THE EXPERT'S VOICE® IN .NET

For a limited time, get the free, fully searchable eBook—a $25 value! See inside for details.

Pro ADO.NET 2.0

The only ADO.NET book you will ever need.

Sahil Malik

Apress®

Pro ADO.NET 2.0

Sahil Malik

Contents

About the Author xv
About the Technical Reviewers xvii
Acknowledgments xix
Introduction xxi

CHAPTER 1 An Introduction to ADO.NET 1

What Is ADO.NET? 2
What Is Wrong with ADO? 3
Meeting the Players: Important Objects in ADO.NET 4
The Connected Objects 4
The Disconnected Objects 5
.NET Data Providers 7
Using the ProviderBase Model 9
Third-Party .NET Data Providers 12
The System.Data.Design Namespace 14
Summary 14

CHAPTER 2 The ADO.NET Object Model 15

This Is a Reference Chapter 15
10,000-Ft. View of ADO.NET 16
Establishing a Connection: DbConnection 17
Executing a Command: DbCommand and DbTransaction 19
Creating Parameterized Commands: DbParameter 20
Holding Disconnected Data: DataSet 22
Fetching Data: Data Reader and Data Adapter 24
The Connected Way to Fetch Data: DbDataReader 24
The Bridge Between Connected and Disconnected: DbDataAdapter 25
Exceptions in ADO.NET 26
Summary 30

Mapping 204
Using the SQL AS Keyword 204
The ADO.NET Mapping Mechanism 206
Summary 212

CHAPTER 8 Sorting, Searching, and Filtering 213

Setting Up the Data Source 214
Working in a DataTable 216
Finding a Row 217
Selecting a Number of Rows 219
Expressions: Calculating Columns on the Fly 222
Performing Aggregate Calculations 224
Working with the DataRelation Object 226
Working with the DataView Object 232
Creating a DataView 232
Leveraging XML to Work with Disconnected Data 240
XmlDataDocument 241
Summary 245

CHAPTER 9 Updating Data 247

Updating a Table: The Easy Drag-and-Drop Approach 248
How Does It All Work? 254
Using the Command Builder Object 262
State Management in a DataRow and Its Use in Updating Data 266
Moving Large Amounts of Data: SqlBulkCopy 276
Editing Disconnected Data 278
Add New Rows 278
Modify Existing Rows 279
Delete Existing Rows 281
A Real-World Example 283
Writing This Application in Oracle 293
Optimizing Your Application: GetChanges and Merge 294
Merge Case 1: Same Table Structures, No Primary Key 302
Merge Case 2: Same Table Structures, with Primary Key 303
Merge Case 3: Common Column, No Primary Key 305
Merge Case 4: Common Column, with Primary Key 306
Merge Case 5: Absolutely Different Table Structures 308
Merging Two DataSets/DataTables with Different Schemas 310
Updating Records Using Mapped Names 311

Introduction

. . . Mission control to reader . . . you are now nearing the ADO.NET planet in the .NET solar system of the Microsoft technology galaxy. Make sure no architectural mistake alien eats you up for dinner . . .

Learning any new topic is like approaching a new planet. As you approach the planet from a distance, you first identify its place in the solar system, then the major geographical features on the surface, and finally you land on it and start digging deep and constructing buildings to finally call it your home. Then one day before you know it, you are married to a Mrs. Alien, have two kids, a mortgage, a car payment, and find yourself worrying about your kidÕs college education fund.

It is true!! Life is like a computer game, it keeps getting harder and then you die.

So why should learning ADO.NET be any different? DoesnÕt it make sense to start at the basics and then graduate to the complex?

This book begins with three rather short (about 50 pages combined) and simple chapters:

- The first chapter identifies where ADO.NET is located in the .NET solar system and its various major building blocks.
- The second chapter begins with identifying the major geographical features of the ADO.NET terrain. It serves very well as a map for the future chapters when you are on the ground digging deeper. Because this chapter is a map, you will be reminded to reference back to the various figures, class names, and namespaces presented in this chapter as you dig deeper in the terrain.
- The third chapter is when you land on the planet and start walking around and create four data-driven applications of increasing complexity.

Once you have landed on the planet, are armed with a map of the area, and have walked around a bit is when itÕs time to start digging deeper and do what we humans do so naturallyÑ exploring (without exploding hopefully).

So let me ask you a question, When you hold a tool such as a hammer in your hand, what do you do with it? You bang things such as a nail with great force on its head, right?

Now what if someone started telling you, here is a hammer, it has two partsÑthe head and the handle. The handle is long and thus helps you exert torque because torque is directly proportional to the radius of the torque arm. The torque translates to a lot of momentum in a rather heavy hammer head. Now because momentum can neither be destroyed nor created per the equation

M1V1 = M2V2

and because the mass of the nail is so little, the momentum gets transferred to the nail, which results in a very high nail velocity thus driving it through the wood.

CHAPTER 1

An Introduction to ADO.NET

A computer can be thought of as an information storage and processing machine. While not every application has a specialized program managing its store of information, itÕs hard to imagine a computer program that doesnÕt work with any kind of data. Certain applications, like Microsoft Word and Notepad, choose to manage their own data, while many other specialized applications, especially those that require vast amounts of data, choose a much more specialized program or architecture that runs on a separate machine, typically referred to as a *database*.

While some applications choose to use a server-based database architecture, like Oracle, Microsoft SQL Server, MySQL, DB2, and others, certain other applications might choose a file-based architecture instead, such as Microsoft Access or Excel.

Even various programs on a computer allow you to manage information effectively: programs that are designed specifically to handle information, such as databases that handle information quite differently than programs that sit between the user and the database. Most databases store their information as tables, which arrange data as a collection of rows and columns and values within them. Most modern databases will also let you specify relationships between these tables, which allow the database to keep data sanctity between various tables that have relationships between them.

However, programming languages have a different method of representing data. In particular, most modern-day object-oriented languages choose to represent data in hierarchical representations of objects.

In fact, one program could work with more than one data source at a time and it needs some sort of data access libraries to accomplish this task, as shown in Figure 1-1.

just like a DataSet. For data access needs where your DataSet might contain only one DataTable, it may make more sense to use a DataTable instead. As you'll see in future chapters, this is not only more convenient, but it's also better performing.

- **DataRow** One of the properties of DataTable is Rows of DataRowCollection type, which represents an enumerable collection of DataRow objects. As data is filled into a DataTable, the DataRowCollection gets new DataRow objects added to itself. The best logical equivalent of a DataRow in a database is a row in a table.

- **DataColumn** A DataTable also contains a Columns property of DataColumnCollection type. Essentially, this represents the structure of a DataTable. The best logical equivalent of a DataColumn object in a database is an individual column in a given table in a database.

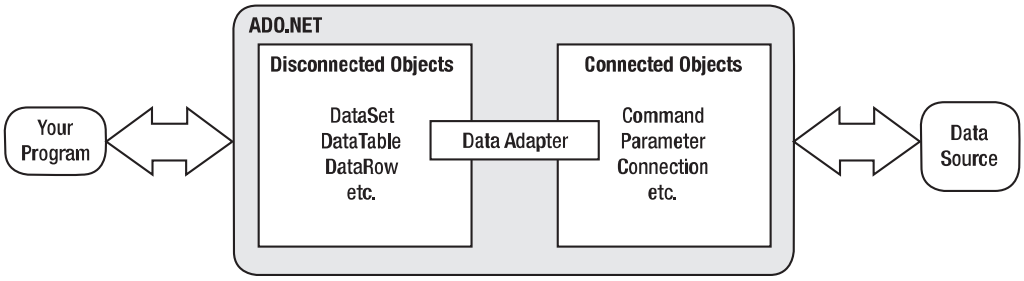
- **DataView** A DataView is most similar to a view in a database. A DataView allows you to create a "view" on a DataTable and view a subset of the data based on a preset condition specified in its Filter property. You could also use the Sort property to sort the filtered subset of the DataTable's data. One DataTable can have multiple views defined on it.

- **Constraint** : A DataTable contains yet another property called Constraints of ConstraintsCollection type. This lets you create ForeignKeyConstraint or UniqueConstraint objects and associate various columns to certain conditions based on which data in the DataTable must pass for it to exist in the DataTable. The most logical equivalent of a ForeignKeyConstraint is a foreign key in a database, and UniqueConstraint specifies a Unique condition on a given column in a database.

- **DataRelation** : A DataSet, like a database, might contain various interrelated tables. A DataRelation object lets you specify relations between various tables that allow you to both validate data across tables and browse parent and child rows in various DataTables. Its most logical equivalent is a foreign key specified between two tables in a database. The difference between a ForeignKeyConstraint and a DataRelation is that a DataRelation , in addition to validating data, gives you a convenient mechanism to browse parent and child rows in a DataSet.

Figure 1-3 shows where the various connected and disconnected objects fit into the bigger picture.

Note that in Figure 1-3, your program talks with ADO.NET as a whole. In other words, it can choose to use the disconnected objects, the DataAdapter, the connected objects, or a combination thereof.



and the DataAdapter

```
    Console.WriteLine("Description:"+ factoryClass["Description"]);
    Console.WriteLine("Invariant Name:"+ factoryClass["InvariantName"]);
    Console.WriteLine("\n");
}
```

Listing 1-3. Enumerating Through Available Providers in Visual Basic .NET

```
Dim factoryClassesTable As DataTable =  DbProviderFactories.GetFactoryClasses()
Dim factoryClass As DataRow
For Each factoryClass In factoryClassesTable.Rows
    Console.WriteLine("Name:" & factoryClass("Name"))
    Console.WriteLine("Description:" & factoryClass("Description"))
    Console.WriteLine("Invariant Name:" & factoryClass("InvariantName"))
    Console.WriteLine("")
Next
```

When this code is run, it produces output as shown here:

```
Name:Odbc Data Provider
Description:.Net Framework Data for Odbc
Invariant Name:System.Data.Odbc

Name:OleDb Data Provider
Description:.Net Framework Data for OleDb
Invariant Name:System.Data.OleDb

Name:OracleClient Data Provider
Description:.Net Framework Data for Oracle
Invariant Name:System.Data.OracleClient

Name:SQL Server CE Data Provider
Description:.NET Framework Data Provider for Microsoft SQL Server 2005 Mobile
Edition
Invariant Name:Microsoft.SqlServerCe.Client
```

If you wanted to actually use one of these data providers, you could use the code shown in Listings 1-4 and 1-5.

Listing 1-4. Putting the ProviderBase Model to Work in C#

```
//Select SQL Client factory - Can change to use any provider later
DbProviderFactory factory = DbProviderFactories.GetFactory("System.Data.SqlClient");

//Create Connection from the factory
SqlConnection testConnection = (SqlConnection)factory.CreateConnection();
testConnection.ConnectionString = "..."; //Specify connection string … See Chapter 4
testConnection.Open();

//Create Command from the factory
```

```
//Execute a  command from the conneciton
testCommand.Connection = testConnection;
testCommand.CommandText = "...";
SqlDataReader reader = testCommand.ExecuteReader();
while (reader.Read())
{
    Console.WriteLine(reader.GetValue(0));
}
```

Listing 1-5. Putting the ProviderBase Model to Work in Visual Basic .NET

```
'Select SQL Client factory - Can change to use any provider later
Dim factory As DbProviderFactory =  _
    DbProviderFactories.GetFactory("System.Data.SqlClient")

'Create Connection from the factory
Dim testConnection As SqlConnection = factory.CreateConnection()
testConnection.ConnectionString = "..." ' Specify connection string … See Chapter 4
testConnection.Open()

'Create Command from the factory
Dim testCommand As SqlCommand =  factory.CreateCommand()

'Execute a  command from the conneciton
testCommand.Connection = testConnection
testCommand.CommandText = "..."
Dim reader As SqlDataReader =  testCommand.ExecuteReader()
While reader.Read()
    Console.WriteLine(reader.GetValue(0))
End While
```

One of the things you might note in Listings 1-4 and 1-5 is that the ProviderBase model allows you to retrieve strongly typed provider-specific objects such as SqlConnection and SqlCommand without knowing in advance which provider the user intends to work with. The user could have chosen "System.Data.SqlClient" through a drop-down on his UI, which is simply passed to the GetFactory method as a string parameter.

This is possible because the DbProviderFactory object always returns objects of data types that are actually base classes to various common objects in a .NET data provider. These common base objects contained in the System.Data.Common namespace can be inherited by any other class to create a third-party .NET data provider.

Third-Party .NET Data Providers

ADO.NET contains various base classes and interfaces that various third-party .NET data providers can derive or implement to create their own specific implementations of commonly used objects such as Connection, Command, and so on. One such example is the DbDataAdapter class that implements the IDbDataAdapter interface. Thus, the SqlDataAdapter that can be found in System.Data.SqlClient derives from DbDataAdapter, and hence implements the IDbDataAdapter e shown in Table 1-2.

Table 1-2. Main System.Data.Common Classes

Class Name	Description
DataAdapter	This class acts as a sentry between the connected and disconnected spheres of ADO.NET. It holds a set of data commands (DbCommand objects) and a data source connection (DbConnection object) that are used to fill the DataSet or DataTable and update the data source. This data source can be any type of data source, unlike the DbDataAdapter, which is only used with relational data sources.
DbCommand	This class is used for executing SQL commands, such as SELECT queries against the data source.
DbCommandBuilder	This class is used to create the INSERT, UPDATE, and DELETE SQL statements for the command objects used by a data adapter. It can be used only with a single data source table, and only if the SELECT SQL statement has been specified and at least one unique column is returned as part of the row schema.
DbConnection	This class is the actual connection to the data source.
DbConnectionOptions	This class is used by the provider factory classes. It's used by the provider factory to create a connection; however, it can also be used manually for purposes such as splitting a connection string into key-value pairs. It has some properties and methods that can convert the string values in a key-value pair to integers and Booleans and check for the existence of a specific key, such as Data Source.
DbConnectionStringBuilder	This is a base class for creating connection strings, and it can be used in connection with a data provider factory to create a connection string, edit a connection string, read a connection string from a configuration file, and save a connection string to a configuration file. This is covered in Chapter 4.
DbDataAdapter	This class is an abstract helper class that is used with the IDbDataAdapter interface. The DbDataAdapter class is derived from the DataAdapter class, and it is used to create a data adapter for a relational database. This is done by creating a class that inherits the DbDataAdapter class and implements the IDbDataAdapter interface.
DbDataReader	This class reads a forward-only stream of rows from a data source, and the rows can be accessed one at a time. The connection must stay open while you're reading the rows from the DbDataReader, because the rows are read directly from the data source when requested.
DbDataRecord	This class implements the IDataRecord and ICustomTypeDescriptor interfaces. This way it provides data-binding support for the DbEnumerator class. It is often used with data binding on ASP.NET pages instead of using the DataBinder.Eval method, which incurs a performance overhead by using reflection.
DbException	This is the generic data exception class, used for throwing data-related exceptions. This abstract class inherits from the ExternalException class.

Continued

CHAPTER 2

The ADO.NET Object Model

Sir Isaac Newton, the British physicist who discovered gravity in the seventeenth century, was a brilliant man. But letÕs say that one day on his way to Cambridge he accidentally walked into a wormhole and somehow showed up in the twenty-first century. What would his reaction be?

Now imagine that he walks up to a large, painted, metal and glass box with four seats inside, four doors, four wheels, a place to pour fuel, and some kind of controls to direct the movement of that box. What would he consider that box to be?

If the last chapter was a little bit like explaining that the box is a car and itÕs controlled by using the pedals on the floor and the steering wheel on the dash, then this chapter is a bit like opening the hood of the car, looking inside at the machinery, and understanding how that liquid fuel makes this contraption move. Also, since you and I are engineers and engineers not only like to drive the car, but also to break it, repair it, and fully understand how the fuel makes the car move, itÕs imperative that we understand how to perform such operations on the ADO.NET car.

Before Sir Isaac Newton can learn how to rebuild an engine or change a timing belt, he first needs to understand what an engine is, what its purpose is, what it looks like, how it works, and where in that box the engine resides.

Chapter 1 introduced you to where ADO.NET fits into your architecture. You also saw how ADO.NET is split into two main parts, connected and disconnected, and how you can have various .NET data providers that allow you to work with different data sources.

This chapter takes the discussion away from logical block diagrams (learning how to drive the car) to a deeper insight into ADO.NET using class diagrams and class hierarchy (learning about the machinery). The purpose of understanding how various classes within ADO.NET are laid out is to enable you to reason how the commonality is enforced amongst diverse .NET data providers, and how the common disconnected parts of ADO.NET are able to perform a common role within ADO.NET.

This Is a Reference Chapter

As engineers, we like to understand what is under the hood before we sit behind the steering wheel; however, simply telling you what exists under the hood of ADO.NET without establishing practical groundwork underneath it all serves very little purpose.

