@RegionID, @RegionDescription)
GO
```

The insert procedure creates a new `Region` record. As the primary key value is generated by the database itself, this value is returned as an output parameter from the procedure (**`@RegionID`**). This is sufficient for this simple example, but for a more complex table (especially one with default values), it is more common not to utilize output parameters, and instead select the entire inserted row and return this to the caller. The .NET classes can cope with either scenario.

```
SqlCommand  aCommand = new SqlCommand("RegionInsert" , conn);
aCommand.CommandType = CommandType.StoredProcedure;
aCommand.Parameters.Add(new SqlParameter("@RegionDescription" ,
                                         SqlDbType.NChar ,
                                         50 ,
                                         "RegionDescription"));
aCommand.Parameters.Add(new SqlParameter("@RegionID" ,
                                         SqlDbType.Int,
                                         0 ,
                                         ParameterDirection.Output ,
                                         false ,
                                         0 ,
                                         0 ,
                                         "RegionID" ,
                                         DataRowVersion.Default ,
                                         null));
aCommand.UpdatedRowSource = UpdateRowSource.OutputParameters;
```

Here, the definition of the parameters is much more complex. The second parameter, `@RegionID`, is defined to include its parameter direction, which in this example is `Output`. In addition to this flag, on the last line of the code, we utilize the `UpdateRowSource` enumeration to indicate that we expect to return data from this stored procedure via output parameters. This flag is mainly used when issuing stored procedure calls from a `DataTable` (covered later in the chapter).

Calling this stored procedure is similar to the previous examples, except in this instance we need to read the output parameter after executing the procedure:

```
aCommand.Parameters["@RegionDescription"].Value = "South West";
aCommand.ExecuteNonQuery();
int newRegionID = (int) aCommand.Parameters["@RegionID"].Value;
```

After executing the command, we read the value of the `@RegionID` parameter and cast this to an integer.

You may be wondering what to do if the stored procedure you call returns output parameters and a set of rows. In this instance, define the parameters as appropriate, and rather than calling `ExecuteNonQuery()`, call one of the other methods (such as `ExecuteReader()`) that will permit you to traverse any record(s) returned.

# Quick Data Access: The Data Reader

A data reader is the simplest and fastest way of selecting some data from a data source, but also the least capable. You cannot directly instantiate a data reader object – an instance is returned from a `SqlCommand` or `OleDbCommand` object having called the `ExecuteReader()` method – from a `SqlCommand` object, a `SqlDataReader` object is returned, and from the `OleDbCommand` object, a `OleDbDataReader` object is returned.

The following code demonstrates how to select data from the `Customers` table in the `Northwind` database. The example connects to the database, selects a number of records, loops through these selected records and outputs them to the console.

This example utilizes the OLE DB provider as a brief respite from the SQL provider. In most cases the classes have a one-to-one correspondence with their `SqlClient` cousins, so for instance there is the `OleDbConnection` object, which is similar to the `SqlConnection` object used in the previous examples.

To execute commands against an OLE DB data source, the `OleDbCommand` class is used. The following code shows an example of executing a simple SQL statement and reading the records by returning an `OleDbDataReader` object.

The code for this example can be found in the `Chapter 09\03_DataReader` directory.

Note the second `using` directive below that makes available the `OleDb` classes:

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

All the data providers currently available are shipped within the same DLL, so it is only necessary to reference the `System.Data.dll` assembly to import all classes used in this section:

```
public class DataReaderExample
{
   public static void Main(string[] args)
   {
      string source = "Provider=SQLOLEDB;" +
                      "server=(local)\\NetSDK;" +
                      "uid=QSUser;pwd=QSPassword;" +
                      "database=northwind";
      string select = "SELECT ContactName,CompanyName FROM Customers";
      OleDbConnection conn = new OleDbConnection(source);
      conn.Open();
      OleDbCommand cmd = new OleDbCommand(select , conn);
      OleDbDataReader aReader = cmd.ExecuteReader();
      while(aReader.Read())
         Console.WriteLine("'{0}' from {1}" ,
                           aReader.GetString(0) , aReader.GetString(1));
      aReader.Close();
      conn.Close();
   }
}
```

The preceding code includes many familiar aspects of C# covered in other chapters. To compile the example, issue the following command:

```
csc /t:exe /debug+ DataReaderExample.cs /r:System.Data.dll
```

The following code from the example above creates a new OLE DB .NET database connection, based on the source connection string:

```
OleDbConnection conn = new OleDbConnection(source);
conn.Open();
OleDbCommand cmd = new OleDbCommand(select, conn);
```

The third line creates a new `OleDbCommand` object, based on a particular `SELECT` statement, and the database connection to be used when the command is executed. When you have a valid command, you then need to execute it, which returns an initialized `OleDbDataReader`:

```
OleDbDataReader aReader = cmd.ExecuteReader();
```

An `OleDbDataReader` is a forward-only "connected" cursor. In other words, you can only traverse through the records returned in one direction, and the database connection used is kept open until the data reader has been closed.

**An `OleDbDataReader` keeps the database connection open until explicitly closed.**

The `OleDbDataReader` class cannot be directly instantiated – it is always returned by a call to the `ExecuteReader()` method of the `OleDbCommand` class. Once you have an open data reader, there are various ways to access the data contained within the reader.

When the `OleDbDataReader` object is closed (via an explicit call to `Close()`, or the object being garbage collected), the underlying connection may also be closed, depending on which of the `ExecuteReader()` methods is called. If you call `ExecuteReader()` and pass `CommandBehavior.CloseConnection`, you can force the connection to be closed when the reader
is closed.

The `OleDbDataReader` class has an indexer that permits access (although not type-safe access) to any field using the familiar array style syntax:

```
object o = aReader[0];
object o = aReader["CategoryID"];
```

Assuming that the `CategoryID` field was the first in the `SELECT` statement used to populate the reader, these two lines are functionally equivalent, although the second is slower than the first – I wrote a simple test application that performed a million iterations of accessing the same column from an open data reader, just to get some numbers that were big enough to read. I know – you probably don't read the same column a million times in a tight loop, but every (micro) second counts, and you might as well write code that is as close to optimal as possible.

Just for interest, the numeric indexer took on average 0.09 seconds for the million accesses, and the textual one 0.63 seconds. The reason for this difference is that the textual method looks up the column number internally from the schema and then accesses it using its ordinal. If you know this information beforehand you can do a better job of accessing the data.

So should you use the numeric indexer? Maybe, but there is a better way.

In addition to the indexers presented above, the `OleDbDataReader` has a set of type-safe methods that can be used to read columns. These are fairly self-explanatory, and all begin with `Get`. There are methods to read most types of data, such as `GetInt32`, `GetFloat`, `GetGuid`, and so on.

My million iterations using `GetInt32` took 0.06 seconds. The overhead in the numeric indexer is incurred while getting the data type, calling the same code as `GetInt32`, then boxing (and in this instance unboxing) an integer. So, if you know the schema beforehand, are willing to use cryptic numbers instead of column names, and you can be bothered to use a type-safe function for each and every column access, you stand to gain somewhere in the region of a ten fold speed increase over using a textual column name (when selecting those million copies of the same column).

Needless to say, there is a tradeoff between maintainability and speed. If you must use numeric indexers, define constants within class scope for each of the columns that you will be accessing.

The code above can be used to select data from any OLE DB database; however, there are a number of SQL Server-specific classes that can be used with the obvious portability tradeoff.

The following example is the same as the above, except in this instance I have replaced the OLE DB provider and all references to OLE DB classes with their SQL counterparts. The changes in the code from the previous example have been highlighted. The example is in the `04_DataReaderSql` directory:

```
using System;
using System.Data.SqlClient;
public class DataReaderSql
{
   public static int Main(string[] args)
   {
      string source = "server=(local)\\NetSDK;" +
                      "uid=QSUser;pwd=QSPassword;" +
                      "database=northwind";
      string select = "SELECT ContactName,CompanyName FROM Customers";
      SqlConnection conn = new SqlConnection(source);
      conn.Open();
      SqlCommand cmd = new SqlCommand(select , conn);
      SqlDataReader aReader = cmd.ExecuteReader();
      while(aReader.Read())
         Console.WriteLine("'{0}' from {1}" , aReader.GetString(0) ,
                           aReader.GetString(1));
      aReader.Close();
      conn.Close();
      return 0;
   }
}
```

Notice the difference? If you're typing this in then do a global replace on `OleDb` with `Sql`, change the data source string and recompile. It's that easy!

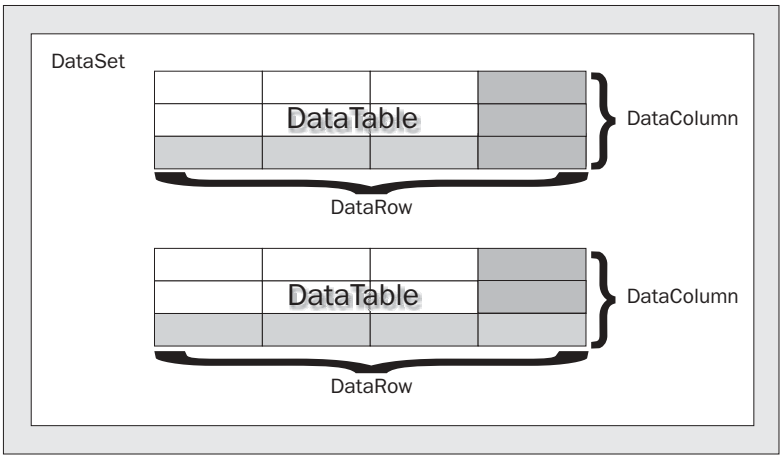
I ran the same performance tests on the indexers for the SQL provider, and this time the numeric indexers were both exactly the same at 0.13 seconds for the million accesses, and the string-based indexer ran at about 0.65 seconds. You would expect the native SQL Server provider to be faster than going through `OleDb`, which up until I tested this section under the release version of .NET it was. I'm reasonably sure that this is an anomaly due to the simplistic test approach I am using (selecting the same value 1,000,000 times), and would expect a real-world test to show better performance from the managed SQL provider.

If you are interested in running the code on your own computer to see what performance is like, see the `05_IndexerTestingOleDb` and `06_IndexerTestingSql` examples included in the code download.

# Managing Data and Relationships: The DataSet

The `DataSet` class has been designed as an offline container of data. It has no notion of database connections. In fact, the data held within a `DataSet` doesn't necessarily need to have come from a database – it could just as easily be records from a CSV file, or points read from a measuring device.

A `DataSet` consists of a set of data tables, each of which will have a set of data columns and data rows. In addition to defining the data, you can also define *links* between tables within the `DataSet`. One common scenario would be when defining a parent-child relationship (commonly known as master/detail). One record in a table (say `Order`) links to many records in another table (say `Order_Details`). This relationship can be defined and navigated within the `DataSet`.



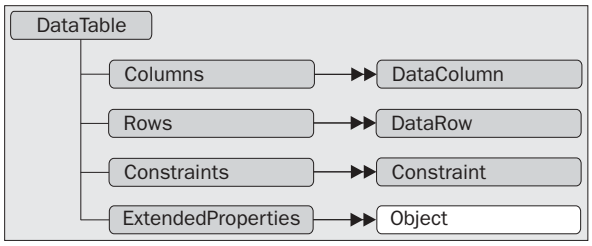

The following sections describe the classes that are used with a `DataSet`.

# Data Tables

A data table is very similar to a physical database table – it consists of a set of columns with particular properties, and may contain zero or more rows of data. A data table may also define a primary key, which can be one or more columns, and may also contain constraints on columns. The generic term for this information used throughout the rest of the chapter is **schema**.

There are several ways to define the schema for a particular data table (and indeed the `DataSet` as a whole). These are discussed after we introduce data columns and data rows.

The following diagram shows some of the objects that are accessible through the data table:




A `DataTable` object (and also a `DataColumn`) can have an arbitrary number of extended properties associated with it. This collection can be populated with any user-defined information pertaining to the object. For example, a given column might have an input mask used to validate the contents of that column – the typical example would be the US social security number. Extended properties are especially useful when the data is constructed within a middle tier and returned to the client for some processing. You could, for example, store validation criteria (such as `min` and `max`) for numeric columns.

When a data table has been populated, either by selecting data from a database, reading data from a file, or manually populating within code, the `Rows` collection will contain this retrieved data.

The `Columns` collection contains `DataColumn` instances that have been added to this table. These define the schema of the data, such as the data type, nullability, default values, and so on. The `Constraints` collection can be populated with either unique or primary key constraints.

One example of where the schema information for a data table is used is when displaying that data in a `DataGrid` (which we'll discuss at length in the next chapter). The `DataGrid` control uses properties such as the data type of the column to decide what control to use for that column. A bit field within the database will be displayed as a checkbox within the `DataGrid`. If a column is defined within the database schema as `NOT NULL`, then this fact will be stored within the `DataColumn` so that it can be tested when the user attempts to move off a row.

## Data Columns

A `DataColumn` object defines properties of a column within the `DataTable`, such as the data type of that column, whether the column is read only, and various other facts. A column can be created in code, or can be automatically generated by the runtime.

When creating a column, it is also useful to give it a name; otherwise the runtime will generate a name for you in the form `Column`*n* where *n* is an incrementing number.

The data type of the column can be set either by supplying it in the constructor, or by setting the `DataType` property. Once you have loaded data into a data table you cannot alter the type of a column – you'll just receive an `ArgumentException`.

Data columns can be created to hold the following .NET Framework data types:

| | | | |
|---|---|---|---|
| `Boolean` | `Decimal` | `Int64` | `TimeSpan` |
| `Byte` | `Double` | `Sbyte` | `UInt16` |
| `Char` | `Int16` | `Single` | `UInt32` |
| `DateTime` | `Int32` | `String` | `UInt64` |

Once created, the next thing to do with a `DataColumn` object is to set up other properties, such as the nullability of the column or the default value. The following code fragment shows a few of the more common options to set on a `DataColumn`:

```
DataColumn customerID = new DataColumn("CustomerID" , typeof(int));
customerID.AllowDBNull = false;
customerID.ReadOnly = false;
customerID.AutoIncrement = true;
customerID.AutoIncrementSeed = 1000;
DataColumn name = new DataColumn("Name" , typeof(string));
name.AllowDBNull = false;
name.Unique = true;
```

The following properties can be set on a `DataColumn`:

| Property | Description |
|---|---|
| `AllowDBNull` | If `true`, permits the column to be set to `DBNull`. |
| `AutoIncrement` | Defines that this column value is automatically generated as an incrementing number. |
| `AutoIncrementSeed` | The initial seed value for an `AutoIncrement` column. |
| `AutoIncrementStep` | Defines the step between automatically generated column values, with a default of one. |
| `Caption` | Can be used for displaying the name of the column on screen. |
| `ColumnMapping` | Defines how a column is mapped into XML when a `DataSet` is saved by calling `DataSet.WriteXml`. |
| `ColumnName` | The name of the column. This is auto-generated by the runtime if not set in the constructor. |
| `DataType` | The `System.Type` value of the column. |
| `DefaultValue` | Can define a default value for a column. |
| `Expression` | This property defines the expression to be used in a computed column. |

## *Data Rows*

This class makes up the other part of the `DataTable` class. The columns within a data table are defined in terms of the `DataColumn` class. The actual data within the table is accessed using the `DataRow` object. The following example shows how to access rows within a data table. The code for this example is available in the `07_SimpleDatasetSql` directory. First, the connection details:

```
string source = "server=(local)\\NetSDK;" +
                "uid=QSUser;pwd=QSPassword;" +
                "database=northwind";
string select = "SELECT ContactName,CompanyName FROM Customers";
SqlConnection  conn = new SqlConnection(source);
```

The following code introduces the `SqlDataAdapter` class, which is used to place data into a `DataSet`. The `SqlDataAdapter` will issue the SQL clause, and fill a table in the `DataSet` called `Customers` with the output of this following query. We'll be discussing the data adapter class further in the *Populating a DataSet* section.

```
SqlDataAdapter da = new SqlDataAdapter(select, conn);
DataSet ds = new DataSet();
da.Fill(ds , "Customers");
```

In the code below, you may notice the use of the `DataRow` indexer to access values from within that row. The value for a given column can be retrieved using one of the several overloaded indexers. These permit you to retrieve a value knowing the column number, name, or `DataColumn`:

```
foreach(DataRow row in ds.Tables["Customers"].Rows)
   Console.WriteLine("'{0}' from {1}" , row[0] ,row[1]);
```

One of the most appealing aspects of a `DataRow` is that it is versioned. This permits you to receive various values for a given column in a particular row. The versions are described in the following table:

| **`DataRowVersion` Value** | **Description** |
|---|---|
| `Current` | The value existing at present within the column. If no edit has occurred, this will be the same as the original value. If an edit (or edits) have occurred, the value will be the last valid value entered. |
| `Default` | The default value (in other words, any default set up for the column). |
| `Original` | The value of the column when originally selected from the database. If the `DataRow`'s `AcceptChanges` method is called, then this value will update to be the `Current` value. |
| `Proposed` | When changes are in progress for a row, it is possible to retrieve this modified value. If you call `BeginEdit()` on the row and make changes, each column will have a proposed value until either `EndEdit()` or `CancelEdit()` is called. |

The version of a given column could be used in many ways. One example is when updating rows within the database, in which instance it is common to issue an SQL statement such as the following:

```
UPDATE Products
SET    Name = Column.Current
WHERE  ProductID = xxx
AND    Name = Column.Original;
```

Obviously this code would never compile, but it shows one use for original and current values of a column within a row.

To retrieve a versioned value from the `DataRow`, use one of the indexer methods that accept a `DataRowVersion` value as a parameter. The following code snippet shows how to obtain all values of each column in a `DataTable`:

```
foreach (DataRow row in ds.Tables["Customers"].Rows )
{
  foreach ( DataColumn dc in ds.Tables["Customers"].Columns )
  {
    Console.WriteLine ("{0} Current  = {1}" , dc.ColumnName ,
                                              row[dc,DataRowVersion.Current]);
    Console.WriteLine ("    Default  = {0}" , row[dc,DataRowVersion.Default]);
    Console.WriteLine ("    Original = {0}" , row[dc,DataRowVersion.Original]);
  }
}
```

The whole row has a state flag called `RowState`, which can be used to determine what operation is needed on the row when it is persisted back to the database. The `RowState` property is set to keep track of all the changes made to the `DataTable`, such as adding new rows, deleting existing rows, and changing columns within the table. When the data is reconciled with the database, the row state flag is used to determine what SQL operations should occur. These flags are defined by the `DataRowState` enumeration:

| `DataRowState` Value | Description |
|---|---|
| `Added` | The row has been newly added to a `DataTable`'s `Rows` collection. All rows created on the client are set to this value, and will ultimately issue SQL `INSERT` statements when reconciled with the database. |
| `Deleted` | This indicates that the row has been marked as deleted from the `DataTable` by means of the `DataRow.Delete()` method. The row still exists within the `DataTable`, but will not normally be viewable on screen (unless a `DataView` has been explicitly set up). `DataView`s will be discussed in the next chapter. Rows marked as deleted in the `DataTable` will be deleted from the database when reconciled. |
| `Detached` | A row is in this state immediately after it is created, and can also be returned to this state by calling `DataRow.Remove()`. A detached row is not considered to be part of any data table, and as such no SQL for rows in this state will be issued. |
| `Modified` | A row will be `Modified` if the value in any column has been changed. |
| `Unchanged` | The row has not been changed since the last call to `AcceptChanges()`. |

The state of the row depends also on what methods have been called on the row. The `AcceptChanges()` method is generally called after successfully updating the data source (that is, after persisting changes to the database).

The most common way to alter data in a `DataRow` is to use the indexer; however, if you have a number of changes to make you also need to consider the `BeginEdit()` and `EndEdit()` methods.

When an alteration is made to a column within a `DataRow`, the `ColumnChanging` event is raised on the row's `DataTable`. This permits you to override the `ProposedValue` property of the `DataColumnChangeEventArgs` class classes, and change it as required. This is one way of performing some data validation on column values. If you call `BeginEdit()` before making changes, the `ColumnChanging` event will not be raised. This permits you to make multiple changes and then call `EndEdit()` to persist these changes. If you wish to revert to the original values, call `CancelEdit()`.

A `DataRow` can be linked in some way to other rows of data. This permits the creation of navigable links between rows, which is common in master/detail scenarios. The `DataRow` contains a `GetChildRows()` method that will return an array of associated rows from another table in the same `DataSet` as the current row. These are discussed in the *Data Relationships* section later in this chapter.

## Schema Generation

There are three ways to create the schema for a `DataTable`. These are:

- Let the runtime do it for you
- Write code to create the table(s)
- Use the XML schema generator

### Runtime Schema Generation

The `DataRow` example shown earlier presented the following code for selecting data from a database and populating a `DataSet`:

```
SqlDataAdapter da = new SqlDataAdapter(select , conn);
DataSet ds = new DataSet();
da.Fill(ds , "Customers");
```

This is obviously easy to use, but it has a few drawbacks too. One example is that you have to make do with the column names selected from the database, which may be fine, but in certain instances you might want to rename a physical database column (say `PKID`) to something more user-friendly.

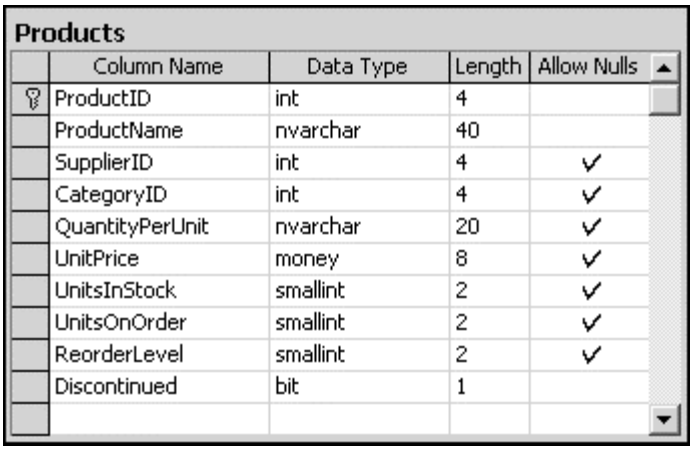
You could naturally rename columns within your SQL clause, as in `SELECT PID AS PersonID FROM PersonTable`; I would always recommend not renaming columns within SQL, as the only place a column really needs to have a "pretty" name is on screen.

Another potential problem with automated `DataTable`/`DataColumn` generation is that you have no control over the column types that the runtime chooses for your data. It does a fairly good job of deciding the correct data type for you, but as usual there are instances where you need more control. You might for example have defined an enumerated type for a given column, so as to simplify user code written against your class. If you accept the default column types that the runtime generates, the column will likely be an integer with a 32-bit range, as opposed to an `enum` with five options.

Lastly, and probably most problematic, is that when using automated table generation, you have no type-safe access to the data within the `DataTable` – you are at the mercy of indexers, which return instances of `object` rather than derived data types. If you like sprinkling your code with typecast expressions then skip the following sections.

### Hand-Coded Schema

Generating the code to create a `DataTable`, replete with associated `DataColumns` is fairly easy. The examples within this section will access the `Products` table from the `Northwind` database shown below. The code for this section is available in the `08_ManufacturedDataSet` example.



**Products**

| | Column Name | Data Type | Length | Allow Nulls |
|---|---|---|---|---|
| 🔑 | ProductID | int | 4 | |
| | ProductName | nvarchar | 40 | |
| | SupplierID | int | 4 | ✓ |
| | CategoryID | int | 4 | ✓ |
| | QuantityPerUnit | nvarchar | 20 | ✓ |
| | UnitPrice | money | 8 | ✓ |
| | UnitsInStock | smallint | 2 | ✓ |
| | UnitsOnOrder | smallint | 2 | ✓ |
| | ReorderLevel | smallint | 2 | ✓ |
| | Discontinued | bit | 1 | |
| | | | | |

The following code manufactures a `DataTable`, which corresponds to the above schema.

```
public static void ManufactureProductDataTable(DataSet ds)
{
   DataTable   products = new DataTable("Products");
   products.Columns.Add(new DataColumn("ProductID", typeof(int)));
   products.Columns.Add(new DataColumn("ProductName", typeof(string)));
   products.Columns.Add(new DataColumn("SupplierID", typeof(int)));
   products.Columns.Add(new DataColumn("CategoryID", typeof(int)));
   products.Columns.Add(new DataColumn("QuantityPerUnit", typeof(string)));
   products.Columns.Add(new DataColumn("UnitPrice", typeof(decimal)));
   products.Columns.Add(new DataColumn("UnitsInStock", typeof(short)));
   products.Columns.Add(new DataColumn("UnitsOnOrder", typeof(short)));
   products.Columns.Add(new DataColumn("ReorderLevel", typeof(short)));
   products.Columns.Add(new DataColumn("Discontinued", typeof(bool)));
   ds.Tables.Add(products);
}
```

You can alter the code in the `DataRow` example to utilize this newly generated table definition as follows:

```
string source = "server=localhost;" +
                "integrated security=sspi;" +
                "database=Northwind";
string select = "SELECT * FROM Products";
SqlConnection conn = new SqlConnection(source);
SqlDataAdapter cmd = new SqlDataAdapter(select, conn);
DataSet ds = new DataSet();
ManufactureProductDataTable(ds);
cmd.Fill(ds, "Products");
foreach(DataRow row in ds.Tables["Products"].Rows)
   Console.WriteLine("'{0}' from {1}", row[0], row[1]);
```

The `ManufactureProductDataTable()` method creates a new `DataTable`, adds each column in turn, and finally appends this to the list of tables within the `DataSet`. The `DataSet` has an indexer that takes the name of the table and returns that `DataTable` to the caller.

The above example is still not really type-safe, as I'm using indexers on columns to retrieve the data. What would be better is a class (or set of classes) derived from `DataSet`, `DataTable`, and `DataRow`, that define type-safe accessors for tables, rows, and columns. You can generate this code yourself – it's not particularly tedious and you end up with truly type-safe data access classes.

If you don't like the sound of generating these type-safe classes yourself then help is at hand. The .NET Framework includes support for using XML schemas to define a `DataSet`, `DataTable`, and the other classes that we have touched on in this section. The *XML Schemas* section later in the chapter details this method; but first, we will look at relationships and constraints within a `DataSet`.

# Data Relationships

When writing an application, it is often necessary to obtain and cache various tables of information. The `DataSet` class is the container for this information. With regular OLE DB it was necessary to provide a strange SQL dialect to enforce hierarchical data relationships, and the provider itself was not without its own subtle quirks.

The `DataSet` class on the other hand has been designed from the start to establish relationships between data tables with ease. For the code in this section I decided to hand-generate and populate two tables with data. So, if you haven't got SQL Server or the `NorthWind` database to hand, you can run this example anyway. The code is available in the `09_DataRelationships` directory:

```
DataSet ds = new DataSet("Relationships");
ds.Tables.Add(CreateBuildingTable());
ds.Tables.Add(CreateRoomTable());
ds.Relations.Add("Rooms",
                 ds.Tables["Building"].Columns["BuildingID"],
                 ds.Tables["Room"].Columns["BuildingID"]);
```

The tables simply contain a primary key and name field, with the `Room` table having `BuildingID` as a foreign key.


These tables were kept deliberately simple, as my fingers were wearing out at this point so I didn't want to add too many columns to either one.

I then added some default data to each table. Once that was done, I could then iterate through the buildings and rooms using the code below.

```
foreach(DataRow theBuilding in ds.Tables["Building"].Rows)
{
   DataRow[] children = theBuilding.GetChildRows("Rooms");
   int roomCount = children.Length;
   Console.WriteLine("Building {0} contains {1} room{2}",
                     theBuilding["Name"],
                     roomCount,
                     roomCount > 1 ? "s" : "");
   // Loop through the rooms
   foreach(DataRow theRoom in children)
      Console.WriteLine("Room: {0}", theRoom["Name"]);
}
```

The big difference between the `DataSet` and the old-style hierarchical `Recordset` object is in the way the relationship is presented. In a hierarchical `Recordset`, the relationship was presented as a pseudo-column within the row. This column itself was a `Recordset` that could be iterated through. Under ADO.NET, however, a relationship is traversed simply by calling the `GetChildRows()` method:

```
DataRow[] children = theBuilding.GetChildRows("Rooms");
```

This method has a number of forms, but the simple example shown above just uses the name of the relationship to traverse between parent and child rows. It returns an array of rows that can be updated as appropriate by using the indexers as shown in earlier examples.

What's more interesting with data relationships is that they can be traversed both ways. Not only can you go from a parent to the child rows, but you can also find a parent row (or rows) from a child record simply by using the `ParentRelations` property on the `DataTable` class. This property returns a `DataRelationCollection`, which can be indexed using the `[]` array syntax (for example, `ParentRelations["Rooms"]`), or as an alternative the `GetParentRows()` method can be called as shown below:

```
foreach(DataRow theRoom in ds.Tables["Room"].Rows)
{
   DataRow[] parents = theRoom.GetParentRows("Rooms");
   foreach(DataRow theBuilding in parents)
      Console.WriteLine("Room {0} is contained in building {1}",
                        theRoom["Name"],
                        theBuilding["Name"]);
}
```

There are two methods with various overrides available for retrieving the parent row(s) – `GetParentRows()` (which returns an array of zero or more rows), or `GetParentRow()` (which retrieves a single parent row given a relationship).

# Data Constraints

Changing the data type of columns created on the client is not the only thing a `DataTable` is good for. ADO.NET permits you to create a set of constraints on a column (or columns), which are then used to enforce rules within the data.

The runtime currently supports the following constraint types, embodied as classes in the `System.Data` namespace.

| Constraint | Description |
|---|---|
| `ForeignKeyConstraint` | Enforce a link between two `DataTable`s within a `DataSet` |
| `UniqueConstraint` | Ensure that entries in a given column are unique |

## *Setting a Primary Key*

As is common for a table in a relational database, you can supply a primary key, which can be based on one or more columns from the `DataTable`.

The code below creates a primary key for the `Products` table, whose schema we constructed by hand earlier, and can be found in the `08_ManufactureDataSet` folder.

Note that a primary key on a table is just one form of constraint. When a primary key is added to a `DataTable`, the runtime also generates a unique constraint over the key column(s). This is because there isn't actually a constraint type of `PrimaryKey` – a primary key is simply a unique constraint over one or more columns.