This chapter introduces a lot of classes, interfaces, and namespaces that exist within ADO.NET. It makes no sense to memorize by rote every single class name, interface, or namespace

Establishing a Connection: DbConnection

In order to work with a data source, your program needs to establish a connection with it. Due to the variety of data sources available, the information required to connect to any given data source might be very different. For instance, you might need to supply a user ID and password along with the server and database name in order to connect to a Microsoft SQL Server database, but connecting to a Microsoft Access database would require a file-path location. Also, every data source might support a different set of operations; for instance, a Microsoft SQL Server database might allow you to change databases, which is an operation thatÕs probably meaningless in Microsoft Access.

There are differences in each data source; however, at the end of the day, they are all data sources, thus there is a large degree of commonality between them as well. At the very least (just because the entity being discussed is a data source you can connect with), there must be some functionality to open a connection, close a connection, and also check the existing connection state.

Given the fact that there are differences and commonalities, it makes sense to have individual implementations for each data sourceÕs connection object, which all inherit from the same base class and implement the same interface.

What a coincidence, this is exactly what ADO.NET does! The connection object inherits from the `DbConnection` base class, which in turn implements the `IDbConnection` interface. Thus, the `SqlConnection` class, which appears at `System.Data.SqlClient.SqlConnection`, inherits from `System.Data.Common.DbConnection`, which in turn implements `System.Data.IDbConnection`. Similarly, the `OracleConnection` class, which appears at `System.Data.OracleClient.OracleConnection`, also inherits from `DbConnection` and implements `IDbConnection`.

This hierarchy can be seen in Figure 2-1.

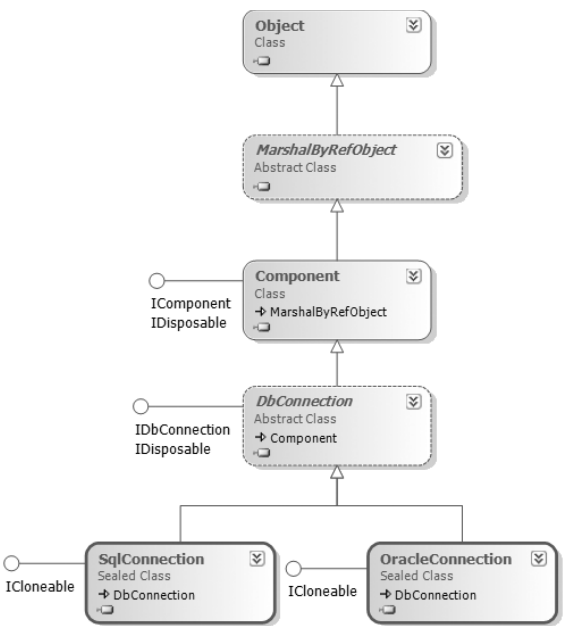


Figure 2-1. faces

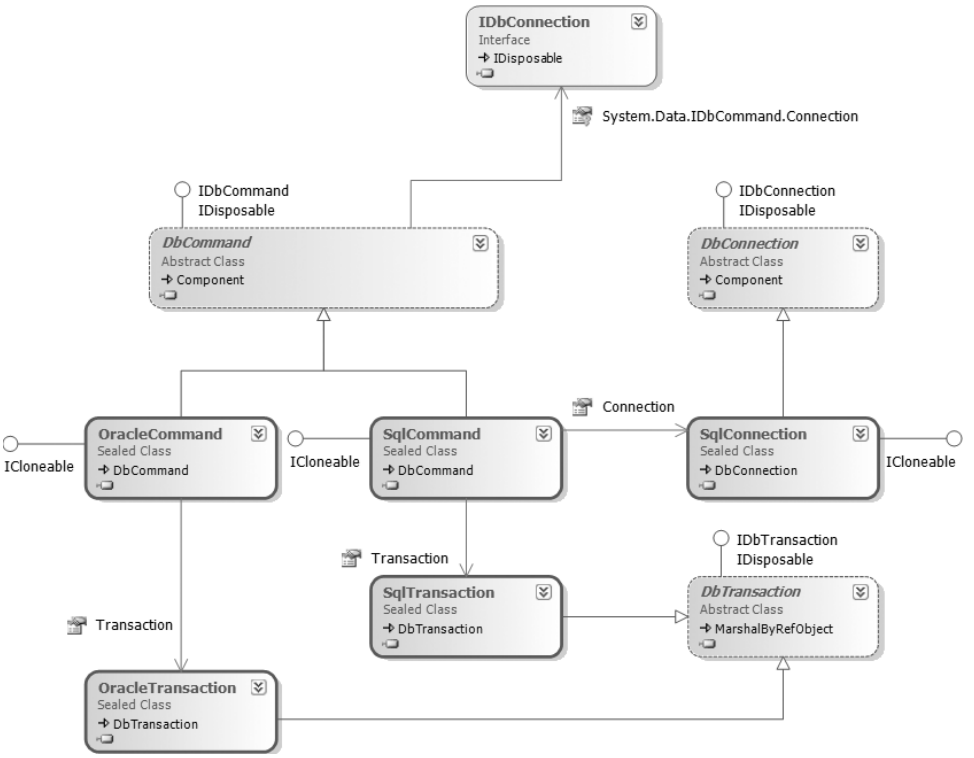


Figure 2-3. The SqlCommand and OracleCommand objects, their Transaction and Connection properties, and their base class and interface implementations

Creating Parameterized Commands: DbParameter

I am lazy. I donÕt want to write a new command for every single customer row that might appear in the Customers table. Fortunately, my boss agrees with me, because he doesnÕt want me wasting time where I shouldnÕt have to. And, fortunately, even most databases agree with me because they try to improve their performance by caching query plans when they see a command with the same repeating structure. Thus, it makes sense to create a parameterized query for every Customer row in the Customers table. Parameterized queries have other advantages too, like being more resistant to injection attacks, in general not requiring you to delimit single quote characters anytime you deal with a string parameter.

This can be achieved by specifying the command text once and using parameters to achieve the flexibility you may need for different Customer rows within the same table.

Thus, instead of writing a query that looks like

```
Update Customers Set FirstName = 'Sahil'  where CustomerID = 1
```

you could, instead, write the query like

```
Update Customers Set FirstName = @FirstName where CustomerID = @CustomerID
```

Writing this query allows you to specify values for the two involved parameters, 'Sahil' .

Since a command is represented by the DbCommand object and every .NET data provider may need to have parameterized commands, this commonality is enforced at the DbCommand level by providing you with a property called Parameters. The Parameters property is of DbParametersCollection data type, which is nothing but a collection of DbParameter objects. This can be seen in Figure 2-4.

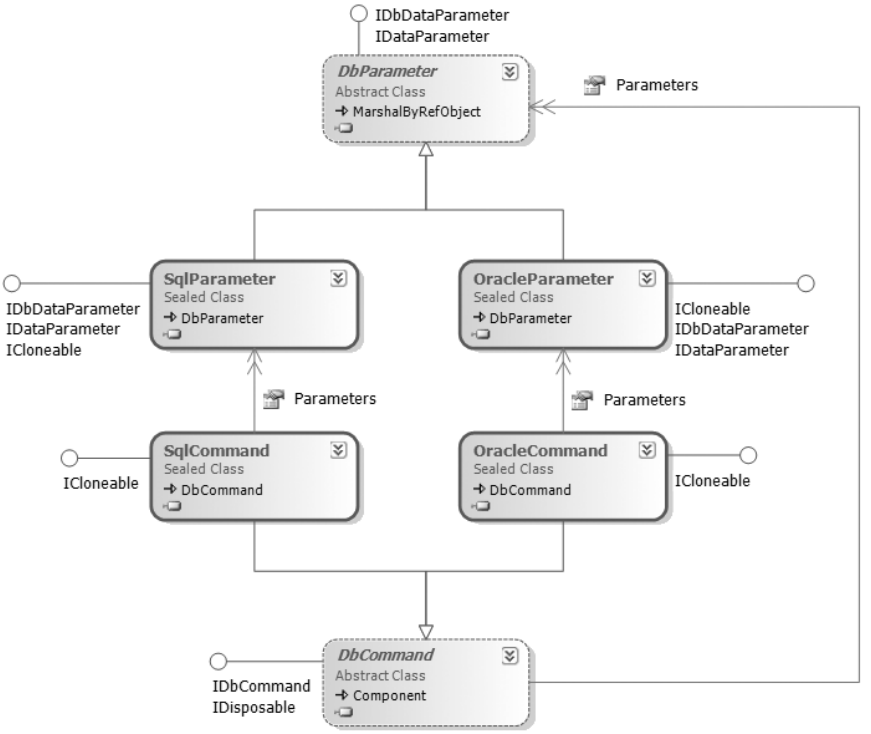


Figure 2-4. Specifying Parameters to the SqlCommand and OracleCommand objects using SqlParameter and OracleParameter objects

An important point to note in Figure 2-4 is that the association between SqlCommand's and OradeCommand's Parameters properties is represented by a double arrow instead of a single arrow, signifying that it is a collection and not a singular object. Also, as you can see from Figure 2-4, just as the DbCommand.Parameters property points to DbParametersCollection, the SqlCommand.Parameters property points to a SqlParameterCollection, which holds a number of SqlParameters. Similarly, the OracleCommand.Parameters property points to an object of OracleParameterCollection type, which is a collection of OracleParameters.

Thus, in this manner, every .NET data provider that must implement a command object must have a Parameters property that holds a variable of type that inherits from DbParametersCollection, which is a collection of DbParameter objects.

After executing commands over a connection, the data fetched needs to be held in some kind of object in .NET code. You could write up your own business objects to represent the data or you could use the standard object that comes as a part of the .NET Framework, the DataSet object, to act as the disconnected cache of your data. Let's examine the structure and purpose

vered in depth in Chapter 6.

Holding Disconnected Data: DataSet

As you saw in Chapter 1, ADO.NET can be split into two major halves: the connected and the disconnected. The connection object, command object, rr ansaction object, and parameter object along with a few others that form a .NET data provider are objects that need to be connected with the underlying data source to work.

Commands can be split into three major categories: Data Definition Language (DDL), which is used to define the structure of the database; Data Manipulation Language (DML), which is used to run queries such as UPDATE, INSERT, and DELETE; and Data Query Language (DQL), wh ich is used to query data out of the database. An example of DQL is the SELECT command.

Once such a command is executed, frequently the results are not singular, but appear as a result set or maybe as a collection of result sets. A result set is simply tabular data that might contain one or more tables. The result set might be read in a connected fashion, using a DataReader object, which is described briefly later in this chapter and in detail in Chapter 5. Another way of reading the result set is to fill an object representation of the data and disconnect with the underlying data source. As you will see in Chapter 4, itÕs critical to open the connection as late as possible and close it as early as you can to facilitate better connection pooling. For now, itÕs enough to understand that connection pooling refers to the technique or process of sharing the valuable open connection with the data source between requests in an effort to significantly improve performance. Most providers in .NET will give you this facility by default; in other words, you donÕt have to write any code, but you need to follow the proper guidelines, as discussed in Chapter 4, to take advantage of connection pooling.

So, you need an object to hold disconnected data. Since the data is disconnected, its implementation doesnÕt have to be data source Ðspecific, as long as thereÕs some object in the middle acting as a bridge for you. In other words, the implementation of the object that holds disconnected data for you must nÕt be specific to a particular underlying data source, such as Microsoft SQL Server or Oracle. This bridge, or the sentry between the connected and disconnected worlds, is the DataReader object described later in this chapter.

As mentioned before, to hold disconnected data, you could write your own business object. While that approach is certainly possible, itÕs also quite possible that thereÕs some work involved in setting up the business objects, especially when you consider the fact that now you will be responsible for writing all the code to make operations such as data binding, state manage ment, row versions history, etc. possible. (These are covered in further depth in Chapter 6.) It can be argued, however, that itÕs possible to create a business object Ðbased architecture that saves you work in the long run or is simply a better architecture suited to your situation. There are various pros and cons to using a business object versus a DataSet, and these are covered in Chapter 14.

Note What is a business object? *Business object* is an object that abstracts the entities in the domain that the program is written to represent. In other words, itÕs a representation of something logical in your architecture. Say youÕre designing a system for an insurance company. The customer will probably understand what a policy is, or what a premium is, but the concept of a DataSet or a DataTable might be alien to him. The business objects would be representations of policy and premium entities that hold DataSets holding policy and premium information. The exact implementation of business objects is specific to your circumstance; however, good reference books for business objects are available (*Expert C# Business Objects*, Apress, 2004/2003).

You can use business objects or you can use the DataSet object and its various constituents. The DataSet object acts as a ready-to-use disconnected cache of data that ships with the .NET Framework. Even though Chapter 6 discusses DataSets and their various constituents in detail, Figure 2-5 gives an overview of the major data providerÐindependent disconnected objects involved in the DataSet object structure.

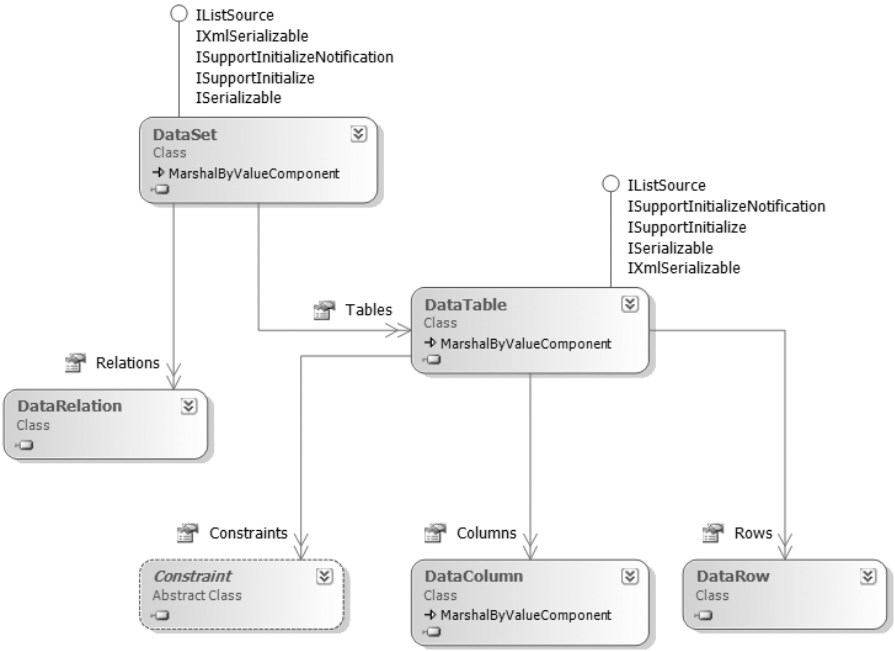


Figure 2-5. The DataSet and its various constituents such as DataTable, DataRelation, DataColumn, DataRow, and Constraint

The best way to understand a DataSetÕs structure is to draw a parallel with an RDBMS. However, I must point out that a DataSet is still an in-memory object. It shouldnÕt be confused or abused as an RDBMS. Its sole purpose is to do what an RDBMS cannot doÑprovide you with relational, structured data as a disconnected, portable, in-memory object.

So if a DataSet is the closest parallel to an RDBMS database, with tables and foreign-key relationships between them, then the DataTable object is the closest simile of a table and the DataRelation is the closest simile of a foreign-key constraint. Similarly, a column is the closest parallel to a DataColumn object and DataRow is the closest parallel to a row.

Thus, the DataSet object contains a collection of DataTable objects as a property called Tables, which is of DataTableCollection type. Also, it contains a property called Relations, which is of DataRelationCollection type that is a collection of DataRelation objects.

Similarly, the DataTable object has a property called Columns, which is a collection of DataColumn objects represented by the DataColumnCollection object. Also, it contains a DataRowCollection type property called Rows to represent the various rows as DataRow objects.

A DataTable may have constraints defined on itself (such as a UniqueConstraint), a collection of which is held by the Constraints property, which is of ConstraintCollection type that holds it from the Constraint object.

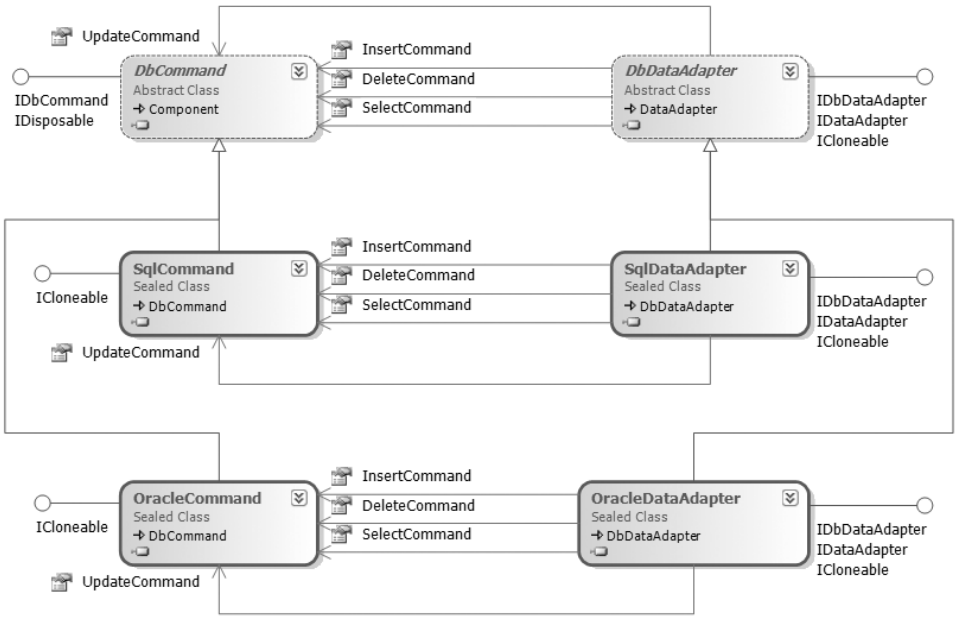


Figure 2-7. SqlDataAdapter and OracleDataAdapter—both inherit from DbDataAdapter

Also worth noting in Figure 2-7 are the two SqlCommand and OracleCommand classes, which both inherit from DbCommand. The data adapter needs various DbCommands to work. It can use up to four DbCommands as the InsertCommand, UpdateCommand, DeleteCommand, and SelectCommand properties for INSERT, UPDATE, DELETE, and SELECT operations. The base class for data adapters, DbDataAdapter, defines these four properties of DbCommand data type.

Following the trend of the rest of the ADO.NET connected objects, the specific data adapters, the SqlDataAdapter, OracleDataAdapter, etc., also provide you with four properties with the same names—InsertCommand, UpdateCommand, DeleteCommand, and SelectCommand—which take data provider–specific command objects such as SqlCommand and OracleCommand.

An important part of any programming architecture is error handling. .NET gives you a mechanism called *exception handling* to facilitate this task. ADO.NET, like any other architecture, contains a number of standard exceptions for various error conditions that may occur. Let's examine various exceptions available within ADO.NET that you should look out for.

Exceptions in ADO.NET

Errors happen. In the .NET Framework, they are thrown as exceptions that are segmented and protected callpaths, which allow developers to handle errors in a more robust manner. You can use a simple try...catch block to catch and act upon predefined possible error conditions. It is important to note, however, that exceptions are implemented as classes, and if your catch blocks are looking for a base-class exception before the specific inherited exception, you may never catch the specific exception. The specific or inherited exception will be masked by the base class.

For instance, in the .NET Framework, SqlTruncateException inherits from SqlTypeException, ere to write your exception-handling code

as shown in Listings 2-1 and 2-2 and a SqlTruncateException occurs, it will be caught in the first System.Exception block and the code inside the catch block for SqlTruncateException will never be executed.

Listing 2-1. Incorrect Way of Ordering Exception-Handling Blocks in C#

```
try
{
    ...
}
catch (Exception ex)
{
    ...
}
catch (SqlTypeException sqlTypeEx)
{
    ...
}
catch (SqlTruncateException sqlTruncateEx)
{
    ...
}
```

Listing 2-2. Incorrect Way of Ordering Exception-Handling Blocks in Visual Basic .NET

```
Try
    ...
Catch ex as Exception
    ...
Catch sqlTypeEx as SqlTypeException
    ...
Catch sqlTruncateEx as SqlTruncateException
    ...
End Try
```

The correct way to order exception-handling blocks is shown in Listings 2-3 and 2-4.

Listing 2-3. Correct Way of Ordering Exception-Handling Blocks in C#

```
try
{
    ...
}
catch (SqlTruncateException sqlTruncateEx)
{
    ...
}
catch (SqlTypeException sqlTypeEx)
```

```
    ...
}
catch (Exception ex)
{
    ...
}
```

Listing 2-4. Correct Way of Ordering Exception-Handling Blocks in Visual Basic .NET

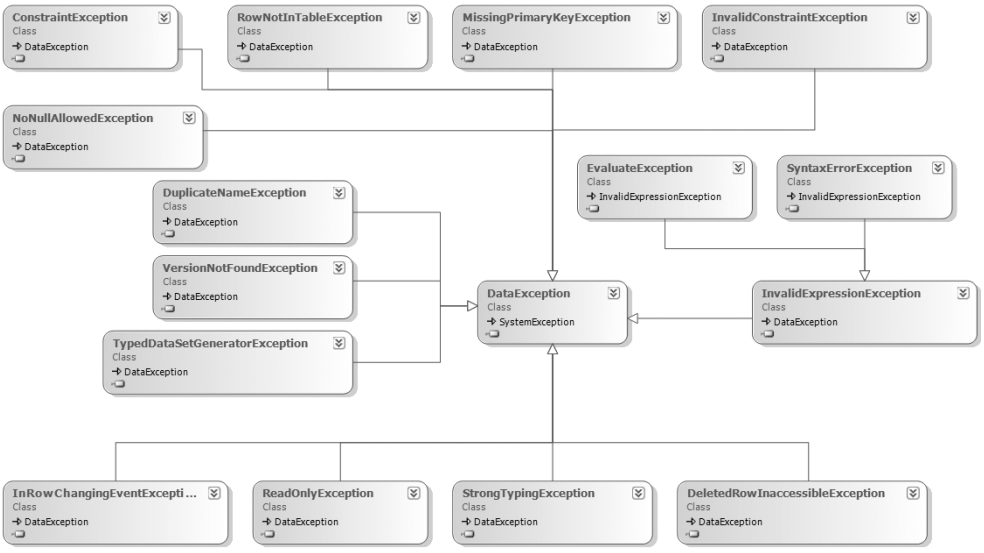
```
Try
    ...
Catch sqlTruncateEx as SqlTruncateException
    ...
Catch sqlTypeEx as SqlTypeException
    ...
Catch ex as Exception
    ...
End Try
```

Thus, to write effective exception-handling code, you have to understand the various exception classes that exist within ADO.NET and use them accordingly.

ADO.NET exceptions have been restructured in .NET 2.0. They can be categorized into four major categories:

¥ Disconnected stack exceptions inheriting from System.Data.DataException:

The various exceptions that may occur when working with disconnected data caches are shown in Figure 2-8.



CHAPTER 3

ADO.NET Hello World!

The first chapter explained the purpose of ADO.NET and where it fits in the overall architecture. It explained using common block diagrams, the very high-level structure of ADO.NET, the connected and disconnected portions of ADO.NET, and the .NET data provider model.

The second chapter took that discussion from a 30,000-ft. view to about a 10,000-ft. view where you saw the various objects, their inheritance layout within ADO.NET, the various namespaces, and the reasoning behind that structure.

ItÕs now time to walk on the ground and write a few real data-driven applications. But before you deal with the intricacies and complexities of a true enterprise-level data-driven architecture, it makes sense to see a few simple applications first. This is in the spirit of Ò crawl before you walk, walk before you run.Ó This chapter begins with extremely simple data-driven applications that require you to write absolutely no code at all. In fact, an entire working application is pr oduced by only dragging and dropping. Then, this approach is taken forward to increasingly more involved examples where youÕll blend some writing with dragging and dropping. YouÕll proceed to a small data-driven console application where youÕll write every part of the code yourself.

Since all the examples presented in this chapter will be data driven, it is probably a good idea to set up the data source being used first. The examples presented in this book exemplify various ADO.NET concepts using a local Microsoft SQL Server 2005 instance. Any differences with Oracle or other major providers will be identified as they arise. For the purposes of this chapter, however, the examples are simple enough that there are no differences between the major data providers.

Setting Up the Hello World Data Source

The quintessential Hello World example serves as a simple introduction to any programming concept. LetÕs try and leverage that to our advantage by setting up a simplistic data source. As mentioned earlier, the data source will be a database on the local running instance of Microsoft SQL Server 2005. The database name will be Test, and it will contain one table called Demo. This table can be set up using the following script:

```
Create Table Demo
(
DemoID int identity primary key,
DemoValue varchar(200)
)
```

In addition, there are some midway application architectures, such as ActiveX and ClickOnce. But for the purposes of this chapter, letÕs concentrate on the two major kinds of applications: web-based (ASP.NET) and Windows FormsÐbased.

LetÕs begin by looking at web-based applications first, or as they are referred to in the .NET platform, ASP.NET applications.

Drag and Drop in ASP.NET 2.0

The simplicity of creating an application using drag and drop in ASP.NET 2.0 is quite remarkable. The approach and underlying architecture are different from ASP.NET 1.1, which will not be discussed here.

The code for this example can be downloaded from the Downloads section for this chapter under DragDropWebsite, or you can easily create it yourself by following the steps here. Do note that because thereÕs no code to write, the instructions are exactly the same for both C# and VB.NET:

1. Start by firing up Visual Studio 2005 and creating a new website. To do this, select File New WebSite.

2. Give it a name; I chose DragDropWebsite, which is demonstrated in Figure 3-2.

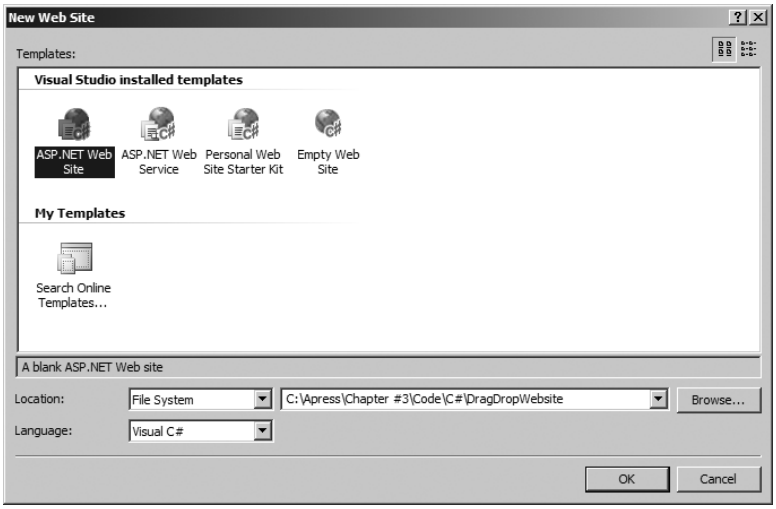


Figure 3-2. Creating a new ASP.NET 2.0 website

Take note that I am using C# in Figure 3-2 as my choice language. Since the steps are exactly the same for VB.NET, if you prefer, you can choose VB.NET as your choice language.

3. Click OK to create the website.

4. The created website should have one Default.aspx page created for you. If that page is not already open inside the Visual Studio IDE, double-click it to open it. At the bottom

.

5. Next, open the Database Explorer/Server Explorer window. If the window is not already visible, you can enable it by selecting View Server Explorer. This window should look similar to the one shown in Figure 3-3.

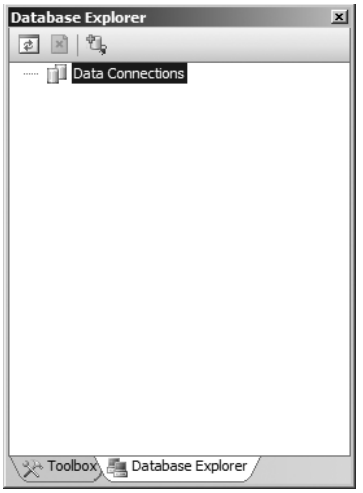


Figure 3-3. The Database Explorer/Server Explorer window

If your Database Explorer/Server Explorer window looks somewhat different than mineÑ donÕt panic. ItÕs only because your Server Explorer already has a few entries and mine is blank.

6. Next, right-click on Data Connections and choose Add Connection as shown in Figure 3-4.

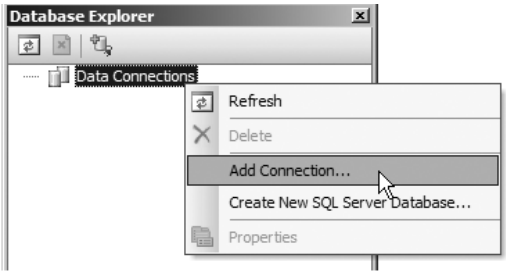


Figure 3-4. Adding a connection to the Database Explorer/Server Explorer

This should show a dialog box that prompts you to select the appropriate data source. At this point, you can go ahead and fill in the various values. When you click Test Connection, you should see a dialog box informing you that the connection is valid and the test succeeded, as shown in Figure 3-5.

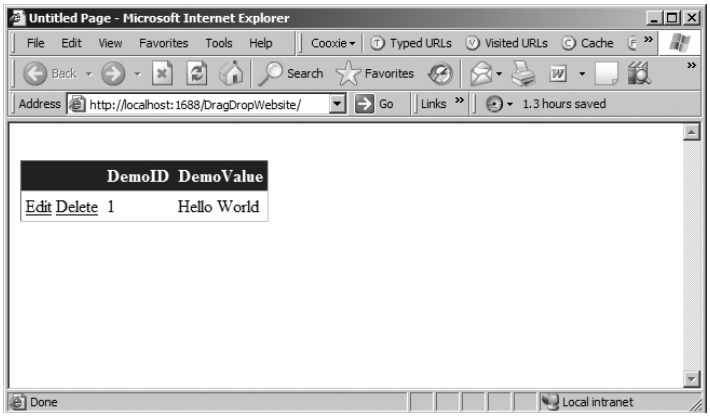


Figure 3-9. Hello World in ASP.NET

Try playing around with the application a bit. YouÕll notice that this is a fully functional application that even lets you modify the underlying data. If you do happen to modify the underlying data, you can easily restore it by running the script provided with this chapter again.

So you wrote absolutely no C# or VB.NET code and you have a data-driven web-based application ready. How did it all work? Well, the framework did all the work for you. It queried the underlying data source for you and encapsulated all that functionality within the various properties set on the GridView control and the SqlDataSourcecontrol.

If you view the source of the .aspx page and check out the listing for SqlDataSource, youÕll see that it looks like the code shown in Listing 3-1.

Listing 3-1. The SqlDataSource Control Defined on the Default.aspx Page

```
<asp:SqlDataSource ID="SqlDataSource1" runat="server"
    ConnectionString="<%$ ConnectionStrings:TestConnectionString1 %>"
    DeleteCommand="DELETE FROM [Demo] WHERE [DemoID] = @original_DemoID"
    InsertCommand="INSERT INTO [Demo] ([DemoValue]) VALUES (@DemoValue)"
    ProviderName="<%$ ConnectionStrings:TestConnectionString1.ProviderName %>"
    SelectCommand="SELECT [DemoID], [DemoValue] FROM [Demo]"
    UpdateCommand=
      "UPDATE [Demo] SET [DemoValue] =
      @DemoValue WHERE [DemoID] = @original_DemoID">
    <InsertParameters>
        <asp:Parameter Name="DemoValue" Type="String" />
    </InsertParameters>
    <UpdateParameters>
        <asp:Parameter Name="DemoValue" Type="String" />
        <asp:Parameter Name="original_DemoID" Type="Int32" />
    </UpdateParameters>
    <DeleteParameters>
        <asp:Parameter Name="original_DemoID" Type="Int32" />
    </DeleteParameters>
```

When you do see this window, as shown in Figure 3-11, click the Add New Data Source link.