```
public static void ManufacturePrimaryKey(DataTable dt)
{
   DataColumn[] pk = new DataColumn[1];
   pk[0] = dt.Columns["ProductID"];
   dt.PrimaryKey = pk;
}
```

As a primary key may contain several columns, it is typed as an array of `DataColumn`s. A table's primary key can be set to those columns simply by assigning an array of columns to the property.

To check the constraints for a table, you can iterate through the `ConstraintCollection`. For the auto-generated constraint produced by the above code, the name of the constraint is `Constraint1`. That's not a very useful name, so to avoid this problem it is always best to create the constraint in code first, then define which column(s) make up the primary key, as we shall do now.

As a long time database programmer, I find named constraints much simpler to understand, as most databases produce cryptic names for constraints, rather than something simple and legible. The code below names the constraint before creating the primary key:

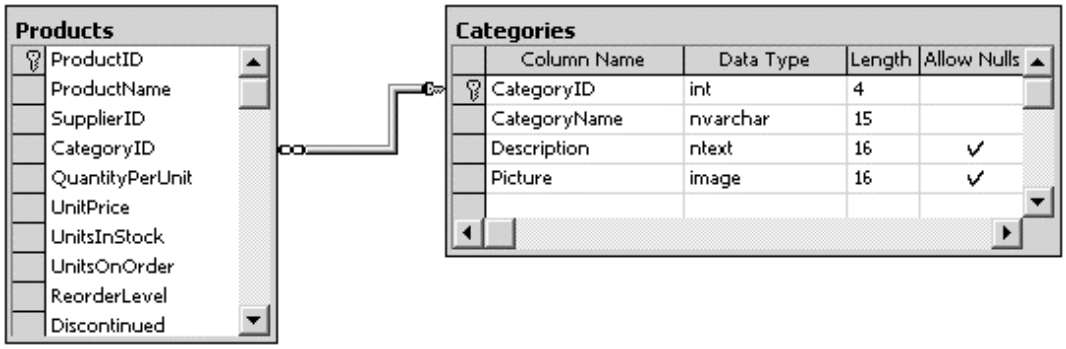
```
DataColumn[] pk = new DataColumn[1];
pk[0] = dt.Columns["ProductID"];
dt.Constraints.Add(new UniqueConstraint("PK_Products", pk[0]));
dt.PrimaryKey = pk;
```

Unique constraints can be applied to as many columns as you wish.

## Setting a Foreign Key

In addition to unique constraints, a `DataTable` may also contain foreign key constraints. These are primarily used to enforce master/detail relationships, but can also be used to replicate columns between tables if you set the constraint up correctly. A master/detail relationship is one where there is commonly one parent record (say an order) and many child records (order lines), linked by the primary key of the parent record.

A foreign key constraint can only operate over tables within the same `DataSet`, so the following example utilizes the `Categories` table from the `Northwind` database, and assigns a constraint between it and the `Products` table.




The first step is to generate a new data table for the `Categories` table. The `08_ManufactureDataSet` example includes this code:

```
DataTable categories = new DataTable("Categories");
categories.Columns.Add(new DataColumn("CategoryID", typeof(int)));
categories.Columns.Add(new DataColumn("CategoryName", typeof(string)));
categories.Columns.Add(new DataColumn("Description", typeof(string)));
categories.Constraints.Add(new UniqueConstraint("PK_Categories",
                                categories.Columns["CategoryID"]));
categories.PrimaryKey = new DataColumn[1]
                               {categories.Columns["CategoryID"]};
```

The last line of the above code creates the primary key for the `Categories` table. The primary key in this instance is a single column; however, it is possible to generate a key over multiple columns using the array syntax shown.

Then I need to create the constraint between the two tables:

```
DataColumn parent = ds.Tables["Categories"].Columns["CategoryID"];
DataColumn child = ds.Tables["Products"].Columns["CategoryID"];
ForeignKeyConstraint fk =
   new ForeignKeyConstraint("FK_Product_CategoryID", parent, child);
fk.UpdateRule = Rule.Cascade;
fk.DeleteRule = Rule.SetNull;
ds.Tables["Products"].Constraints.Add(fk);
```

This constraint applies to the link between `Categories.CategoryID` and `Products.CategoryID`. There are four different constructors for `ForeignKeyConstraint`, but again I would suggest using those that permit you to name the constraint.

### *Setting Update and Delete Constaints*

In addition to defining the fact that there is some type of constraint between parent and child tables, you can define what should happen when a column in the constraint is updated.

The above example sets the update rule and the delete rule. These rules are used when an action occurs to a column (or row) within the parent table, and the rule is used to decide what should happen to row(s) within the child table that could be affected. There are four different rules that can be applied through the `Rule` enumeration:

- `Cascade` – If the parent key was updated then copy the new key value to all child records. If the parent record was deleted, delete the child records also. This is the default option.
- `None` – No action whatsoever. This option will leave orphaned rows within the child data table.
- `SetDefault` – Each child record affected has the foreign key column(s) set to their default value, if one has been defined.
- `SetNull` – All child rows have the key column(s) set to `DBNull`. (Following on from the naming convention that Microsoft uses, this should really be `SetDBNull`).

> **Constraints are only enforced within a `DataSet` if the `EnforceConstraints` property of the `DataSet` is `true`.**

I have covered the main classes that make up the constituent parts of the `DataSet`, and shown how to manually generate each of these classes in code. There is another way to define a `DataTable`, `DataRow`, `DataColumn`, `DataRelation`, and `Constraint` – by using the XML schema file(s) and the XSD tool that ships with .NET. The following section describes how to set up a simple schema and generate type-safe classes to access your data.

# XML Schemas

XML is firmly entrenched into ADO.NET – indeed, the remoting format for passing data between objects is now XML. With the .NET runtime, it is now possible to describe a `DataTable` within an XML schema definition file (XSD). What's more, you can define an entire `DataSet`, with a number of `DataTable`s, a set of relationships between these tables, and include various other details to fully describe the data.

When you have defined an XSD file, there is a new tool in the runtime that will convert this schema to the corresponding data access class(es), such as the type-safe product `DataTable` class shown above. In this section we'll start with a simple XSD file that describes the same information as the `Products` sample previously shown, and then extend this to include some extra functionality. This file is `Products.xsd`, found in the `10_XSD_DataSet` folder:

```
<?xml version="1.0" encoding="utf-8" ?>
<xs:schema
   id="Products"
   targetNamespace="http://tempuri.org/XMLSchema1.xsd"
   elementFormDefault="qualified"
   xmlns="http://tempuri.org/XMLSchema1.xsd"
   xmlns:mstns="http://tempuri.org/XMLSchema1.xsd"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:msdata="urn:schemas-microsoft-com:xml-msdata">
  <xs:element name="Product">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="ProductID" type="xs:int" />
        <xs:element name="ProductName" type="xs:string" />
        <xs:element name="SupplierID" type="xs:int" minOccurs="0" />
        <xs:element name="CategoryID" type="xs:int" minOccurs="0" />
        <xs:element name="QuantityPerUnit" type="xs:string" minOccurs="0" />
        <xs:element name="UnitPrice" type="xs:decimal" minOccurs="0" />
        <xs:element name="UnitsInStock" type="xs:short" minOccurs="0" />
        <xs:element name="UnitsOnOrder" type="xs:short" minOccurs="0" />
        <xs:element name="ReorderLevel" type="xs:short" minOccurs="0" />
        <xs:element name="Discontinued" type="xs:boolean" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

We'll take a closer look at some of the options within this file in Chapter 11; for now, this file basically defines a schema with the `id` attribute set to `Products`. A complex type called `Product` is defined, which contains a number of elements, one for each of the fields within the `Products` table.

These items map onto data classes as follows. The `Products` schema maps to a class derived from `DataSet`. The `Product` complex type maps to a class derived from `DataTable`. Each sub-element maps to a class derived from `DataColumn`. The collection of all columns maps onto a class derived from `DataRow`.

Thankfully there is a tool within the .NET Framework that will produce all of the code for these classes given only the input XSD file. Because its sole job in life is to perform various functions on XSD files, the tool itself is called `XSD.EXE`.

## Generating Code with XSD

Assuming you save the above file as `Product.xsd`, you would convert the file into code by issuing the following command in a command prompt:

```
xsd Product.xsd /d
```

This creates the file `Product.cs`.