Figure 3-11.The Data Sources window

3. When prompted to choose the Data Source type, select Database, as shown in Figure 3-12.

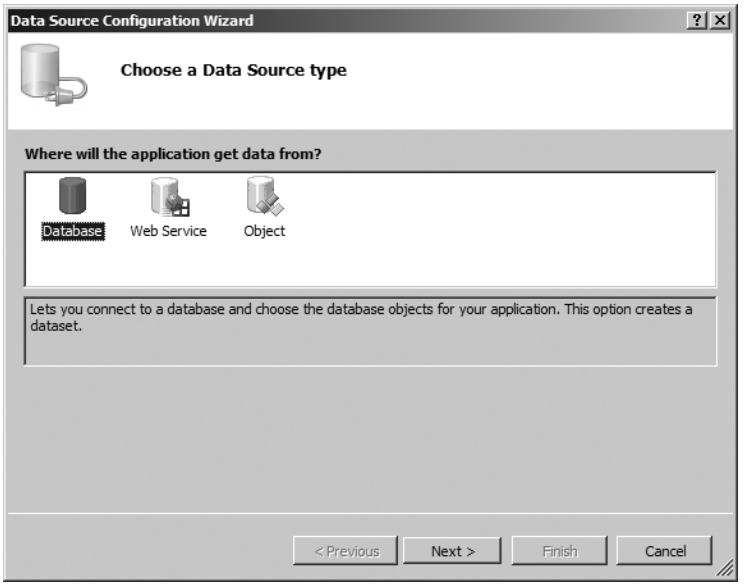


Figure 3-12.Choosing the Data Source type

4. When prompted to choose your data connection (see Figure 3-13), either choose the connection if it is already available in the list or click New Connection to add a new connection in a dialog box very much like Figure 3-5, which you saw in the ASP.NET

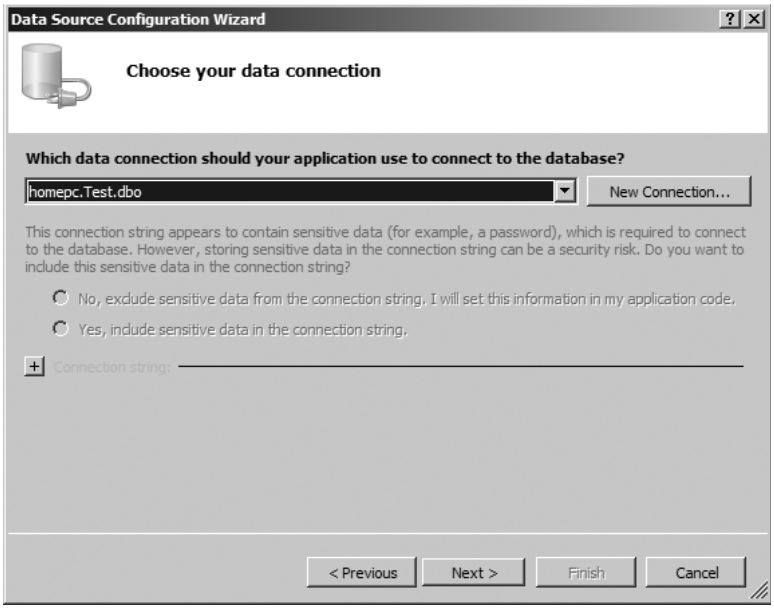


Figure 3-13.Choosing the data connection

5. When prompted, choose to save the connection string, as shown in Figure 3-14.

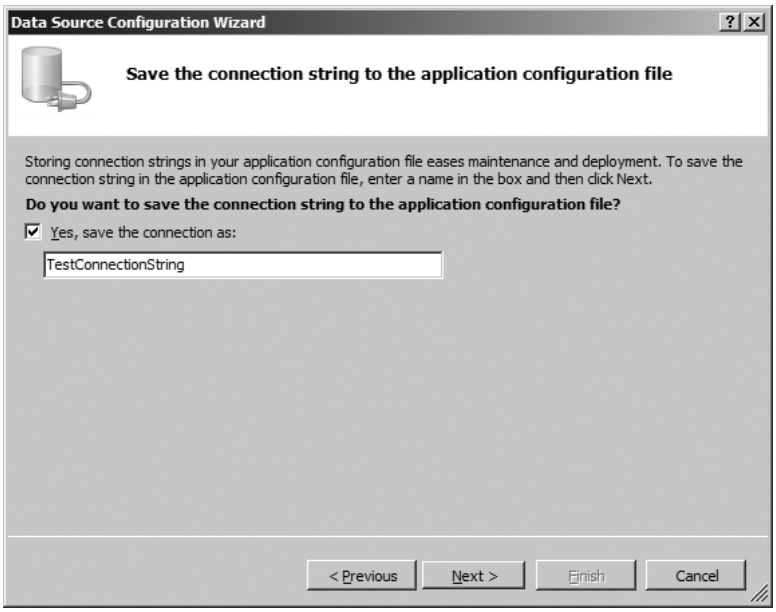


Figure 3-14.Choosing to save the connection string

6. When prompted to choose the database objects in your data source, choose the Demo table, as shown in Figure 3-15.

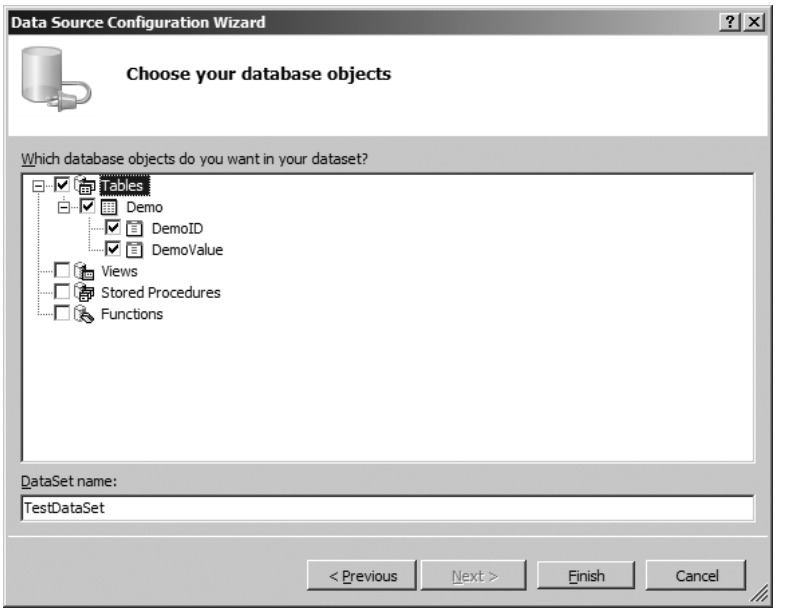


Figure 3-15.Choosing the Demo table to be a part of your data source

7. Click Next and Finish, which adds the TestDataSet data source to your application. At this point, you should see the TestDataSet data source added in the Data Sources window. If you select the Demo table under the data source, you can see a drop-down arrow next to it. For the purposes of this application, you can select DataGridView, as shown in Figure 3-16.

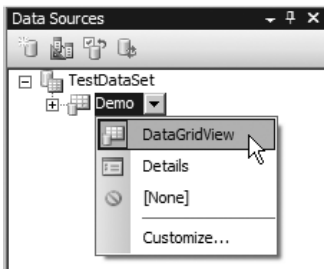


Figure 3-16.Configuring the data sourceÕs Demo table

8. Next, drag and drop the Demo table onto the surface of the form. This operation should add a number of controls to the surface of the form, as shown in Figure 3-17 (after

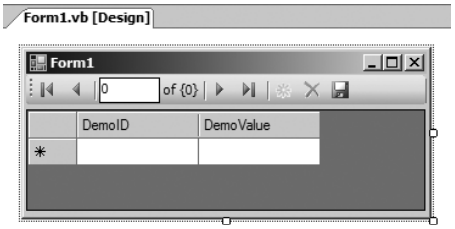


Figure 3-17.The form in Design mode after rearranging the autogenerated controls

This operation also adds a number of controls in the component tray under the form, as shown in Figure 3-18.

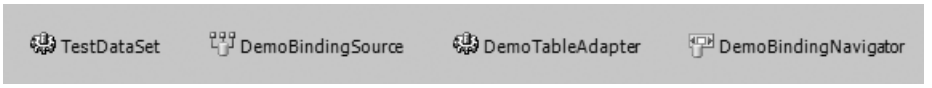


Figure 3-18.Various controls added for you in the component tray

9. ThatÕs it. Your data-driven application is ready. Compile and run it to see a fully operational window, as shown in Figure 3-19. YouÕll also see that you can edit the underlying data using this application. As an exercise to the reader, you can repeat this application by selecting something other than DataGridViewin step 7.

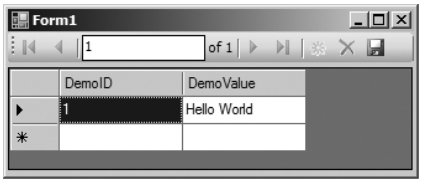


Figure 3-19.A fully running data-driven Windows Forms application, created using drag and drop

Again, you just created a fully functional data-driven application without actually having to write any code. As it turns out, in this case, the framework actually wrote some code for you. If you open the App.Config file, youÕll see that the application has saved the connection string as an element, as shown in Listing 3-4.

Listing 3-4. The Connection String in the App.Config File

```
<connectionStrings>
    <add name="DragDropWinApp.Settings.TestConnectionString"
     connectionString=
     "Data Source=(local);Initial Catalog=Test;Integrated Security=True"
        providerName="System.Data.SqlClient" />
```

As youÕll see in Chapter 9, the SQL Query is embedded deep inside the autogenerated code for TestDataSet. But letÕs leave that for Chapter 6. For now, letÕs look at a simple but purely write-yourself approach and create a simple application that connects to the data source and fetches the same results for you.

Data-Driven Application: The ÒWrite Code YourselfÓ Approach

In the last example, I posed a question: Where is the SQL Query?

I gave the answer along with the questionÑit is embedded deep inside autogenerated code. But why should a simple query such as that have to be embedded in so much code? It really doesnÕt have to be. As a matter of fact, when you do see the autogenerated queries in the strongly typed DataSet in Chapter 9, youÕll see that the queries take an extra-safe approach by comparing all columns and specifying highly inefficient, but accurate, UPDATE and DELETE queries. This is because the autogenerated code must work in all situations and canÕt make any assumptions as it is written in a one-size-fits-all approach.

Usually, in any application architecture, youÕll have to make a choice between performance, flexibility, and initial effort. Application architecture is a black art. Unfortunately or fortunately, you canÕt fully automate the process: unfortunately because it means more work for you and me, and fortunately because you and I will have jobs for a very long time. There are many instances where youÕll have to consciously decide and pick between various approaches. ThatÕs what this book intends to help you doÑgive you enough knowledge so you can make those decisions intelligently as an application architect. As a matter of fact, generally in a full-blown enterprise application, youÕll see yourself doing more hands-on work writing code yourself, rather than dragging and dropping.

Thus, itÕs important that the underlying data-access architecture gives you the ability, or fine-level control, to selectively choose what you need. Luckily, ADO.NET does give you this ability.

LetÕs look at a quick example that achieves the same results as the previous console application example, but this time around with no drag-and-drop help. You can download the code for this exercise from the associated code download, or you can easily create it using the following steps:

1. Create a new console application. Call it ConsoleApp2

2. Add the necessary using or Imports statements at the top of Program.cs or Module1.vb:

   C#
   ```
   using System.Data.SqlClient ;
   ```

   VB.NET
   ```
   Imports System.Data.SqlClient
   ```

3. LetÕs cheat here a bit. You need a connection string to connect to the data source. Connection strings have been mentioned briefly earlier in this chapter, but they will be covered in more detail in Chapter 4. For now just copy and paste the connection string from any of the previous examples. Create the connection string as a private string, so it is accessible to the rest of the code within the class, but not outside of it, like so:

CHAPTER 4

Connecting to a Data Source

ItÕs hard to imagine an application that doesnÕt have a data source. If a computer can be thought of as an information processing and storage machine, more often than not, the data storage for any serious application requires a dedicated data source. The data source could be as simple as a Microsoft Access database for a simple front-end point-of-sale application for your local convenience store, or it could be as complicated as a full-blown implementation of a leading commercial-quality database, such as Microsoft SQL Server or Oracle, for an e-commerce website.

Given the wide variety of applications that exist today, the demands on the database for a given application could range from a continuously connected user to multiple users seamlessly sharing connections using a connection pool.

Whether an application is simple or complicated, one fact remains: ADO.NET needs to be able to establish and manage a connection with a data source.

In this chapter, youÕll examine the various methods available within ADO.NET to connect to a database. YouÕll learn how to create, open, and close connections. Since, in the modern Internet-driven world, itÕs far too common to see applications that require a vast number of users requesting real-time interaction with the system, and given that connecting to the database is one of the most expensive operations a server-based application might need to do, most modern-day data access architectures have implemented mechanisms like connection pooling to address this problem. In this chapter, youÕll also examine how ADO.NET supports connection pooling and learn the proper way to close and destroy connection objects to take advantage of pooling.

The Ability to Connect

The need to connect to a data source is essential for any data access architecture. ADO.NET gives you the ability to connect to a data source by providing a suitable connection object, which is part of a data sourceÐspecific data provider. So the .NET Framework data provider for SQL Server would have a class by the name of SqlConnection that will allow you to establish a connection with a Microsoft SQL Server database, and the OracleClient data provider would have a class by the name of OracleConnection that will allow you to establish a connection to an Oracle database.

Whichever data source you are connecting to, there are a few common characteristics between connection implementations. Obviously, at the very least, you should be able to connect, or open, a connection. To open a connection, you need to specify what information

VB.NET

```
Dim testConnection As New SqlConnection
Dim testConnectionString As String = "Data Source=(local);" & _
    "Initial Catalog=Test;Integrated Security=SSPI"
testConnection.ConnectionString = testConnectionString
```

This code would prepare the connection object to open a connection to a SQL Server running on your local machine using Windows authentication and connect to a database called ÒTestÓ. Alternatively, you could use one of the constructors of the SqlConnection class to give us a ready-to-use SqlConnection object, as shown here:

C#

```
SqlConnection testConnection=
    new SqlConnection(
    "Data Source=(local);Initial Catalog=Test;Integrated Security=SSPI");
```

VB.NET

```
Dim testConnection As New SqlConnection( _
    "Data Source=(local);Initial Catalog=Test;Integrated Security=SSPI")
```

Thus, as long as you know the ConnectionString , you could have a ready-to-use SqlConnection object in as little as one line of code. Also, with the connection object prepared here, all you need to do now is call the Openmethod to successfully open a connection to the specified database.

Note Do note, however, that a connection object is not the same as a physical connection to the database. As youÕll see later in this chapter, if youÕre using connection pooling, which is turned on by default for ma providers, then calling Openon a connection object will either allow you to use an already open and unused physical connection, or open a brand new connection if sufficient open physical connections donÕt exist.

Using the connection strings shown in Listings 4-1 and 4-2, a quick example can be written that connects to a database called ÒTestÓ running on a local SQL Server instance. The examples use Windows authentication to connect to the database. They simply create the connection object, establish a connection, and check the state of the connection to verify if the connection was successfully opened or not, and finally they close the connection. The code can be seen in Listings 4-1 and 4-2 or can be downloaded as Exercise 4.1 from the Downloads section of the Apress website (http://www.apress.com).

Listing 4-1. Working with a Connection Object in C#

```
static void Main(string[] args)175

{
    SqlConnection testConnection =
                        new SqlConnection(
```

```
    try
    {
        testConnection.Open();
        if (testConnection.State == ConnectionState.Open)
        {
            Console.WriteLine("Successfully opened a  connection");
        }
    }
    catch (Exception)
    {
        if (testConnection.State != ConnectionState.Open)
        {
            Console.WriteLine("Failed to open a  connection");
        }
    }
    finally
    {
        // Closing a  connection ensures connection pooling.
        if (testConnection.State == ConnectionState.Open)
        {
            testConnection.Close();
        }
        testConnection.Dispose();
    }
}
```

Listing 4-2. Working with a Connection Object in Visual Basic .NET

```
Sub Main()
    Dim testConnection As SqlConnection = _
        New SqlConnection( _
        "Server=(local);Database=Test;Integrated Security=SSPI")
    Try
        testConnection.Open()
        If testConnection.State = ConnectionState.Open Then
            Console.WriteLine("Successfully opened a  connection")
        End If
    Catch
        If testConnection.State <> ConnectionState.Open Then
            Console.WriteLine("Failed to open a  connection")
        End If
    Finally
        ' Closing a  connection ensures connection pooling.
        If testConnection.State = ConnectionState.Open Then
            testConnection.Close()
        End If
        testConnection.Dispose()
    End Try
```

3. Open the Provider tab and choose the Microsoft Jet 4.0 OLE DB Provider.
4. Open the Connection tab, specify the properties needed (if any), such as user ID and password, and click Test Connection to verify that everything works. A message box indicating the test result will now display.
5. Click OK twice: first to close the message box and then to close the Data Link Properties dialog box. Your changes are now saved.

The myfile.udl file can now be viewed in Notepad and it should look like Listing 4-5. Please note that the listing has been formatted for display purposes.

Listing 4-5. myfile.udl File Content

```
[oledb]
; Everything after this line is an OLE DB initstring
Provider=Microsoft.Jet.OLEDB.4.0;Password="";
Data Source=C:\Apress\Mydb.mdb;Persist Security Info=True
```

ThatÕs it. Now you can copy and paste the connection string into your code and begin connecting to the Microsoft Access database as shown here:

C#

```
OleDbConnection testConnection = new OleDbConnection
    ("Provider=Microsoft.Jet.OLEDB.4.0;" +
    "Data Source=C:\\Apress\\mydb.mdb;Persist Security Info=False");
```

VB.NET

```
Dim testConnection As New OleDbConnection( _
    "Provider=Microsoft.Jet.OLEDB.4.0;" & _
    "Data Source=C:\Apress\mydb.mdb;Persist Security Info=False")
```

Even better, you can be ÒlazyÓ and directly specify the udl file as your connection string, like this:

C#

```
OleDbConnection testConnection = new OleDbConnection(
    "File Name=C:\\Apress\\myfile.udl");
```

VB.NET

```
Dim testConnection As New OleDbConnection( _
    "File Name=C:\Apress\myfile.udl")
```

While this technique works out of the box for OleDb or ODBC .NET data providers, thereÕs one little nuance you need to be careful of when using this technique with a specific .NET data provider such as SqlClient or OracleClient : When accessing a data source through the OleDb rovider used in the connection string,

such as Microsoft.Jet.OLEDB.4.0. When using the native .NET Framework data providers for SQL Server or Oracle, you don't need this piece of information, as the provider is fixed. Hence, you need to remove the Provider key-value pair from the generated connection string in order to use it with a SQL Server or Oracle database.

Securing Connection Strings

So far you've looked at the overall general architecture of the connection object. You saw how to create a connection object, and you saw a couple of easy ways to create a connection string.

While the examples shown so far work well, in a production application you would probably not want to hard code the connection string due to the following reasons:

- Connection strings often need to be changed after deployment. You don't want to recompile your application because someone changed a password or moved the database to a different server.
- .NET code can be disassembled and thus connection strings, including passwords, can be read.
- By storing the connection string in a common place, you ensure everyone uses exactly the same connection string. As you'll see later in this chapter, this ensures effective connection pooling. Effective connection pooling is critical to the performance of an application that needs to support multiple database requests. However, it's important to note that in ADO.NET physical connections are pooled at the client, not at the database server. Please see the "Connection Pooling" section later in this chapter for more information on this subject.

A common approach to this dilemma is to store the connection string in an XML-based configuration file instead, such as the Web.Configfile for ASP.NET Web-based applications.

Using the .NET Framework 1.x, you could specify the connection string as an appSetting element, so your configuration file could look somewhat like this:

```
<?xml version="1.0" encoding="utf-8"?>
    <configuration>
        <appSettings>
            <add key="connectString" value="...." />
        </appSettings>
    </configuration>
```

While this certainly works, you're stuck with implementing your own encryption mechanisms to protect your connection strings from prying eyes.

The .NET Framework 2.0 includes a useful security enhancement. The engineers at Microsoft realized that one of the most common uses of a configuration file was to store connection strings, so not only did they create a separate section in the configuration file for connection strings, but they also created an infrastructure to support encryption of configuration file sections. There is a new protectedData section in configuration files specifically for this purpose.

The usage is fairly simple: You simply need to add a protectedData element to your configuration file and add the relevant sections you wish to protect as protectedDataSections elements inside the protectedData element. In the protectedDataSections, you can specify which provider you wish to use in order to encrypt the particular section of your configuration file.

Listing 4-8. Saving Encrypted Data to a Configuration File in Visual Basic .NET

```
Dim config as Configuration = _
    ConfigurationManager.OpenExeConfiguration(ConfigurationUserLevel.None, "")
config.ConnectionStrings.ConnectionStrings.Add( _
    New ConnectionStringSettings( _
      "MyConnectionString", _
      "Server=local; Database=Test; Password=myPassword; User Id=myUser;") _
      )
```

As you can see, yet another neat feature of .NET 2.0 is that configuration files can be edited directly through code. This was not possible to do in .NET 1.1 unless you resorted to editing the configuration file using an `XmlDocument` object. If you ran this code as is, it would probably throw an exception informing you that the `KeySet` was not found. What this means is that you first need to set up either a machine-level or user-level `KeySet` on the machine this code is being executed upon. The easiest way to do this would be to run the following at the command line:

```
aspnet_regiis -pc "MyKeys" …exp
```

Now with everything set up, if you ran the code shown in Listings 4-7 and 4-8 and opened the configuration file after execution, it would look like the code shown in Listing 4-9. Also note that if you're executing the code as a `ConsoleApplication`, these changes won't make it to App.Config but to `ConsoleApplication1.exe.config`, where `ConsoleApplication1.exe` is the name of the console application executable.

Listing 4-9. Configuration File with Encrypted Section and Encrypted Data

```
<configuration>
    <connectionStrings>
        <EncryptedData Type="http://www.w3.org/2001/04/xmlenc#Element"
            xmlns="http://www.w3.org/2001/04/xmlenc#">
 <EncryptionMethod Algorithm="http://www.w3.org/2001/04/xmlenc#tripledes-cbc" />
            <KeyInfo xmlns="http://www.w3.org/2000/09/xmldsig#">
                <EncryptedKey Recipient="" xmlns="http://www.w3.org/2001/04/xmlenc#">
          <EncryptionMethod Algorithm="http://www.w3.org/2001/04/xmlenc#rsa-1_5" />
                    <KeyInfo xmlns="http://www.w3.org/2000/09/xmldsig#">
                        <KeyName>Rsa Key</KeyName>
                    </KeyInfo>
                    <CipherData>
                        <CipherValue>fquSeRPQvoa47qFzEys62yWC1VxNABD318DrCQc/hL6zLnuaG
                        GgQE6qxYSStHOccUntJ67HrDTjlpM0pRTxgXLLGzIq3vVLLMdKnRTE6eFAZcQe
                        pB7qBiK+PWuWTAcy4mFXfaHznPNiQNU4bDtkJCUO3j9FbLhUqeprSUCjOp1c=
                        </CipherValue>
                    </CipherData>
                </EncryptedKey>
            </KeyInfo>
            <CipherData>
                <CipherValue>b/6ILwoFPKGop5jyGQfbHAu00Q48M9JaHSYUJf1rTy4Tt1Kqr8qIY0x
```

In the next part of this chapter, youÕll examine what connection pooling is and how it is implemented in ADO.NET. YouÕll see how to tweak connection pooling to fit your requirements and also understand the best practices about how to use connection pooling in a typical application.

Connection Pooling

In a highly concurrent/highly availableÐbased application, itÕs important to realize that the majority of the time the user might hold an open connection and not actively use it because he is busy with other parts of the application. The application could essentially Òtime sliceÓ his expensive resourceÑan open connectionÑand pool it between multiple users.

Using connection pooling with ADO.NET is really simple because you donÕt have to do anything to use connection pooling with the default settings; instead, you have to turn it off explicitly should you decide not to use it. For instance, for SqlClient , if you donÕt wish to pool your connections, you simply add the following key-value pair to your connection string:

```
Pooling=false;
```

Similarly, you can tweak connection-pooling settings by designating specific key-value pairs on the connection string. These are shown for SqlConnection in Table 4-1. As you saw earlier, in order to connect to a database, you need to specify a connection string. ADO.NET maintains a pool of open connections internally for each connection string. In other words, it maintains a collection of pools using the connection string as the key. When you request an open connection via the Opencommand on a connection object, ADO.NET internally checks to see if thereÕs an unused physical database connection available. A physical database connection is ready to be pooled if no other user or portion of the application is actively using it. This means, for effective connection pooling, you must open as late as possible, and close as early as you can

The real picture is a little more involved because ADO.NET makes decisions based on the application load and might maintain more than one open physical connection concurrently if it receives too many requests simultaneously. Alternatively, the number of open connections in a connection pool might decrease if the number of requested open connections is too low. There are default numbers set for these connection-pooling parameters, but they are configurable via the connection string. Table 4-1 shows the various connection-pooling parameters that can be set using the connection string.

Table 4-1. SqlClient Connection Pooling Key-Value Pairs

Name	Definition
Connection Lifetime	This parameter is useful in clustered environments. A value specified here will result in the connection being destroyed if the current time minus the creation time exceeds the connection lifetime specified. This allows a new server, brought online in a clustered environment, to start sharing the load immediately. The default value of 0 indicates an unlimited c onnection lifetime.
Connection Reset	This causes the connection to be reset every time itÕs pulled out of the pool for use. Specifically for Microsoft SQL Server 7.0, setting this to false avoids an extra database trip. In addition, the connection state itself is not reset. The default value is true .

(Continued)

CHAPTER 5

Retrieving Data in a Connected Fashion

The data source for an application is typically a dedicated external resource; it could be a file or server software like Microsoft SQL Server or Oracle running on a dedicated server, accessed over a network connection.

In the last chapter you read about establishing a connection with such an external resource. Even though the data itself might reside in an external resource, i.e., the data source, the data is eventually mapped into object representations of the entities/types of the data source, and the logic to process upon the object representations of the data resides within the application itself. Therefore, to process data, the application needs to retrieve the parts of the data from the data source, which the application should process.

As you saw in Chapters 1 and 2, ADO.NET is split into two main halves: the connected and disconnected. This chapter will concern itself with the connected fashion of retrieving data, which involves connecting to the data source over an ADO.NET connection, sending commands, and retrieving the results while remaining connected to the database or external data source.

The need for an external data source arises because applications tend to be better at storing the logic that will process the data rather than storing the actual data. ItÕs not uncommon for an enterprise applicationÕs data to span several gigabytes, but usually, at a given time, the application is processing only a small subset of the data. Thus, the application needs a way to communicate with the data source requesting the subset of the data that itÕs interested in at a given time.

Communicating with the Data Source

After a connection has been made, communicating with the data source involves two operations:

1. Specifying what data the application is interested in.
2. Receiving results back.

Specifying what data the application is interested in involves sending a command/request via a predefined language or format. Even though ADO.NET doesnÕt limit you to a par ticular language or syntax, the most commonly accepted format of data source query language is

- It must execute against a databaseÑbut we didnÕt specify which connection this command will use to execute.
- We need to specify which action the command performsÑbut we didnÕt specify the command text (yet).

So at the very least, you need to specify the connection the command object should use and the action it should perform. LetÕs start with the first part of the information you must specifyÑthe database the command will execute against.

Which Database to Execute Against

In the exercises in this chapter, youÕll be connecting to a Microsoft SQL Server 2005 database named ÒTestÓ running on your local machine, using Windows authentication. This database will have one table called TestDemowith data, as shown in Figure 5-1.

	TestDemo	Description
1	1	One
2	2	Two
3	3	Three

Figure 5-1.TestDemo for examples in this chapter

Assuming that you have set up such a database on your machine, you can start writing your first exercise. You can either create the application by following the steps here, or you can find it in the Downloads section of the Apress website (http://www.apress.com) as Example 5.1:

1. Start up Visual Studio and create a new Console Application project named ÒExample 5.1.Ó
2. Open the Program.csclass file (C#) or the Module1.vbmodule file (VB.NET).
3. Import the relevant disconnected and connected namespaces:

   C#

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

   VB.NET

   ```
   Imports System.Data
   Imports System.Data.SqlClient
   ```

4. Create and set up the SqlConnection object in the Mainprocedure:

   C#

   ```
   string connectionString =
       "Data Source=(local);Initial Catalog=Test;Integrated Security=SSPI;"
   using (SqlConnection testConnection = new SqlConnection(connectionString))
   {
       // Code will be added here
   ```

Listing 5-4. Executing a Simple Command in Visual Basic .NET

```
Dim connectionString As String = _
    "Data Source=(local);Initial Catalog=Test;Integrated Security=SSPI;"

Using testConnection As New SqlConnection(connectionString)
    Dim testCommand As New SqlCommand( _
        "SELECT COUNT(*) FROM TestDemo", testConnection)
    testConnection.Open()
    Dim numResults As Integer = CInt(testCommand.ExecuteScalar())
    Console.WriteLine("Total number of rows in TestDemo: " & numResults)
    testConnection.Close()
End Using

Console.Read()
```

When you compile and run the application, it produces the following output:

```
Total number of rows in TestDemo: 3
```

As expected, the application correctly reports the number of rows in the `TestDemo` table.

Even though Example 5.1 creates a fully functional ADO.NET application, itÕs obviously not very complex. Usually, in querying data sources, the results to a query are in the form of rows and columns, which is also commonly referred to as a *result set*.

So now, letÕs create an application that helps you retrieve a result set.

Retrieving a Result Set

As you saw in the previous section, `ExecuteScalar` is good for retrieving a single scalar value from a `SELECT` statement. So the natural question is, ÒWhat can you do to retrieve an entire result set?Ó

A full result set is obviously the result of a command such as `SELECT * FROM TESTDEMO` specified as the command text for a command object. For an object to qualify as a valid ADO.NET command object, it needs to implement the `IDBCommand` interface. One of the methods that the `IDBCommand` interface requires you to implement is the `ExecuteReader` method. `ExecuteReader` is the method that gives you access to the multiple rows in a result set. As youÕll see later, `ExecuteReader` also gives you the ability to browse through multiple result sets using the `NextResult` method, which typically might be the result of a batched query (SQL Server) or multiple `REF CURSOR`s (Oracle).

`ExecuteReader` returns an object of `IDataReader` data type, and `IDataReader` allows you to iterate through the various rows and columns in a result set in a read-only/forward-only fashion.

The `IDataReader` interface also implements `IDataRecord` and `IDisposable` interfaces. As mentioned in Chapter 4, `IDisposable` requires you to implement a method called `Dispose` that ensures the freeing, releasing, or resetting of unmanaged resources. On the other hand, the `IDataRecord` interface represents an individual record in ADO.NET. So you can safely assume that `IDataRecord` is the minimum functionality that any row representation in ADO.NET must implement. Here are a few important methods `IDataRecord` implements:

```
    {
        Console.WriteLine("TestDemo: " + sqlDr.GetInt32(0)
            + " and Description : " + sqlDr.GetString(1));
    }
}
```

Listing 5-6. Reading the Results from a SqlDataReader in Visual Basic .NET

```
If sqlDr.HasRows Then
    While sqlDr.Read
        Console.WriteLine("TestDemo: " & sqlDr.GetInt32(0) & _
            " and Description : " & sqlDr.GetString(1))
    End While
End If
```

In this code, using the `HasRows` property, you first check and see if the data reader returned any rows in the one result set you expect.

If, indeed, there are rows available, you start iterating through those rows in a forward-only fashion by calling the `Read` method. This is because the cursor in the result set is placed at the front of the first row, so to read that row you have to call the `Read` method.

And while your data reader object `sqlDr` is focused on a particular row, you can retrieve various column values out of that row by calling the `GetInt32` and `GetString` methods.

When you compile and run the application, it produces the output shown in Figure 5-2.

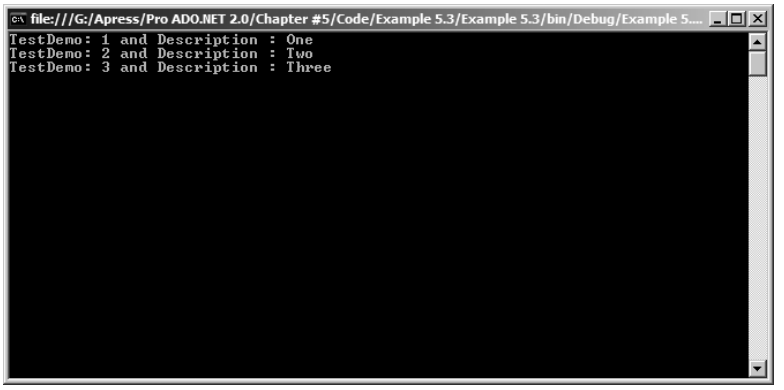


Figure 5-2. Results of using ExecuteReader to iterate through a result set

ItÕs important to note that the data reader gives you no way of going backward in the result set, nor does it give you any ability to edit the results. What you have is a read-only/forward-only method that is continuously connected to the database.

You might want to run a quick test on a larger result set being read into a `SqlDataReader`. Set a breakpoint after, say, the first execution of `SqlDataReader.Read`, and stop the underlying SQL Server and continue to iterate through the data reader. You might be surprised to learn that even with the database stopped, the `SqlDataReader` continues to retrieve results. Even though this behavior might appear surprising, it is, indeed, logical. The best way to imagine

```
            testCommand.ExecuteReader(CommandBehavior.CloseConnection)
        If sqlDr.HasRows Then
            For Each rec As DbDataRecord In sqlDr
                dbRecordsHolder.Add(rec) ' dbRecordsHolder is an ArrayList
            Next
        End If
    End Using ' testConnection.Dispose is called automatically
```

7. Finally, add another button to the form, call it btnDataBind, change its text to DataBind, and double-click it to add the following code:

C#

```
myDataGrid.DataSource = dbRecordsHolder;
```

VB.NET

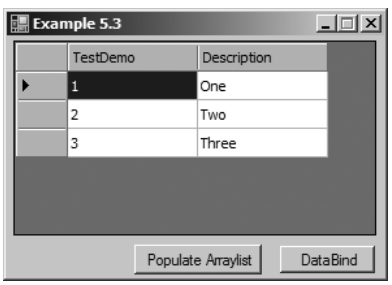
```
myDataGrid.DataSource = dbRecordsHolder
```

Let's examine what the code in Listings 5-7 and 5-8 does. It creates a command and fetches a data reader just like the console application you saw in Example 5.2; however, what is new this time around is that, instead of retrieving our data using the "Get" functions, you instead enumerated through various DbDataRecords and put those into an ArrayList called dbRecordsHolder. This ArrayList exists as a private variable in our class and is instantiated in the constructor as you can see in the associated code download. Since the ArrayList object is directly data-bindable, you can simply data bind it with myDataGrid as shown in step 7. Do note, however, that in an ASP.NET application, you can data bind the data reader directly.