There are various switches that can be used with XSD to alter the output generated. Some of the more commonly used are shown in the table below.

| Switch | Description |
|---|---|
| `/dataset (/d)` | Generate classes derived from `DataSet`, `DataTable`, and `DataRow`. |
| `/language:<language>` | Permits you to choose which language the output file will be written in. C# is the default, but you can choose `VB` for a Visual Basic .NET file. |
| `/namespace:<namespace>` | Define the namespace that the generated code should reside within. The default is no namespace. |

An abridged version of the output from XSD for the `Products` schema is shown below. I've removed some of the less necessary code to concentrate on the most important aspects, and done some reformatting so that it will fit within the confines of a couple of pages. To see the complete output, run `XSD.EXE` on the `Products` schema (or one of your own making) and take a look at the `.cs` file generated. The example includes the entire sourcecode plus the `Product.xsd` file, and can be found in the `10_XSD_DataSet` directory:

```
//------------------------------------------------------------------------------
// <autogenerated>
//     This code was generated by a tool.
//     Runtime Version: 1.0.3512.0
//
//     Changes to this file may cause incorrect behavior and will be lost if
//     the code is regenerated.
// </autogenerated>
//------------------------------------------------------------------------------

//
// This source code was auto-generated by xsd, Version=1.0.3512.0.
//
using System;
using System.Data;
using System.Xml;
using System.Runtime.Serialization;

[Serializable()]
[System.ComponentModel.DesignerCategoryAttribute("code")]
[System.Diagnostics.DebuggerStepThrough()]
[System.ComponentModel.ToolboxItem(true)]
public class Products : DataSet
{
   private ProductDataTable tableProduct;
   public Products()
   public ProductDataTable Product
   public override DataSet Clone()
   public delegate void ProductRowChangeEventHandler ( object sender,
                                                        ProductRowChangeEvent e);

   [System.Diagnostics.DebuggerStepThrough()]
   public class ProductDataTable : DataTable, System.Collections.IEnumerable

   [System.Diagnostics.DebuggerStepThrough()]
   public class ProductRow : DataRow
}
```

I have taken some liberties with this sourcecode, as I have split it into three sections and removed any protected and private members so that we can concentrate on the public interface. The emboldened `ProductDataTable` and `ProductRow` definitions show the positions of two nested classes, which we're going to implement next. We'll look at the code for these after a brief explanation of the `DataSet` derived class.

The `Products()` constructor calls a private method, `InitClass()`, which constructs an instance of the `DataTable` class derived class `ProductDataTable`, and adds the table to the `Tables` collection of the `DataSet`. The `Products` data table can be accessed by the following code:

```
DataSet ds = new Products();
DataTable products = ds.Tables["Products"];
```

Or, more simply by using the property `Product`, available on the derived `DataSet` object:

```
DataTable products = ds.Product;
```

As the `Product` property is strongly typed, you could naturally use `ProductDataTable` rather than the `DataTable` reference I showed above.

The `ProductDataTable` class includes far more code:

```
[System.Diagnostics.DebuggerStepThrough()]
public class ProductDataTable : DataTable, System.Collections.IEnumerable
{
   private DataColumn columnProductID;
   private DataColumn columnProductName;
   private DataColumn columnSupplierID;
   private DataColumn columnCategoryID;
   private DataColumn columnQuantityPerUnit;
   private DataColumn columnUnitPrice;
   private DataColumn columnUnitsInStock;
   private DataColumn columnUnitsOnOrder;
   private DataColumn columnReorderLevel;
   private DataColumn columnDiscontinued;

   internal ProductDataTable() : base("Product")
   {
      this.InitClass();
  }
```

The `ProductDataTable` class, derived from `DataTable` and implementing the `IEnumerable` interface, defines a private `DataColumn` instance for each of the columns within the table. These are initialized again from the constructor by calling the private `InitClass()` member. Each column is given an internal accessor, which the `DataRow` class described later uses.

```
[System.ComponentModel.Browsable(false)]
public int Count
{
   get { return this.Rows.Count; }
}
internal DataColumn ProductIDColumn
{
   get { return this.columnProductID; }
}
// Other row accessors removed for clarity – there is one for each of the columns
```

Adding rows to the table is taken care of by the two overloaded (and significantly different, except unfortunately by name) `AddProductRow()` methods. The first takes an already constructed `DataRow` and returns a void. The latter takes a set of values, one for each of the columns in the `DataTable`, constructs a new row, sets the values within this new row, adds the row to the `DataTable` and returns the row to the caller. Such widely different functions shouldn't really have the same name, in my opinion.

```
public void AddProductRow(ProductRow row)
{
   this.Rows.Add(row);
}

public ProductRow AddProductRow ( string ProductName , int SupplierID ,
                                  int CategoryID , string QuantityPerUnit ,
                                  System.Decimal UnitPrice , short UnitsInStock ,
                                  short UnitsOnOrder , short ReorderLevel ,
                                  bool Discontinued )
{
   ProductRow rowProductRow = ((ProductRow)(this.NewRow()));
   rowProductRow.ItemArray = new object[]
   {
      null,
      ProductName,
      SupplierID,
      CategoryID,
      QuantityPerUnit,
      UnitPrice,
      UnitsInStock,
      UnitsOnOrder,
      ReorderLevel,
      Discontinued
   };
   this.Rows.Add(rowProductRow);
   return rowProductRow;
}
```

Just like the `InitClass()` member in the `DataSet` derived class, which added the table into the `DataSet`, the `InitClass()` member in `ProductDataTable` adds in columns to the `DataTable`. Each column's properties are set as appropriate, and the column is then appended to the columns collection.

```
private void InitClass()
{
   this.columnProductID = new DataColumn ( "ProductID",
                                           typeof(int),
                                           null,
                                           System.Data.MappingType.Element);
   this.Columns.Add(this.columnProductID);
   // Other columns removed for clarity

   this.columnProductID.AutoIncrement = true;
   this.columnProductID.AllowDBNull = false;
   this.columnProductID.ReadOnly = true;
   this.columnProductName.AllowDBNull = false;
   this.columnDiscontinued.AllowDBNull = false;
}

public ProductRow NewProductRow()
{
    return ((ProductRow)(this.NewRow()));
}
```

The last method I want to discuss, `NewRowFromBuilder()`, is called internally from the `DataTable`'s `NewRow()` method. Here it creates a new strongly typed row. The `DataRowBuilder` instance is created by the `DataTable`, and its members are only accessible within the `System.Data` assembly.

```
protected override DataRow NewRowFromBuilder(DataRowBuilder builder)
{
   return new ProductRow(builder);
}
```

The last class to discuss is the `ProductRow` class, derived from `DataRow`. This class is used to provide type-safe access to all fields in the data table. It wraps the storage for a particular row, and provides members to read (and write) each of the fields in the table.

In addition, for each nullable field, there are functions to set the field to `null`, and check if the field is `null`. The example below shows the functions for the `SupplierID` column:

```
[System.Diagnostics.DebuggerStepThrough()]
public class ProductRow : DataRow
{
   private ProductDataTable tableProduct;

   internal ProductRow(DataRowBuilder rb) : base(rb)
   {
      this.tableProduct = ((ProductDataTable)(this.Table));
   }

   public int ProductID
   {
      get { return ((int)(this[this.tableProduct.ProductIDColumn])); }
      set { this[this.tableProduct.ProductIDColumn] = value; }
   }
  // Other column accessors/mutators removed for clarity

   public bool IsSupplierIDNull()
   {
      return this.IsNull(this.tableProduct.SupplierIDColumn);
   }

   public void SetSupplierIDNull()
   {
      this[this.tableProduct.SupplierIDColumn] = System.Convert.DBNull;
   }
}
```

Now that the sourcecode for these data access classes has been generated by `XSD.EXE`, we can incorporate the classes into code. The following code utilizes these classes to retrieve data from the `Products` table and display that data to the console:

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

public class XSD_DataSet
{
   public static void Main()
```

```
      {
         string source = "server=(local)\\NetSDK;" +
                         "uid=QSUser;pwd=QSPassword;" +
                         "database=northwind";
         string select = "SELECT * FROM Products";
         SqlConnection conn = new SqlConnection(source);
         SqlDataAdapter da = new SqlDataAdapter(select , conn);
         Products ds = new Products();
         da.Fill(ds , "Product");
         foreach(Products.ProductRow row in ds.Product )
         Console.WriteLine("'{0}' from {1}" ,
                           row.ProductID ,
                           row.ProductName);
      }
   }
```

The main areas of interest are highlighted. The output of the XSD file contains a class derived from `DataSet`, `Products`, which is created and then filled by the use of the data adapter. The `foreach` statement utilizes the strongly-typed `ProductRow` and also the `Product` property, which returns the `Product` data table.

To compile this example, issue the following commands:

```
xsd product.xsd /d
```

and

```
csc /recurse:*.cs
```

The first generates the `Products.cs` file from the `Products.XSD` schema, and then the `csc` command utilizes the `/recurse:*.cs` parameter to go through all files with the extension `.cs` and add these to the resulting assembly.

# Populating a DataSet

Once you have fully defined the schema of your data set, replete with `DataTable`s, `DataColumn`s, `Constraint`s, and whatever else was necessary, you need to be able to populate the `DataSet` with some information. There are two main ways to read data from an external source and insert it into the `DataSet`:

- Use a data adapter
- Read XML into the `DataSet`

# Populating a DataSet Using a DataAdapter

The section on data rows briefly introduced the `SqlDataAdapter` class, as shown in the following code:

```
string select = "SELECT ContactName,CompanyName FROM Customers";
SqlConnection conn = new SqlConnection(source);
SqlDataAdapter da = new SqlDataAdapter(select , conn);
DataSet ds = new DataSet();
da.Fill(ds , "Customers");
```

The two highlighted lines show the `SqlDataAdapter` in use – the `OleDbDataAdapter` is again virtually identical in functionality to the `Sql` equivalent.

The `SqlDataAdapter` and `OleDbDataAdapter` are two of the classes that are derived from a common base class rather than a set of interfaces, as are most of the other `SqlClient-` or `OleDb-` specific classes. The inheritance hierarchy is shown below:

```
System.Data.Common.DataAdapter
   System.Data.Common.DbDataAdapter
      System.Data.OleDb.OleDbDataAdapter
      System.Data.SqlClient.SqlDataAdapter
```

In order to retrieve data into a `DataSet`, it is necessary to have some form of command that is executed to select that data. The command in question could be a SQL `SELECT` statement, a call to a stored procedure, or for the OLE DB provider, a `TableDirect` command. The example above utilizes one of the constructors available on `SqlDataAdapter` that converts the passed SQL `SELECT` statement into a `SqlCommand`, and issues this when the `Fill()` method is called on the adapter.

Going back to the example on stored procedures earlier in the chapter, I defined stored procedures to `INSERT`, `UPDATE`, and `DELETE`, but didn't present a procedure to `SELECT` data. We'll fill that gap in this next section, and show how you can call a stored procedure from an `SqlDataAdapter` to populate data in a `DataSet`.

## *Using a Stored Procedure in a DataAdapter*

First off we need to define a stored procedure and install it into the database. The code for this example is available in the `11_DataAdapter` directory. The stored procedure to `SELECT` data is as follows:

```
CREATE PROCEDURE RegionSelect AS
  SET NOCOUNT OFF
  SELECT * FROM Region
GO
```

Again this example is fairly trivial, and not really worthy of a stored procedure, as a direct SQL statement would normally suffice. This stored procedure can be typed directly into the SQL Server Query Analyzer, or you can run the `StoredProc.sql` file that is provided for use by this example.

Next, we need to define a `SqlCommand` that will execute this stored procedure. Again the code is very simple, and most of it was already presented in the earlier section on issuing commands:

```
private static SqlCommand GenerateSelectCommand(SqlConnection conn )
{
   SqlCommand  aCommand = new SqlCommand("RegionSelect" , conn);
   aCommand.CommandType = CommandType.StoredProcedure;
   aCommand.UpdatedRowSource = UpdateRowSource.None;
   return aCommand;
}
```

This method generates the `SqlCommand` that will call the `RegionSelect` procedure when executed. All that remains is to hook this command up to a `SqlDataAdapter`, and call the `Fill()` method:

```
DataSet ds = new DataSet();
// Create a data adapter to fill the DataSet
SqlDataAdapter da = new SqlDataAdapter();
// Set the data adapter's select command
da.SelectCommand = GenerateSelectCommand (conn);
da.Fill(ds , "Region");
```

Here I create a new `SqlDataAdapter`, assign the generated `SqlCommand` to the `SelectCommand` property of the data adapter, and then call `Fill()`, which will execute the stored procedure and insert all rows returned into the `Region DataTable` (which in this instance is generated by the runtime).

There's more to a data adapter than just selecting data by issuing a command. In the *Persisting DataSet Changes* section I will explore the rest of the facilities of the data adapter.

# Populating a DataSet from XML

In addition to generating the schema for a given `DataSet` and associated tables and so on, a `DataSet` can read and write data in native XML, such as a file on disk, a stream, or a text reader.

To load XML into a `DataSet`, simply call one of the `ReadXML()` methods, such as that shown below, which will read data from a disk file:

```
DataSet ds = new DataSet();
ds.ReadXml(".\\MyData.xml");
```

The `ReadXml()` method attempts to load any inline schema information from the input XML, and if found, uses this schema in the validation of any data loaded from that file. If no inline schema is found then the `DataSet` will extend its internal structure as data is loaded. This is similar to the behavior of `Fill()` in the previous example, which retrieves the data and constructs a `DataTable` based on the data selected.

# Persisting DataSet Changes

After editing data within a `DataSet`, it is probably necessary to persist these changes. The most common example would be selecting data from a database, displaying it to the user, and returning those updates back to the database.

In a less "connected" application, changes might be persisted to an XML file, transported to a middle-tier application server, and then processed to update several data sources.

A `DataSet` can be used for either of these examples, and what's more it's really easy to do.

# Updating with Data Adapters

In addition to the `SelectCommand` that an `SqlDataAdapter` most likely includes, you can also define an `InsertCommand`, `UpdateCommand`, and `DeleteCommand`. As these names imply, these objects are instances of `SqlCommand` (or `OleDbCommand` for the `OleDbDataAdapter`), so any of these commands could be straight SQL or a stored procedure.

With this level of flexibility, you are free to tune the application by judicious use of stored procedures for frequently used commands (say `SELECT` and `INSERT`), and use straight SQL for less commonly used commands such as `DELETE`.

For the example in this section I have resurrected the stored procedure code from the *Calling Stored Procedures* section for inserting, updating, and deleting `Region` records, coupled these with the `RegionSelect` procedure written above, and produced an example utilizes each of these commands to retrieve and update data in a `DataSet`. The main body of code is shown below; the full sourcecode is available in the `12_DataAdapter2` directory.

## *Inserting a New Row*

There are two ways to add a new row to a `DataTable`. The first way is to call the `NewRow()` method, which returns a blank row that you then populate and add to the `Rows` collection, as follows:

```
DataRow r = ds.Tables["Region"].NewRow();
r["RegionID"]=999;
r["RegionDescription"]="North West";
ds.Tables["Region"].Rows.Add(r);
```

The second way to add a new row would be to pass an array of data to the `Rows.Add()` method as shown in the following code:

```
DataRow r = ds.Tables["Region"].Rows.Add
             (new object [] { 999 , "North West" });
```

Each new row within the `DataTable` will have its `RowState` set to `Added`. The example dumps out the records before each change is made to the database, so after adding the following row (either way) to the `DataTable`, the rows will look something like the following. Note that the right-hand column shows the row state.

```
New row pending inserting into database
  1   Eastern                                          Unchanged
  2   Western                                          Unchanged
  3   Northern                                         Unchanged
  4   Southern                                         Unchanged
  999 North West                                       Added
```

To update the database from the `DataAdapter`, call one of the `Update()` methods as shown below:

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

For the new row within the `DataTable`, this will execute the stored procedure (in this instance `RegionInsert`), and subsequently I dump the records in the `DataTable` again.

```
New row updated and new RegionID assigned by database
  1   Eastern                                            Unchanged
  2   Western                                            Unchanged
  3   Northern                                           Unchanged
  4   Southern                                           Unchanged
  5   North West                                         Unchanged
```

Look at the last row in the `DataTable`. I had set the `RegionID` in code to `999`, but after executing the `RegionInsert` stored procedure the value has been changed to `5`. This is intentional – the database will often generate primary keys for you, and the updated data in the `DataTable` is due to the fact that the `SqlCommand` definition within our sourcecode has the `UpdatedRowSource` property set to `UpdateRowSource.OutputParameters`:

```
SqlCommand aCommand = new SqlCommand("RegionInsert" , conn);

aCommand.CommandType = CommandType.StoredProcedure;
aCommand.Parameters.Add(new SqlParameter("@RegionDescription" ,
                        SqlDbType.NChar ,
                        50 ,
                        "RegionDescription"));
aCommand.Parameters.Add(new SqlParameter("@RegionID" ,
                        SqlDbType.Int,
                        0 ,
                        ParameterDirection.Output ,
                        false ,
                        0 ,
                        0 ,
                        "RegionID" ,   // Defines the SOURCE column
                        DataRowVersion.Default ,
                        null));
aCommand.UpdatedRowSource = UpdateRowSource.OutputParameters;
```

What this means is that whenever a data adapter issues this command, the output parameters should be mapped back to the source of the row, which in this instance was a row in a `DataTable`. The flag states what data should be updated – the stored procedure has an output parameter that is mapped back into the `DataRow`. The column it applies to is `RegionID`, as this is defined within the command definition.

The values for `UpdateRowSource` are as follows:

| **`UpdateRowSource` Value** | **Description** |
|---|---|
| `Both` | A stored procedure may return output parameters and also a complete database record. Both of these data sources are used to update the source row. |
| `FirstReturnedRecord` | This infers that the command returns a single record, and that the contents of that record should be merged into the original source `DataRow`. This is useful where a given table has a number of default (or computed) columns, as after an `INSERT` statement these need to be synchronized with the `DataRow` on the client. An example might be '`INSERT` (columns) `INTO` (table) `WITH` (primarykey)', then '`SELECT` (columns) `FROM` (table) `WHERE` (primarykey)'. The returned record would then be merged into the original row. |

| UpdateRowSource Value | Description |
|---|---|
| None | All data returned from the command is discarded. |
| OutputParameters | Any output parameters from the command are mapped onto the appropriate column(s) in the DataRow. |

### Updating an Existing Row

Updating a row that already exists within the `DataTable` is just a case of utilizing the `DataRow` class's indexer with either a column name or column number, as shown in the following code:

```
r["RegionDescription"]="North West England";
r[1] = "North East England";
```

Both of these statements are equivalent (in this example):

```
Changed RegionID 5 description
  1   Eastern                                          Unchanged
  2   Western                                          Unchanged
  3   Northern                                         Unchanged
  4   Southern                                         Unchanged
  5   North West England                               Modified
```

Prior to updating the database, the row updated has its state set to `Modified` as shown above.

### Deleting a Row

Deleting a row is a matter of calling the `Delete()` method:

```
r.Delete();
```

A deleted row has its row state set to `Deleted`, but you cannot read columns from the deleted `DataRow` as these are no longer valid. When the adaptor's `Update()` method is called, all deleted rows will utilize the `DeleteCommand`, which in this instance executes the `RegionDelete` stored procedure.

## Writing XML Output

As you have seen already, the `DataSet` has great support for defining its schema in XML, and as you can read data from an XML document, you can also write data to an XML document.

The `DataSet.WriteXml()` method permits you to output various parts of the data stored within the `DataSet`. You can elect to output just the data, or the data and the schema. The following code shows an example of both for the `Region` example shown above:

```
ds.WriteXml(".\\WithoutSchema.xml");
ds.WriteXml(".\\WithSchema.xml" , XmlWriteMode.WriteSchema);
```

The first file, `WithoutSchema.xml` is shown below:

```
<?xml version="1.0" standalone="yes"?>
<NewDataSet>
   <Region>
      <RegionID>1</RegionID>
      <RegionDescription>Eastern                        </RegionDescription>
   </Region>
   <Region>
      <RegionID>2</RegionID>
      <RegionDescription>Western                        </RegionDescription>
   </Region>
   <Region>
      <RegionID>3</RegionID>
      <RegionDescription>Northern                       </RegionDescription>
   </Region>
   <Region>
      <RegionID>4</RegionID>
      <RegionDescription>Southern                       </RegionDescription>
   </Region>
</NewDataSet>
```

The closing tag on `RegionDescription` is over to the right of the page as the database column is defined as `NCHAR(50)`, which is a 50 character string padded with spaces.

The output produced in the `WithSchema.xml` file includes, not surprisingly, the XML schema for the `DataSet` as well as the data itself:

```
<?xml version="1.0" standalone="yes"?>
<NewDataSet>
   <xs:schema id="NewDataSet" xmlns=""
              xmlns:xs="http://www.w3.org/2001/XMLSchema"
              xmlns:msdata="urn:schemas-microsoft-com:xml-msdata">
      <xs:element name="NewDataSet" msdata:IsDataSet="true">
         <xs:complexType>
            <xs:choice maxOccurs="unbounded">
               <xs:element name="Region">
                  <xs:complexType>
                     <xs:sequence>
                        <xs:element name="RegionID"
                                    msdata:AutoIncrement="true"
                                    msdata:AutoIncrementSeed="1"
                                    type="xs:int" />
                        <xs:element name="RegionDescription"
                                    type="xs:string" />
                     </xs:sequence>
                  </xs:complexType>
               </xs:element>
            </xs:choice>
         </xs:complexType>
      </xs:element>
   </xs:schema>
   <Region>
      <RegionID>1</RegionID>
      <RegionDescription>Eastern                        </RegionDescription>
```

```
    </Region>
    <Region>
       <RegionID>2</RegionID>
       <RegionDescription>Western                        </RegionDescription>
    </Region>
    <Region>
       <RegionID>3</RegionID>
       <RegionDescription>Northern                      </RegionDescription>
    </Region>
    <Region>
       <RegionID>4</RegionID>
       <RegionDescription>Southern                      </RegionDescription>
    </Region>
 </NewDataSet>
```

Note the use in this file of the `msdata` schema, which defines extra attributes for columns within a `DataSet`, such as `AutoIncrement` and `AutoIncrementSeed` – these attributes correspond directly with the properties definable on a `DataColumn`.

# Working with ADO.NET

This last section will attempt to address some common scenarios when developing data access applications with ADO.NET.

# Tiered Development

Producing an application that interacts with data is often done by splitting the application up into tiers. A common model is to have an application tier (the front end), a data services tier, and the database itself.

One of the difficulties with this model is deciding what data to transport between tiers, and the format that it should be transported in. With ADO.NET you'll be pleased to hear that these wrinkles have been ironed out, and support for this style of architecture has been designed in from the start.

## *Copying and Merging Data*

Ever tried copying an entire OLE DB recordset? In .NET it's easy to copy a `DataSet`:

```
DataSet source = {some dataset};
DataSet dest = source.Copy();
```

This will create an exact copy of the source `DataSet` – each `DataTable`, `DataColumn`, `DataRow`, and `Relation` will be copied across verbatim, and all data will be in exactly the same state as it was in the source. If all you want to copy is the schema of the `DataSet`, you can try the following:

```
DataSet source = {some dataset};
DataSet dest = source.Clone();
```

This will again copy all tables, relations, and so on. However, each copied `DataTable` will be empty. It really couldn't be more straightforward.

A common requirement when writing a tiered system, whether based on Win32 or the web, is to be able to ship as little data as possible between tiers. This reduces the amount of resources consumed.

To cope with this requirement, the `DataSet` has the `GetChanges()` method. This simple method performs a huge amount of work, and returns a `DataSet` with only the changed rows from the source dataset. This is ideal for passing between tiers, as only a minimal set of data has to be passed across the wire.

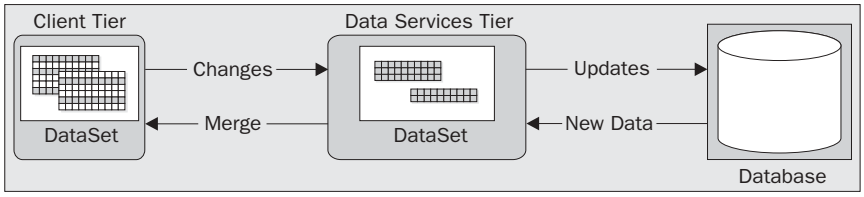
The following example shows how to generate a "changes" `DataSet`:

```
DataSet source = {some dataset};
DataSet dest = source.GetChanges();
```

Again, this is trivial. Under the covers things are a little more interesting. There are two overloads of the `GetChanges()` method. One overload takes a value of the `DataRowState` enumeration, and returns only rows that correspond to that state (or states). `GetChanges()` simply calls `GetChanges(Deleted | Modified | Added)`, and first checks to ensure that there are some changes by calling `HasChanges()`. If no changes have been made, then a `null` is returned to the caller immediately.

The next operation is to clone the current `DataSet`. Once done, the new `DataSet` is set up to ignore constraint violations (`EnforceConstraints = false` ), and then each changed row for every table is copied into the new `DataSet`.

Once you have a `DataSet` that just contains changes, you can then move these off to the data services tier for processing. Once the data is updated in the database, the "changes" `DataSet` can be returned to the caller (as there may, for example, be some output parameters from the stored procedures that have updated values in the columns). These changes can then be merged into the original `DataSet` using the `Merge()` method. This sequence of operations is depicted below:




# Key Generation with SQL Server

The `RegionInsert` stored procedure presented earlier in the chapter was one example of generating a primary key value on insertion into the database. The method for generating the key was fairly crude and wouldn't scale well, so for a real application you should look at utilizing some other strategy for generating keys.

Your first instinct might be simply to define an identity column, and return the `@@IDENTITY` value from the stored procedure. The following stored procedure shows how this might be defined for the `Categories` table in the `Northwind` example database. Type this stored procedure into SQL Query Analyzer, or run the `StoredProcs.sql` file in the `13_SQLServerKeys` directory:

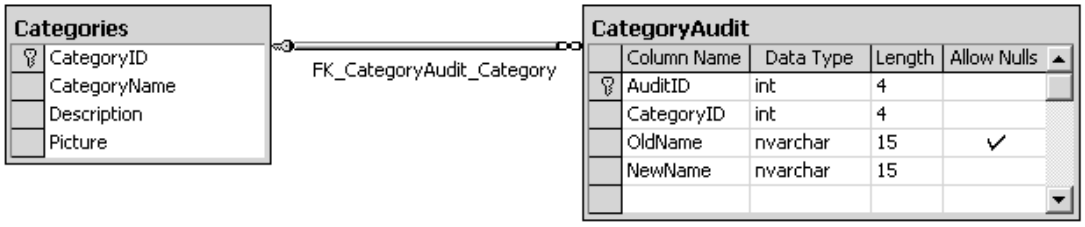
```
CREATE PROCEDURE CategoryInsert(@CategoryName NVARCHAR(15),
                                @Description NTEXT,
                                @CategoryID INTEGER OUTPUT) AS
   SET NOCOUNT OFF
   INSERT INTO Categories (CategoryName, Description)
      VALUES(@CategoryName, @Description)
   SELECT @CategoryID = @@IDENTITY
GO
```

This inserts a new row into the `Category` table, and returns the generated primary key to the caller. You can test the procedure by typing in the following SQL in Query Analyzer:

```
DECLARE @CatID int;
EXECUTE CategoryInsert 'Pasties' , 'Heaven Sent Food' , @CatID OUTPUT;
PRINT @CatID;
```

When executed as a batch of commands, this will insert a new row into the `Categories` table, and return the identity of the new record, which is then displayed to the user.

Let's say that some months down the line, someone decides to add in a simple audit trail, which will record all insertions and modifications made to the category name. You define a table such as that shown below, which will record the old and new value of the category:




The creation script for this table is included in the `StoredProcs.sql` file. The `AuditID` column is defined as an `IDENTITY` column. You then construct a couple of database triggers that will record changes to the `CategoryName` field:

```
CREATE TRIGGER CategoryInsertTrigger
   ON Categories
   AFTER UPDATE
AS
   INSERT INTO CategoryAudit(CategoryID , OldName , NewName )
      SELECT old.CategoryID, old.CategoryName, new.CategoryName
      FROM Deleted AS old,
           Categories AS new
      WHERE old.CategoryID = new.CategoryID;
GO
```

For those of you used to Oracle stored procedures, SQL Server doesn't exactly have the concept of `OLD` and `NEW` rows, instead for an insert trigger there is an in memory table called `Inserted`, and for deletes and updates the old rows are available within the `Deleted` table.

This trigger retrieves the `CategoryID` of the record(s) affected, and stores this together with the old and new value of the `CategoryName` column.

Now, when you call your original stored procedure to insert a new `CategoryID`, you receive an identity value; however, this is no longer the identity value from the row inserted into the `Categories` table, it is now the new value generated for the row in the `CategoryAudit` table. Ouch!

To view the problem first hand, open up a copy of SQL Server Enterprise manager, and view the contents of the `Categories` table.

| | CategoryID | CategoryName | Description |
|---|---|---|---|
| | 1 | Beverages | Soft drinks, coffees, teas, beers, and ales |
| | 2 | Condiments | Sweet and savory sauces, relishes, spreads, and seasonings |
| | 3 | Confections | Desserts, candies, and sweet breads |
| | 4 | Dairy Products | Cheeses |
| | 5 | Grains/Cereals | Breads, crackers, pasta, and cereal |
| | 6 | Meat/Poultry | Prepared meats |
| | 7 | Produce | Dried fruit and bean curd |
| | 8 | Seafood | Seaweed and fish |
| ▶ | 20 | Pasties | Heaven Sent Grub |
| * | | | |

This lists all the categories I have in my instance of the database.

The next identity value for the `Categories` table should be `21`, so we'll insert a new row by executing the code shown below, and see what `ID` is returned as follows:

```
DECLARE @CatID int;
EXECUTE CategoryInsert 'Pasties' , 'Heaven Sent Food' , @CatID OUTPUT;
PRINT @CatID;
```

The output value of this on my PC was `17`. If I look into the `CategoryAudit` table, I find that this is the identity of the newly inserted audit record, not that of the category record created.

| | AuditID | CategoryID | OldName | NewName |
|---|---|---|---|---|
| ▶ | 17 | 30 | <NULL> | Vegetables |
| * | | | | |

The problem lies in the way that `@@IDENTITY` actually works. It returns the `LAST` identity value created by your session, so as shown above it isn't completely reliable.

There are two other identity functions that you can utilize instead of `@@IDENTITY`, but neither are free from possible problems. The first, `SCOPE_IDENTITY()`, will return the last identity value created within the current "scope". SQL Server defines scope as a stored procedure, trigger, or function. This may work most of the time, but if for some reason someone adds another `INSERT` statement into the stored procedure, then you will receive this value rather than the one you expected.

The other, `IDENT_CURRENT()` will return the last identity value generated for a given table in any scope, so for instance, if two users were accessing SQL Server at exactly the same time, it might be possible to receive the other user's generated identity value.

As you might imagine, tracking down a problem of this nature isn't easy. The moral of the story is to beware when utilizing `IDENTITY` columns in SQL Server.

# Naming Conventions

Having worked with database applications all my working life, I've picked up a few recommendations for naming entities, which are worth sharing. I know, this isn't really .NET related, but the conventions are useful especially when naming constraints as above. Feel free to skip this section if you already have your own views on the subject.

## *Database Tables*

- Always use singular names – `Product` rather than `Products`. This one is largely due to having to explain to customers a database schema – it's much better grammatically to say "The `Product` table contains products" than "The `Products` table contains products". Have a look at the `Northwind` database as an example of how not to do this.
- Adopt some form of naming convention for the fields that go into a table – ours is `<Table>_ID` for the primary key of a table (assuming that the primary key is a single column), `Name` for the field considered to be the user-friendly name of the record, and `Description` for any textual information about the record itself. Having a good table convention means you can look at virtually any table in the database and instinctively know what the fields are used for.

## *Database Columns*

- Use singular rather than plural names again.
- Any columns that link to another table should be named the same as the primary key of that table. So, a link to the `Product` table would be `Product_ID`, and to the `Sample` table `Sample_ID`. This isn't always possible, especially if one table has multiple references to another. In that case use your own judgment.
- Date fields should have a suffix of `_On`, as in `Modified_On`, `Created_On`. Then it's easy to read some SQL output and infer what a column means just by its name.
- Fields that record the user should be suffixed with `_By`, as in `Modified_By` and `Created_By`. Again, this aids legibility.

## *Constraints*

- If possible, include in the name of the constraint the table and column name, as in `CK_<Table>_<Field>`. Examples would be `CK_PERSON_SEX` for a check constraint on the `SEX` column of the `PERSON` table. A foreign key example would be `FK_Product_Supplier_ID`, for the foreign key relationship between product and supplier.
- Show the type of constraint with a prefix, such as `CK` for a check constraint and `FK` for a foreign key constraint. Feel free to be more specific, as in `CK_PERSON_AGE_GT0` for a constraint on the age column indicating that the age should be greater than zero.
- If you have to trim the length of the constraint, do it on the table name part rather than the column name. When you get a constraint violation, it's usually easy to infer which table was in error, but sometimes not so easy to check which column caused the problem. Oracle has a 30-character limit on names, which you can easily hit.

### *Stored Procedures*

Just like the obsession many have fallen into over the past few years of putting a 'C' in front of each and every class they have declared (you know you have!), many SQL Server developers feel compelled to prefix every stored procedure with 'sp_' or something similar. It's not a good idea.

SQL Server uses the 'sp_' prefix for all (well, most) system stored procedures. So, on the one hand, you risk confusing your users into thinking that 'sp_widget' is something that comes as standard with SQL Server. In addition, when looking for a stored procedure, SQL Server will treat procedures with the 'sp_' prefix differently from those without.

If you use this prefix, and do not qualify the database/owner of the stored procedure, then SQL Server will look in the current scope, then jump into the master database and look up the stored procedure there. Without the 'sp_' prefix your users would get an error a little earlier. What's worse, and also possible to do, is to create a local stored procedure (one within your database) that has the same name and parameters as an system stored procedure. I'd avoid this at all costs – if in doubt, don't prefix.

Above all, when naming entities, whether within the database or within code, *be consistent*.

## Performance

The current set of managed providers available for .NET are somewhat limited – you can choose `OleDb` or `SqlClient`; `OleDb` permits connection to any data source exposed with an OLE DB driver (such as Oracle), and the `SqlClient` provider is tailored for `SqlServer`.

The `SqlClient` provider has been written completely in managed code, and uses as few layers as possible to connect to the database. This provider writes **TDS** (**Tabular Data Stream**) packets direct to SQL Server, which should be substantially faster than the `OleDb` provider, which naturally has to go through a number of layers before actually hitting the database.

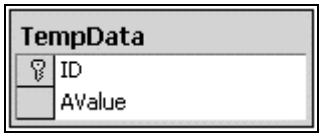
To test the theory, the following code was run against the same database on the same machine, the only difference being the use of the `SqlClient` managed provider over the ADO provider:

```
SqlConnection conn = new SqlConnection(Login.Connection);
conn.Open();
SqlCommand cmd = new SqlCommand ( "update tempdata set AValue=1 Where ID=1" ,
                                  conn);

DateTime   initial, elapsed ;
initial = DateTime.Now ;
for(int i = 0; i < iterations; i++)
   cmd.ExecuteNonQuery();
elapsed = DateTime.Now ;

conn.Close();
```

Naturally the OLE DB version utilizes `OleDbCommand` rather than `SqlCommand`. I created a simple database table with two columns as shown below, and manually added a single row:




The SQL clause used was a simple `UPDATE` command:

```
UPDATE TempData SET AValue = 1 WHERE ID = 1.
```

The SQL was kept deliberately simple to attempt to highlight the differences in the providers. The results (in seconds) achieved for various combinations of iterations were as follows :

| Provider | 100 | 1000 | 10000 | 50000 |
|---|---|---|---|---|
| `OleDb` | 0.109 | 0.798 | 7.95 | 39.11 |
| `Sql` | 0.078 | 0.626 | 6.23 | 29.27 |

If you are only targeting SQL Server then the obvious choice is the `Sql` provider. Back in the real world, if you target anything other than SQL Server you naturally have to use the `OleDb` provider. Or do you?

As Microsoft has done an excellent job of making database access generic with the `System.Data.Common` classes, it would be better to write code against those classes, and use the appropriate managed provider at run time. It's fairly simple to swap between `OleDb` and `Sql` now, and if other database vendors write managed providers for their products, you will be able to swap out ADO for a native provider with little (or no) code changes. For an example of the versatility of .NET data access, The "Scientific Data Center" case study in "*Data-Centric .NET Programming with C#*" (Wrox Press, ISBN 1-861005-92-x) details using C# to query a MySQL database.

# Summary

The subject of data access is a large one, especially in .NET as there is an abundance of new material to cover. This chapter has provided an outline of the main classes in the ADO.NET namespaces, and shown how to use the classes when manipulating data from a data source.

Firstly, we explored the use of the `Connection` object, through the use of both the `SqlConnection` (SQL Server specific) and `OleDbConnection` (for any OLE DB data sources). The programming model for these two classes is so similar that one can normally be substituted for the other and the code will continue to run.

After illustrating how to connect to and disconnect from the data source, we then discussed how to do it properly, so that scarce resources, such as database connections, could be closed as early as possible. Both of the connection classes implement the `IDisposable` interface, called when the object is placed within a `using` clause. If there's one thing I'd like you to take away from this chapter is the importance of closing database connections as early as possible.

We then discussed database commands, through examples that executed with no returned data, to calling stored procedures with input and output parameters. Various execute methods were described, including the `ExecuteXmlReader` method available only on the SQL Server provider. This vastly simplifies the selection and manipulation of XML-based data.

The generic classes within the `System.Data` namespace were all described in detail, from the `DataSet` class through `DataTable`, `DataColumn`, `DataRow` and on to relationships and constraints. The `DataSet` class is an excellent container of data, and various methods make it ideal for cross tier data flow. The data within a `DataSet` can be represented in XML for transport, and in addition, methods are available that will pass a minimal amount of data between tiers. The ability of having many tables of data within a single `DataSet` can greatly increase its usability; being able to maintain relationships automatically between master/details rows will be expanded upon in the next chapter.

Having the schema stored within a `DataSet` is one thing, but .NET also includes the data adapter that along with various `Command` objects can be used to select data into a `DataSet` and subsequently update data in the data store. One of the beneficial aspects of a data adapter is that a distinct command can be defined for each of the four actions – `SELECT`, `INSERT`, `UPDATE` and `DELETE`. The system can create a default set of commands based on database schema information and a `SELECT` statement, but for the best performance, a set of stored procedures can be used, with the `DataAdapter`'s commands defined appropriately to pass only the necessary information to these stored procedures.

As XML and XSD schemas have become feverishly popular over the past couple of years, we discussed how to convert an XSD schema into a set of database classes using the XSD tool `XSD.EXE` that ships with .NET. The classes produced are ready to be used within an application, and their automatic generation can save many hours of laborious typing.

During the last few pages of the chapter we've gone through some best practices and naming conventions for database development. Although not strictly .NET-related, these were thought to be a worthwhile inclusion. A set of conventions should always be adhered to when programming, whether in C# against a SQL Server database or in Perl scripts on Linux.

Armed with this knowledge, we're now in a good position to move on to the next chapter, where we'll explore the use of Visual Studio and .NET's Windows Forms data controls.