Interestingly, if you now click Populate Arraylist, the ArrayList is filled up and the connection is closed because Dispose is called on the connection object. So at that point, you have all the data in the ArrayList, which is completely disconnected from the database. So what you're binding to the data grid is, indeed, disconnected from the database. The biggest advantage of this is that you didn't keep the SqlConnection open while your application was busy data binding with the UI.

The practical upshot of not having to keep the SqlConnection open while you were data binding is that you kept the connection open for as little time as possible. This results in much better connection pooling performance.

Compile and run the application. Click Populate DataGrid first and then DataBind. The output looks like that shown in Figure 5-4.



DataGridView

The DataReaderIsReady method is called once cmd.BeginExecuteReader is done with its processing. When you started the BeginExecuteReader, you passed in the command object. That object can now be retrieved using the IAsyncResult.AsyncState property. Once you have the SqlCommand object back, you can now get the prepared SqlDataReader by executing the EndExecuteReader method. From then on you can use the SqlDataReader as you had used it in the synchronous example you saw in Example 5.3.

The big difference, however, is that, while your underlying database (SQL Server in this case) is busy executing the query and preparing the results to send, your application is not tied up waiting to hear back from SQL Server. The framework lets you know when the results are ready, so you can begin listening at that time.

Note As you'll see in Chapter 13 when you write stored procedures in SQLCLR (the CLR inside SQL Server 2005), you have the option of sending the results in SqlContext.Pipe.ExecuteAndSend() or row by row (SqlContext.Pipe.SendResultsStart(), SqlContext.Pipe.SendResultsRow, and SqlContext.Pipe.SendResultsEnd()). This can be seen in the example SqlServerStoredProc in Chapter 13.

While ExecuteAndSend() works as a blocking call [i.e., the caller client application (data reader or data adapter) has to wait for the entire result set to be ready before it can access the first row], the SendResultsStart(), SendResultsRow(), and SendResultsEnd() methods allow your client application to begin reading the first row without having to wait for the entire result set to be ready first.

This technique, however, cannot be used when using T-SQL queries since SQL Server itself will not let SendResultsRow() execute unless the entire query has finished executing first. This is because SQL Server tries to minimize the time you hold contentious resources.

Just remember to close the connection as a good coding practice in the end. Do note, however, that you don't have the using block or code limited to a single method's scope to do this for you here, so you need to remember to do this cleanup.

3. A rather thoughtful addition to ADO.NET 2.0 and SQL Server 2005 is that any features that could potentially be misused, thanks to bad architecture or simply little knowledge, need to be consciously allowed before they can be used. As mentioned earlier, asynchronous command execution, although essential to an application, has the possibility of being misused or improperly used. So to use asynchronous commands, you have to consciously make the decision to do so and tell ADO.NET to use them via your connection string. A change needs to be made to the connection string to allow your code to use asynchronous commands—thus, add the following to your connection string:

   Asynchronous Processing=true

4. Let's compile and run the application now. The application looks exactly the same as Example 5.5 except now, when you click Populate Arraylist, your UI remains responsive. You could have even showed a nice animation while the data was being filled.

One thing you might want to do is show a progress bar as the results are being fetched. However, that's not possible because a progress bar has a minimum and a maximum. You know

ou have fully iterated through

Tip In all disconnected computing scenarios, you should always try and make your communication chunky, not chatty.

So, you need an easy way to retrieve multiple result sets in one single database hit. For this let's see an exercise that will query two tables—UserBasicInformation and PermissionsTable from the TestDemo database on your local Test database. This exercise can be created in the following steps or can be downloaded in the associated code download under Example 5.5:

1. Starting with the code from Example 5.2 in the associated code download (this is the example that demonstrates a data reader with one result set), change the command text specified to the SqlCommand as shown here:

   C#

   ```
   SqlCommand cmd =
     new SqlCommand("SELECT * FROM USERBASICINFORMATION" + ";" +
       "SELECT * FROM PERMISSIONSTABLE", conn);
   ```

   VB.NET

   ```
   Dim cmd as SqlCommand = _
     New SqlCommand("SELECT * FROM USERBASICINFORMATION" & ";" & _
       "SELECT * FROM PERMISSIONSTABLE", conn)
   ```

   As you can see, all that I did was concatenate the two command strings and put a ; character in the middle. This is commonly referred to as a "batched SQL command" in Microsoft SQL Server. In Oracle, even though batched queries are supported, they might be de-supported in the future. In Oracle, if you wish to return multiple result sets, instead you could create a stored procedure that returns multiple output REF CURSORs as shown here:

   ```
   CREATE OR REPLACE PACKAGE UserPermsPkg AS
     TYPE ResultCurr IS REF CURSOR;
     PROCEDURE GetUserPerms (UserCur OUT ResultCurr,
                             PermsCur OUT ResultCurr);
   END UserPermsPkg;

   CREATE OR REPLACE PACKAGE BODY UserPermsPkg AS
     PROCEDURE GetUserPerms (UserCur OUT ResultCurr,
                             PermsCur OUT ResultCurr)
     IS
       LocalUserCur ResultCurr;
       LocalPermsCur ResultCurr;
     BEGIN
       OPEN LocalUserCur FOR
           SELECT * FROM USERBASICINFORMATION;
   ```

```
        OPEN LocalPermsCur FOR
            SELECT * FROM PERMISSIONSTABLE;

        UserCur := LocalUserCur;
        PermsCur := LocalPermsCur;
    END GetUserPerms;
END UserPermsPkg;
/
```

Thus, with a stored procedure such as this, you could use OracleDataReader and work with multiple result sets in one OracleDataReader that you can iterate over using the NextResult method.

2. The next change involves reading multiple result sets out of the SqlDataReader. The SqlDataReader object has a method called NextResult. The NextResult method allows you to move to the next result set. If there are no more result sets, it will return a false. This change is shown in Listings 5-11 and 5-12.

Listing 5-11. Reading Multiple Result Sets Out of a Data Reader in C#

```
if (sqlDr.HasRows)
{
    do
    {
        Console.WriteLine("                ");
        while (sqlDr.Read())
        {
            Console.WriteLine(sqlDr.GetInt32(0)
                + " : " + sqlDr.GetString(1));
        }
    } while (sqlDr.NextResult());
}
```

Listing 5-12. Reading Multiple Result Sets Out of a Data Reader in Visual Basic .NET

```
If sqlDr.HasRows Then
    Do
        Console.WriteLine("                ")
        While sqlDr.Read
            Console.WriteLine(sqlDr.GetInt32(0) _
                & " : " & sqlDr.GetString(1))
        End While
    Loop While sqlDr.NextResult()
End If
```

Note that I checked sqlDr.NextResult in a do...while / Do...Loop While loop. The main reason for that is because, by default, SqlDataReader points to the first result set in your collection. By calling NextResult, you wouldÕve moved to the next result set. This kind of loop lets you check for this condition after the first iteration.

wn in Figure 5-7 is produced.

Figure 5-7. Results of iterating through multiple result sets

Thus, without executing multiple data readers, you have the ability to execute and fetch multiple data result sets in the same data reader. This is especially useful when working with disconnected data such as DataSets with more than one DataTable. That scenario will be covered in Chapters 6 and 7.

One interesting thing about all the examples you have seen so far is that the only data types you have fetched are the data types that are intrinsic to a given database. These are also referred to as inbuilt scalar types . You looked at querying a row, querying a result set, and querying multiple result sets, and in all these cases the data retrieved consisted of such inbuilt scalar types.

In object-oriented development (OOD), however, most business objects are represented via an object map representation. This hierarchical object map generally needs to be translated to and from a flat relational structure into a database, a process typically referred to as O/R mapping (object-relational mapping). While databases prefer to work with a relational structure, applications tend to prefer working with object representations of the data.

Thus, we have a mismatch.

Object-Oriented vs. Relational Representation

LetÕs assume a hypothetical situation: say you were creating a geographical-mapping application. Each point on a map is represented using an X,Y coordinate pair. How could you store this information in the database?

You could store this information as two integer columns in your database. But then you really donÕt have any good way of differentiating a coordinate with say, a kilometer-to-mile conversion lookup table, which also might store its data in a similar format. Not only that, any operations you may wish to do on the two coordinates would require you to read the two columns from the database and populate them into a business object or a class representation called XYCoOrdinate, in which you might implement commonly used functions such as distance from the center (0,0) coordinate.

But then, what if you wanted to execute a SQL query and find out which coordinate within a given set of coordinates is the furthest away from the center.

Another way of storing this data could be as a varchar column that stores the coordinates as X,Y, but again you would have to retrieve them from the database and convert them into an object representation if you really wanted to do any operation on it.

It would be rather nice if you could take the object and directly persist that into the data-

```
            Return Null
        End If
        Dim u As XYCoOrdinate = New XYCoOrdinate()

        Dim str As String = s.ToString().Trim()
        Dim commaLocation As Integer = str.IndexOf(",")
        Try
            u.X = CInt(str.Substring(0, commaLocation))
            u.Y = CInt( _
                str.Substring(commaLocation + 1, str.Length - commaLocation - 1))
        Catch ex As Exception
            Throw New ApplicationException( _
             "Error converting " + str + " to a  co-ordinate.", ex)
        End Try
        Return u
    End Function
End Structure
```

LetÕs quickly cover the basics that allow you to convert the autogenerated code to a working UDT. At a first glance, youÕll notice the following method stubs created for you:

- ToString() : This is used when you use the UDT in a SQL query. TheToString() representation of the object is displayed in the SQL Server Management Studio results window.
- IsNull, Null : These help the mismatch of null handling between databases and .NET data types. You can write logic here to inform the database when your object should be interpreted as a null, or what its null representation means.
- Parse: This method is used to perform a translation between a scalar type and the UDT itself. This method is valuable because it allows you to insert into a UDT by specifying an instantiation scalar value via a simple SQL command.

An entire chapter could be written about UDTs alone, but since this is an ADO.NET book and not a SQL Server 2005 book we will not go into the depths of writing a UDT.

4. Once the UDT is compiled into a DLL, it can be easily registered into the SQL Server. You need to register the assembly in the SQL Server. This can be done using the following script:

   ```
   Create Assembly XYCoOrdinate FROM 'C:\Apress\UDT.dll'
   ```

5. Once the assembly is added in SQL Server, you can register the UDT as a type from the previous registered assembly. This can be done using the following script:

   ```
   Create Type XYCoOrdinate External Name UDT.XYCoOrdinate
   ```

6. With the UDT created, create a table that uses the previous UDT:

   ```
   Create Table MyTest(
   TestColumn XYCoOrdinate
   )
   ```

CHAPTER 6

DataSets

In the last chapter you examined the connected data access mode. You examined various objects and examples that comprise the connected portion of ADO.NET. That is the most logical method of data access, where you connect, interact, and disconnect.

In Chapter 4 where you also read about ADO.NET connections in depth, you also looked at the possibility of a high load, highly available system. Usually in such a system there are multiple users vying for the same valuable resource: the data source connection. You saw an example where such a valuable resource could be pooled in effect to give you as much as tens of times performance benefit in a multiuser, highly concurrent, highly available system.

For such pooling to work, individual users need to be respectful of a shared common resource, such as the data source connection, by releasing it as soon as they can and acquiring it as late as they can. As youÕll see in this chapter, for such data access architecture to work you have to query as much data as you practically can, work in a disconnected fashion, and then reconnect to the data source and persist your changes while checking for concurrency at that time. You need to do this not only for connection pooling reasons, but also because disconnected data access is critical in scenarios such as distributed systems, low bandwidth connections, delegation of responsibility, etc.

In this chapter, youÕll examine the centerpiece of ADO.NETÕs disconnected architecture, the DataSet. The DataSet is the object that allows you to load a portion of the data source and acts as the bucket of data that gives you a familiar relational structure of a disconnected nature and data sourceÐindependent data storage mechanism. While making it intuitive and easy to work on the data, the DataSet also maintains a history of up to one change per cell, thus making concurrency and persistence of changes back to the data source easier.

But first, letÕs see why it is critical to have a disconnected model.

The Case for the Disconnected Model

Everyone likes a bedtime story, so let me start this chapter with one.

Jon is a busy executive. Most of his work is on his desktop. His desktop is a powerful computer with lots of hard disk space. He has many Excel sheets and Word documents that he needs to work on. Frequently, Jon goes to the boardroom for meetings. The first time Jon went to such a meeting, he thought he could remain ÒconnectedÓ with his desktop and make frequent roundtrips to his desktop and bring any documents he might need to the meetings. In such a situation, Jon would begin discussing something, and frequently he would have to leave the meeting and run back to his desktop and fetch the documents he needed. While this certainly

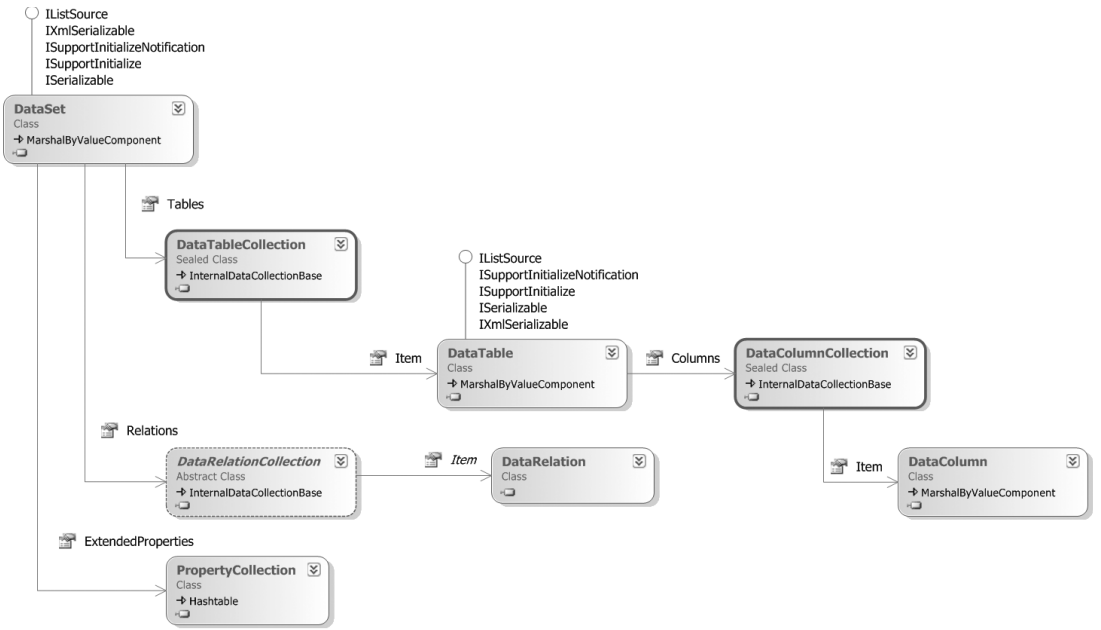
In Chapter 5, you saw an ArrayList being filled in with DbDataRecords in Examples 5.3 and 5.4. For the same amount of data involved, an ArrayList is much lighter weight than a DataSet is, but such an object does not satisfy the last two desirable characteristics just mentioned. In some situations, though, that might be a better choice. For example, if you don't need XML flexibility, or you're working with a read-only system [such as the presentation portion of a content management website, or maybe you intend to re-query the database at update to maintain concurrency checks (more on this in Chapters 9 and 10)], you might want to choose an ArrayList as your data object representation.

However, if an ArrayList does not satisfy the requirements of your architecture, you could create your own business object representation that satisfies all three conditions, or even more if your architecture demands it. Alternatively, you could use the DataSet object that comes as a part of ADO.NET that satisfies all three conditions and more.

As an example, when you're working with disconnected data, one of the things you'll need to worry about is being able to maintain an updated, refreshed copy of your data. The DataSet has inbuilt mechanisms, such as being able to extract changes or merge with another DataSet, that allow you to write code to avoid stale data. While it's possible to implement something like that in another object, such as an ArrayList, you would have to start from zero and put in relatively more work when compared with the DataSet.

The DataSet Object Model

Chapter 3 presented an abbreviated picture of the object model that springs from the DataSet object. Let's add some flesh to those bones and look at the bigger picture, shown in Figure 6-1. A more complete idea of the model will help you understand the power and usefulness of DataSets.



- Add(myDataColumn): Adds the specified, preexisting DataColumn object to the DataColumnCollection.
- Add("SubTotal", Type.GetType("System.Single"), "Sum(Price)"): Creates and adds a DataColumn with the specified name, data type, and Expression property. The expression can be used to filter rows, to calculate the values in a column, or (as in this case) to create an aggregate column.

The following code snippet creates a new DataTable object through one of its three constructors (the other two allow for the creation of a table with a default name, and specify both a table name and a table namespace). Once created, it then defines the table's schema by creating three new columns in the Columns collection:

C#

```
// Create a new DataTable
DataTable productsTable = new DataTable("Products") ;
// Build the products schema
productsTable.Columns.Add("ID", typeof(System.Int32)) ;
productsTable.Columns.Add("Name", typeof(System.String)) ;
productsTable.Columns.Add("Category", typeof(System.Int32)) ;
```

VB.NET

```
' Create a new DataTable
Dim productsTable As New DataTable("Products")
' Build the Products schema
productsTable.Columns.Add("ID", GetType(System.Int32))
productsTable.Columns.Add("Name", GetType(System.String))
productsTable.Columns.Add("Category", GetType(System.Int32))
```

To build the schema for this new table, you can call the Add() method of the DataTable.Columns collection once for each column that you want to add to the DataTable. For each DataColumn, you can pass in arguments for the ColumnName and DataType properties. The result is a DataTable named "Products" that's made up of three columns named "ID", "Name", and "Category" of data types Int32, String, and Int32, respectively.

DataRow

With the DataTable constructed and the columns defined, you can now begin populating the DataTable with data. This process involves adding new DataRow objects to the DataTable.Rows collection, which is of DataRowCollection type. To create a new row in the DataTable, you need to first invoke the DataTable.NewRow() method, which returns a DataRow that conforms to the DataTable's current schema. Next, you can set the value of each column in the DataRow before calling the DataTable.Rows.Add() method and passing the new DataRow object as the only argument:

C#

```
// Create a new DataRow with the same schema as the DataTable
DataRow tempRow = productsTable.NewRow() ;
```

Listing 6-3. Filling the productsTable in C#

```
DataRow tempRow;
// Populate the Products table with 10 cars
for (int i = 0; i < 10; i++)
{
    tempRow = productsTable.NewRow();
    // Make every even row Caterham Seven de Dion
    if (Math.IEEERemainder(i, 2) == 0)
    {
        tempRow["Name"] = "Caterham Seven de Dion #" + i.ToString();
        tempRow["Category"] = 1;
    }
    else
    {
        tempRow["Name"] = "Dodge Viper #" + i.ToString();
        tempRow["Category"] = 2;
    }
    productsTable.Rows.Add(tempRow);
}
productsTable.WriteXml("productsTable.xml") ;
```

Listing 6-4. Filling the productsTable in Visual Basic .NET

```
Dim tempRow As DataRow

' Populate the Products table with 10 cars
Dim i As Int32 = 0
For i = 0 To 9
   tempRow = productsTable.NewRow()

   ' Make every even row a  Caterham Seven de Dion
   If Math.IEEERemainder(i, 2) = 0 Then
      tempRow("Name") = "Caterham Seven de Dion #" & i.ToString()
      tempRow("Category") = 1
   Else
      tempRow("Name") = "Dodge Viper #" & i.ToString()
      tempRow("Category") = 2
   End If

   productsTable.Rows.Add(tempRow)
Next i
productsTable.WriteXml("productsTable.xml")
```

The interesting thing about this code, compared to what you had earlier, is that you donÕt need to set a value for the ID columnÑitÕs a read-only, auto-increment column. This can be seen in the contents of productsTable.xml shown in Listing 6-5.

Listing 6-6. Setting Up Various Event Handlers on a DataTable in C#

```
productsTable.ColumnChanged +=
    new DataColumnChangeEventHandler(productsTable_ColumnChanged);
productsTable.ColumnChanging +=
    new DataColumnChangeEventHandler(productsTable_ColumnChanging);
productsTable.RowChanged +=
    new DataRowChangeEventHandler(productsTable_RowChanged);
productsTable.RowChanging +=
    new DataRowChangeEventHandler(productsTable_RowChanging);
productsTable.RowDeleted +=
    new DataRowChangeEventHandler(productsTable_RowDeleted);
productsTable.RowDeleting +=
    new DataRowChangeEventHandler(productsTable_RowDeleting);
```

Listing 6-7. Setting Up Various Event Handlers on a DataTable in Visual Basic .NET

```
AddHandler productsTable.ColumnChanged, AddressOf productsTable_ColumnChanged
AddHandler productsTable.ColumnChanging, AddressOf productsTable_ColumnChanging
AddHandler productsTable.RowChanged, AddressOf productsTable_RowChanged
AddHandler productsTable.RowChanging, AddressOf productsTable_RowChanging
AddHandler productsTable.RowDeleted, AddressOf productsTable_RowDeleted
AddHandler productsTable.RowDeleting, AddressOf productsTable_RowDeleting
```

Next, you need to write implementations for the various event handlers though Visual Studio might have already created the stubs for you. This can be seen in Listings 6-8 and 6-9.

Listing 6-8. Implementations for the Event Handlers in C# [1]

```
private static void productsTable_ColumnChanged
    (object sender, DataColumnChangeEventArgs e)
{
        Console.WriteLine("productsTable_ColumnChanged");
        Console.WriteLine("   Value: " + e.Row["Name"].ToString());
        Console.WriteLine("   RowState: " + e.Row.RowState.ToString());
}

private static void productsTable_ColumnChanging
    (object sender, DataColumnChangeEventArgs e)
{
        Console.WriteLine("productsTable_ColumnChanging");
        Console.WriteLine("   Value: " + e.Row["Name"].ToString());
        Console.WriteLine("   RowState: " + e.Row.RowState.ToString());
}
```

1. Here and elsewhere in the book, the event handlers are marked static only because the main class . This is not otherwise required.

```
private static void productsTable_RowChanged
    (object sender, DataRowChangeEventArgs e)
{
        Console.WriteLine("productsTable_RowChanged");
        Console.WriteLine("   Value: " + e.Row["Name"].ToString());
        Console.WriteLine("   RowState: " + e.Row.RowState.ToString());
}

private static void productsTable_RowChanging
    (object sender, DataRowChangeEventArgs e)
{
        Console.WriteLine("productsTable_RowChanging");
        Console.WriteLine("   Value: " + e.Row["Name"].ToString());
        Console.WriteLine("   RowState: " + e.Row.RowState.ToString());
}

private static void productsTable_RowDeleted
    (object sender, DataRowChangeEventArgs e)
{
        Console.WriteLine("productsTable_RowDeleted");
        Console.WriteLine("   RowState: " + e.Row.RowState.ToString());
}

private static void productsTable_RowDeleting
    (object sender, DataRowChangeEventArgs e)
{
        Console.WriteLine("productsTable_RowDeleting");
        Console.WriteLine("   RowState: " + e.Row.RowState.ToString());
}
```

Listing 6-9. Implementations for the Event Handlers in Visual Basic .NET

```
Private Sub productsTable_ColumnChanged(ByVal sender As Object, _
  ByVal e As DataColumnChangeEventArgs)
    Console.WriteLine("productsTable_ColumnChanged.")
    Console.WriteLine("Value: " & e.Row("Name").ToString())
    Console.WriteLine("RowState: " & e.Row.RowState.ToString())
End Sub

Private Sub productsTable_ColumnChanging(ByVal sender As Object, _
  ByVal e As DataColumnChangeEventArgs)
    Console.WriteLine("productsTable_ColumnChanging.")
    Console.WriteLine("Value: " & e.Row("Name").ToString())
    Console.WriteLine("RowState: " & e.Row.RowState.ToString())
End Sub

Private Sub productsTable_RowChanged(ByVal sender As Object, _
```

If you try deleting a row using

```
productsTable.Rows[0].Delete() ;
```

the following output is produced:

```
productsTable_RowDeleting
    RowState: Modified
productsTable_RowDeleted
    RowState: Deleted
```

Hence, as you can see, the ColumnChanging, RowChanging, and RowDeleting events fire before the actual change has occurred, and these can be used to validate or even reject the proposed change. Whereas, the ColumnChanged, RowChanged, and RowDeleted events fire after the actual change has occurred and can thus be used for cleanup purposes.

Note Do closely note, however, that there is a minute difference between the ColumnChanging and RowChanging. ColumnChanging occurs before the actual value change, and RowChanging occurs before the RowState change.

As an exercise, you could also add event handlers for TableClearing and TableCleared events, and run those with the DataTable.Clear() method. You'll see similar behavior there.

You could also try your hand at a new event introduced in .NET 2.0: the TableNewRow event. This is where you could include business logic in the creation of a new row. While all other events allow you to work on a row that is being added, deleted, or modified when currently in the Row collection of the table, this event allows you to modify its values based on custom business rules at its creation. A good practical use for such an event is a row generator class that inherits from DataTable. You might have written such a class to encapsulate your business logic, while generating a plain old DataRow that can be easily sent to the DataLayer as a part of the DataTable.

Relational Data

If a DataSet is like having a mini RDBMS in memory, a DataTable is the closest logical equivalent of a table in such a database. Most databases also support the concept of being able to create foreign-key constraints, which can be used to define relations between tables. As you can see in the object model in Figure 6-1, a DataSet consists of a Tables collection (DataTableCollection), and a Relations collection (DataRelationCollection). Thus, a DataSet can be referred to as a collection of Tables and Relations. The minimum number of either Tables or Relations that exist within a DataSet can be zero.

The Relations Collection

The DataSet.Relations property is an instance of the DataRelationCollection class. The Relations collection contains DataRelation objects, which are used to create parent-child relationships between DataTables in the DataSet. A primary key–foreign key relationship is an example of this type of relation.

We can create a relation between two tables in a DataSet by invoking the DataSet.Relations.Add() method. There are seven overloaded Add() methods:

- Add(DataRelation) : Adds the specified DataRelation object to the collection.
- Add(DataColumn, DataColumn) Creates a DataRelation object in the collection based on the two DataColumns. The first argument is the parent column, and the second is the child column. The DataRelation is given a default name.
- VB.NET: Add(DataColumn(), DataColumn()) or C#: Add(DataColumn[], DataColumn[]) : Creates a DataRelation object in the collection based on the two DataColumn arrays. The first argument is the parent column array, and the second is the child column array. The DataRelation is given a default name.
- Add(String, DataColumn, DataColumn) As DataRelation : Creates a DataRelation object in the collection with the specified string as the DataRelation.RelationName property.
- VB.NET: Add(String, DataColumn(), DataColumn()) or C#: Add(String, DataColumn[], DataColumn[]): Creates a DataRelation object in the collection with the specified string as the DataRelation.RelationName property.
- Add(String, DataColumn, DataColumn, Boolean) : Creates a DataRelation object in the collection with the specified string as the DataRelation.RelationName property. The DataRelation is based on the two DataColumns. The Boolean argument indicates whether to create constraints (the default setting is True).
- VB.NET: Add(String, DataColumn(), DataColumn(), Boolean) or C#: Add(String, DataColumn[], DataColumn[], Boolean) : Creates a DataRelation object in the collection with the specified string as the DataRelation.RelationName property. The DataRelation is based on the two DataColumn arrays. The Boolean argument indicates whether to create constraints.

You can create a relationship in a DataSet using any of these overloaded methods. HereÕs an example using the overload that accepts a relation name, and the two DataColumns that form the relation:

C#

```
//Create a relation between Customers and Orders.
myDataSet.Relations.Add("CustomersToOrders",
    myDataSet.Tables["Customers"].Columns("CustomerID"],
    myDataSet.Tables["Orders"].Columns["CustomerID"]) ;
```

VB.NET

```
' Create a relation between Customers and Orders
myDataSet.Relations.Add("CustomersToOrders", _
    myDataSet.Tables("Customers").Columns("CustomerID"), _
    myDataSet.Tables("Orders").Columns("CustomerID"))
```

This code invokes the Add() method of the myDataSet.Relations object to create a new DataRelation object in the myDataSet.Relations collection. As a result, a new UniqueConstraint is added to the Orders

The DataSet that will be created in this example has two tables in it. These are described in Tables 6-8 and 6-9.

Table 6-8. The Animal Table

Property	Description
AnimalID	This is the primary key.
AnimalName	String representation of the userÕs name.

Table 6-9. The Pets Table

Property	Description
PetID=	This is the primary key.
AnimalID	Has a foreign-key constraint with AnimalID.
PetName	The name of the pet.

As indicated, AnimalID serves as a primary key for the Animal table and as a foreign-key constraint between the Pets and Animal tables.

Example 6.3 provides you with a Windows Form that looks like what is shown in Figure 6-2.

Figure 6-2. Example 6.3Õs user interface

As you can see, there are two buttons: Create DataSet and DataBind, and Show XML. Create DataSet and DataBind creates the bare DataSet schema, and binds the two tables to their individual DataGridViews. Such a DataSet can be set up using the code shown in Listings 6-10 and 6-11. You can find this code in the CreateSchema method in Example 6.3 of the associated

```
myDataColumn = New DataColumn("PetID", GetType(System.Int32))
myDataColumn.AutoIncrement = True
petsTable.Columns.Add(myDataColumn)

myDataColumn = New DataColumn("AnimalID", GetType(System.Int32))
petsTable.Columns.Add(myDataColumn)

myDataColumn = New DataColumn("PetName", GetType(System.String))
petsTable.Columns.Add(myDataColumn)

Dim toReturn As DataSet = New DataSet("petsData")
toReturn.Tables.Add(AnimalTable)
toReturn.Tables.Add(petsTable)
toReturn.Relations.Add( _
    New DataRelation("AnimalsPets", _
    AnimalTable.Columns("AnimalID"), petsTable.Columns("AnimalID")))

Return toReturn
```

It then data binds it with the appropriate grids, as shown in Listings 6-12 and 6-13.

Listing 6-12. Data Binding petsData in C#

```
petsData = CreateSchema();
petsGrid.DataSource = petsData.Tables["Pet"];
AnimalsGrid.DataSource = petsData.Tables["Animal"];
```

Listing 6-13. Data Binding petsData in Visual Basic .NET

```
petsData = CreateSchema()
petsGrid.DataSource = petsData.Tables("Pet")
AnimalsGrid.DataSource = petsData.Tables("Animal")
```

Show XML simply shows the contents of the DataSet as XML. This can be done using either WriteXml or GetXml. Since GetXml directly returns the DataSet as an XML string, let's use that instead:

C#

```
MessageBox.Show(petsData.GetXml(), "DataSet Contents");
```

VB.NET

```
MessageBox.Show(petsData.GetXml(), "DataSet Contents")
```

Finally, there is a checkbox that sets the nested property on the relation:

C#

```
if (petsData != null)
{
    petsData.Relations[0].Nested = nestedRelation.Checked;
```

VB.NET

```
If Not petsData Is Nothing Then
    petsData.Relations(0).Nested = nestedRelation.Checked
End If
```

Once this code is put together and the example is run, it produces a Windows Form as shown in Figure 6-2. Try adding some data as shown in Figure 6-3.

Figure 6-3. Example 6.3 in action

In editing the DataSet, you'll notice that

- The primary keys PetID and AnimalID are automatically generated for you.
- If you try to enter data that violates the relation set up for you, you'll get a System.Data.InvalidConstraint and the form will not let you enter the invalid data.

You can view the DataSet's contents at any time by clicking Show XML. One curious thing to note, however, is the difference between the XML generated when the relation's Nested property is set to true versus when it is set to false (see Listings 6-14 and 6-15).

Listing 6-14. Contents of the DataSet with the Nested Property Set to false

```
<petsData>
  <Animal>
    <AnimalID>0</AnimalID>
    <AnimalName>Dog</AnimalName>
  </Animal>
  <Animal>
    <AnimalID>1</AnimalID>
```

```
  </Animal>
  <Animal>
    <AnimalID>2</AnimalID>
    <AnimalName>Goldfish</AnimalName>
  </Animal>
  <petsTable>
    <PetID>0</PetID>
    <AnimalID>0</AnimalID>
    <PetName>Tashu</PetName>
  </petsTable>
  <petsTable>
    <PetID>1</PetID>
    <AnimalID>0</AnimalID>
    <PetName>Campy</PetName>
  </petsTable>
  <petsTable>
    <PetID>2</PetID>
    <AnimalID>1</AnimalID>
    <PetName>Jimmy</PetName>
  </petsTable>
</petsData>
```

Listing 6-15. Contents of the DataSet with the Nested Property Set to true

```
<petsData>
  <Animal>
    <AnimalID>0</AnimalID>
    <AnimalName>Dog</AnimalName>
    <petsTable>
      <PetID>0</PetID>
      <AnimalID>0</AnimalID>
      <PetName>Tashu</PetName>
    </petsTable>
    <petsTable>
      <PetID>1</PetID>
      <AnimalID>0</AnimalID>
    <PetName>Campy</PetName>
    </petsTable>
  </Animal>
  <Animal>
    <AnimalID>1</AnimalID>
    <AnimalName>Parrot</AnimalName>
    <petsTable>
      <PetID>2</PetID>
      <AnimalID>1</AnimalID>
      <PetName>Jimmy</PetName>
    </petsTable>
```

Earlier in this chapter, you looked at various desired characteristics of the object that would hold the responsibility of acting as the data carrier between various layers, processes, and machines. You saw that the DataSet object that comes bundled as a part of the Microsoft .NET Framework satisfies most of those requirements.

Frequently, you'd want to pass DataSets over Remoting or web-service boundaries. There is an important performance trick involved in sending DataSets over such boundaries. The problem is when DataSets are serialized—even if you use BinaryFormatter to serialize them, by default they will always be serialized as XML.

Note There are mainly two kinds of serialization in the .NET Framework. XML serialization and serialization done by a formatter that implements IFormatter. When I say DataSet is serialized as XML, I mean to say that even if you didn't use XML serialization (hence you used BinaryFormatter which implements IFormatter), the resultant output is still XML.

That leads to not only performance issues, but when you send "Time" data types in strongly typed DataSets (covered later) over a Remoting interface that spans two time zones, the time zone translation actually might occur twice. [2]

This is obviously a problem. Let's see this using an example. The code for this example can be found in the associated code download as Example 6.4, but I will mention the important parts of the example here.

This example will create some sample data using a method called GiveMeFakeData(), as shown in Listings 6-16 and 6-17. This DataSet has only one DataTable with two columns and a thousand rows.

Listing 6-16. GiveMeFakeData() in C#

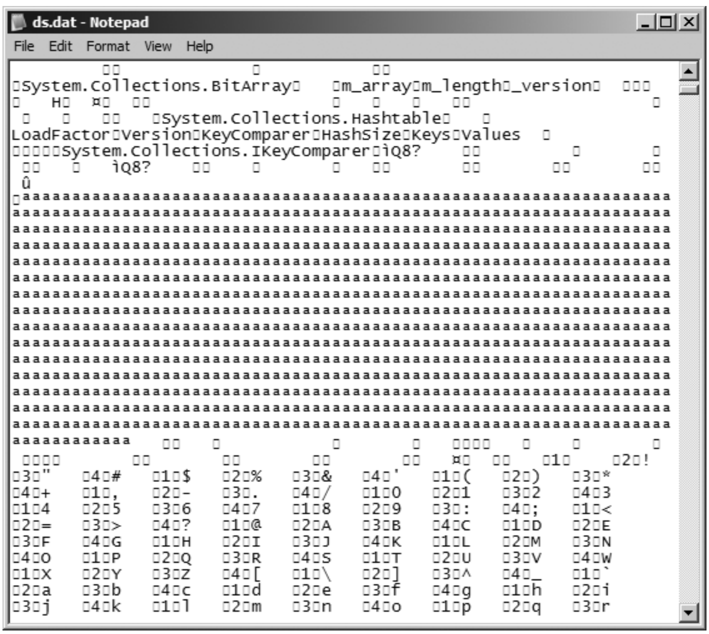
```
public static DataSet GiveMeFakeData()
{
    DataSet ds = new DataSet();
    DataTable dt = new DataTable("Animal");
    dt.Columns.Add(new DataColumn("AnimalID"));
    dt.Columns.Add(new DataColumn("AnimalType"));

    DataRow dr;
    for (int i = 0; i <= 999; i++)
    {
            dr = dt.NewRow();
            dr[0] = 1;
            dr[1] = "Rabbit";
            dt.Rows.Add(dr);
```

---

2. This discussion is bordering a discussion on Remoting. While this is not a Remoting book, you can refer to Advanced .NET Remoting, Second Edition by Ingo Rammer and Mario Szpuszta (Apress, 2005)

Figure 6-4. DataSet serialized as XML using the BinaryFormatter

And when the DataSet is serialized with SerializationFormat.Binary and the file is opened in Notepad, it looks like what is shown in Figure 6-5.



inaryFormatter

Table 6-10.Valid Data Types in an XSD Schema

Primitive XML Data Type	Description
string	Represents a character string.
Boolean	Represents a true or false value.
decimal	Represents numbers of arbitrary precision.
float	Represents a single-precision, 32-bit floating point number.
double	Represents a double-precision, 64-bit floating point number.
duration	Represents a length of time.
dateTime	Represents a specific point in time.
time	Represents a given time of day.
date	Represents a calendar date.
gYearMonth	Represents a Gregorian month and Gregorian year.
gYear	Represents a Gregorian year.
gMonthDay	Represents a Gregorian month and Gregorian day. A specific date that occurs once a year.
gDay	Represents a Gregorian day of the month.
gMonth	Represents a Gregorian month.
hexBinary	Represents hex-encoded binary data.
base64Binary	Represents Base64-encoded arbitrary binary data.
anyURI	Represents any URI (as defined by RFC 2396)ÑMay be absolute or relative.
QName	Represents a qualified name. Composed of a prefix and a local name, separated by a colon. The prefix must be a namespace thatÕs been defined by a namespace declaration.
NOTATION	Represents a set ofQNamse

To start making things a little clearer, the following is an example of an XSD file that (among other things) utilizes a couple of the primitive types described in Table 6-10. You might not understand all of the syntax at first, but the remainder will be described shortly. For now, just take a look at the attribute declarations and their associated data types (in bold):

```
<?xml version="1.0" encoding="utf-8" ?>
<xs:schema targetNamespace="http://tempuri.org/XMLSchema.xsd"
           elementFormDefault="qualified"
           xmlns="http://tempuri.org/XMLSchema.xsd"
           xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="MyElement">
    <xs:complexType>
      <xs:attribute name="MyString" type="xs:string" />
      <xs:attribute name="MyTime" type="xs:time" />
      <xs:attribute name="MyBool" type="xs:boolean" />
      <xs:attribute name="MyDecimal" type="xs:decimal" />
    </xs:complexType>
```

User-Defined Types

These are not to be confused with the User-Defined Types (UDTs) for SQL Server discussed in Chapter 5. In the context of an XML schema, a user-defined type is a restriction thatÕs placed on the content of an element or an attribute. As you saw previously, enumerations can be used to create a kind of user-defined type, as they place a restriction on the values that primitive types can use. A user-defined simple type will always restrict the contents of the element or attribute to which itÕs applied to a subset of the base type from which itÕs derived. In other words, when you create a user-defined simple type, you create a restriction on a primitive type.

The way in which individual restrictions are placed on primitive types for the purpose of defining user-defined simple types is through XSD elements called facets.

Facets

Facets are elements that are used to define a legal set of values for a simple type (which can be a user-defined simple type, or a primitive type like string or float). Constraining facets appear as child elements of a <restriction> node, which is in turn a child of a <simpleType>node. Table 6-11 is a list of the constraining facets that can be applied to a simple type (either built-in or user-defined).

Table 6-11.Constraining Facets

Constraining Facet	Description
enumeration	As you have already seen, this facet constrains the value of a simple type to a specified list of values.
fractionDigits	Specifies the maximum number of allowable digits in the fractional portion of the value.
length	Specifies the number of units of length. The units are determined by the base type of the simple type to which the facet is being applied. All values must be exactly this length.
maxExclusive	Maximum value. All values must be less than this value to qualify.
maxInclusive	Maximum value. All values must be less than or equal to this value in order to qualify.
maxLength	Maximum number of units of length. Units are determined by data type.
minExclusive	Minimum value. All values must be greater than this value to qualify.
minInclusive	Minimum value. All values must be greater than or equal to this value to qualify.
minLength	Minimum allowed length of the value. Units of length depend on the data type.
pattern	Specifies a regular expression that all values must match. A favorite!
totalDigits	Value must have a specific maximum number of total digits.
whiteSpace	Indicates whether the element should preserve, replace, or collapse whitespace.

```
            </xs:element>
          </xs:choice>
        </xs:complexType>
    </xs:element>
</xs:schema>
```

There are a couple of new things here that might immediately jump out at you. The first is that an ID is assigned to the schema. Also, you'll see that there is an `elementFormDefault` attribute with the value `qualified`. This means that, by default, all elements within the schema must be qualified with their appropriate namespace prefix. You'll also notice that a target namespace has been defined for instance documents of this XML schema. While you could get away with leaving this blank, it's a good idea to give this information in order to avoid any potential collisions when transferring `DataSet`s between domains, machines, or platforms.

Looking deeper into the schema, you'll notice that you can specify an unlimited number of `<Books>` elements, which then contain a sequence of elements called `<Title>` and `<Publisher>`, in that order. You could create a `Books.xml` document for our sample application to read that looks like this:

```
<BookDataSet xmlns="urn:apress-proadonet-chapter6-BookDataSet.xsd">
  <Books>
    <Title>Pro ADO.NET</Title>
    <Publisher>Apress Ltd</Publisher>
  </Books>
  <Books>
    <Title>Professional .NET Framework</Title>
    <Publisher>Apress Ltd</Publisher>
  </Books>
</BookDataSet>
```

Listings 6-20 and 6-21 demonstrate a part of a small console application that creates an ordinary `DataSet`, reads the XSD file (`BookDataSet.xsd`), loads the `Books.xml` file using the `ReadXml` method, and then prints the information in the XML file to the console, using the relational paradigm of tables, columns, and rows. This can be found in the associated code download as Example 6.5. The `BookDataSet.xsd` and `Books.xml` files are both in the `bin` directory beneath the project.

Listing 6-20. Loading DataSet Data and Schema from XML and XSD in C#

```
DataSet bookDataSet = new DataSet();
BookDataSet.ReadXmlSchema("BookDataSet.xsd");
BookDataSet.ReadXml("Books.xml");

Console.WriteLine("Recent Books:");
Console.WriteLine("--------------");

foreach (DataRow xRow in BookDataSet.Tables["Books"].Rows)
{
    Console.WriteLine("{0} by {1}", xRow["Title"], xRow["Publisher"]);
}
```

```
<xs:unique name="KeyTitle">
  <xs:selector xpath=".//Books" />
  <xsd:field xpath="Title" />
</xs:unique>
```

Foreign Keys (<keyref>) and Relationships

<keyref> elements within an XML schema provide a facility for declaring links within the document. The functionality they establish is similar in nature to that of foreign-key relationships in relational databases like SQL Server. If a DataSet encounters a <keyref> element when loading a schema, the DataSet will create an appropriate foreign-key constraint. It will also create a parent-child relationship (discussed shortly).

This example creates a new table in our DataSet called BookReviews. Then, the <keyref> element is used to create a foreign-key relationship between the BookReviews table and the Books table. To make things easier (and more realistic), a BookID column is also added. This code can be found in Example 6.6 of the associated code download.

Here's the new BookDataSet.xsd file for Example 6.6 (you'll have to find the BookDataSet.xsd on the file system as it is not added as a part of the project in Example 6.6):

```
<?xml version="1.0" encoding="utf-8" ?>
<xs:schema id="BookDataSet"
           targetNamespace="urn:apress-proadonet-chapter6-BookDataSet.xsd"
           elementFormDefault="qualified"
           xmlns="urn:apress-proadonet-chapter6-BookDataSet.xsd"
           xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:msdata="urn:schemas-microsoft-com:xml-msdata">

  <xs:element name="BookDataSet" msdata:IsDataSet="true">
    <xs:complexType>
      <xs:choice maxOccurs="unbounded">
        <xs:element name="Books">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="BookID" type="xs:integer" minOccurs="1" />
              <xs:element name="Title" type="xs:string" minOccurs="1" />
              <xs:element name="Publisher" type="xs:string" minOccurs="1" />
            </xs:sequence>
          </xs:complexType>
        </xs:element>
..... continued below
```

To this point, everything should look pretty familiar. You'll notice that the BookID column has been added. BookID is an integer column that must appear any time a <Books> element appears (minOccurs="1"). In other words, in the DataSet, this would be not nullable. Next, a new definition for a table is created. Remembering that a table in XSD is nothing more than an element that contains more elements, it's pretty easy to do:

```
.....  continued from above
```

```
            <xs:complexType>
              <xs:sequence>
                <xs:element name="BookID" type="xs:integer" minOccurs="1" />
                <xs:element name="Rating" type="xs:integer" minOccurs="1" />
                <xs:element name="Review" type="xs:string" minOccurs="0" />
              </xs:sequence>
            </xs:complexType>
          </xs:element>
        </xs:choice>
      </xs:complexType>
```

After that, some more familiar-looking code. You went over the <key>element earlier: the xpath attribute of the <selector> element indicates the Bookstable, while the xpath attribute for the <field> indicates the BookIDcolumn:

```
      <xs:key name="KeyTitle">
        <xs:selector xpath=".//Books" />
        <xs:field xpath="BookID" />
      </xs:key>
```

Finally, a <keyref> element is used to indicate that youÕre creating a foreign key originating from the BookIDcolumn of the BookReviewstable, referring to the <key>element named KeyTitle :

```
      <xs:keyref name="KeyTitleRef" refer="KeyTitle">
        <xs:selector xpath=".//BookReviews" />
        <xs:field xpath="BookID" />
      </xs:keyref>
    </xs:element>
</xs:schema>
```

With the XSD file in place, you can next write the code to read the XSD and the XML into a DataSet. This code can be found in Listings 6-22 and 6-23.

Listing 6-22. Loading Schema and Data for a Relational DataSet in C#

```
DataSet bookDataSet = new DataSet();

BookDataSet.ReadXmlSchema("BookDataSet.xsd");
BookDataSet.ReadXml("Books.xml");

Console.WriteLine("Relations Created:");

foreach (DataRelation xRelation in BookDataSet.Relations)
{
    Console.WriteLine(xRelation.RelationName);
}

Console.WriteLine("Apress Books");
```

Listing 6-23. Loading Schema and Data for a Relational DataSet in Visual Basic .NET

```
Dim BookDataSet As New DataSet()

BookDataSet.ReadXmlSchema("BookDataSet.xsd")
BookDataSet.ReadXml("Books.xml")

Console.WriteLine("Relations Created:")
Dim xRelation As DataRelation
For Each xRelation In BookDataSet.Relations
   Console.WriteLine(xRelation.RelationName)
Next

Console.WriteLine("Apress Books")
Console.WriteLine("----------")
Console.WriteLine()
```

Programmers like proof. To prove that a relationship in the DataSet has been created with exactly the same name as the <keyref> element in the XSD file, you can iterate through the Relations collection, printing out the name of the relationship. YouÕll see that there is indeed a relationship called KeyTitleRef in the DataSet immediately after the schema is loaded:

C#

```
foreach (DataRow xRow in BookDataSet.Tables["Books"].Rows)
{
    Console.WriteLine(xRow["Title"]);
......
}
```

VB.NET

```
For Each xRow as DataRow In BookDataSet.Tables("Books").Rows
    Console.WriteLine(xRow("Title"))
......
Next
```

So far, this is pretty straightforward: just iterate through each of the rows in the DataSet, and print out the value in the Title column for each of the rows:

C#

```
foreach (DataRow xRow in BookDataSet.Tables["Books"].Rows)
{
    Console.WriteLine(xRow["Title"]);
    // Obtain child rows using the KeyTitleRef relation
    foreach (DataRow zRow in xRow.GetChildRows("KeyTitleRef"))
    {
        Console.WriteLine("  {0}", zRow["Rating"]);
    }
```

In short, strongly typed DataSets let you turn the following lines of code:

C#

```
myRow = myDataSet.Tables["Customers"].Rows[0]
Console.WriteLine(myRow["FirstName"])
```

VB.NET

```
myRow = myDataSet.Tables("Customers").Rows(0)
Console.WriteLine(myRow("FirstName"))
```

into this short, easy-to-read line of code:

C#

```
Console.WriteLine(CustomerDataSet.Customers[0].FirstName) ;
```

VB.NET

```
Console.WriteLine(CustomerDataSet.Customers(0).FirstName)
```

This change is made possible by the strong support in the .NET Framework for inheritance and clear, logical object hierarchies. The essential concept behind a strongly typed DataSet is that a new class is created that derives from the DataSet class. This new class implements properties and methods that are strongly typed, based on the XSD schema that was used in order to generate it.

When this derived class is instantiated, it will provide logically named properties of integer type (if your schema called for integer columns), and further logically named properties that appear as arrays of strongly typed rows. So, rather than dealing with DataRows and DataTables, you end up dealing with task-specialized concepts such as a CustomersRow or a CustomersTable, which may contain columns called FirstName or LastName, rather than forcing you to employ difficult-to-use array indices/indexer accessors.

In .NET, there are two (equally valid) approaches to creating strongly typed DataSets. You can create them visually using a designer through the tools provided within Visual Studio .NET, or you can use command-line utilities and compiler arguments. Both approaches will be covered here, and you can pick the one that best fits your skills, style, and preferences.

Building Typed DataSets in Visual Studio .NET

Visual Studio .NET provides an incredibly easy, point-and-click approach to creating strongly typed DataSets. ItÕs such a natural extension of Visual Studio .NET that if youÕre using it to develop your application, thereÕs no sensible reason not to use it. To start out, letÕs take the concept from the Òbooks and book reviewsÓ example and develop a strongly typed DataSet for it in Visual Studio .NET:

1. As usual, start out by creating a new console application, this time named Example 6.7.
2. Next, right-click on the project name, select Add → Add New Item, and then select the DataSet icon and type in BookDataSet.xsd as the filename.
3. From the Toolbox, drag two DataTables into open areas of the design surface. From there, you can use the visual interface to add columns to each DataTable you dropped

VB.NET

```
Console.WriteLine(book.Title)

For Each review As BookDataSet.BookReviewsRow In book.GetBookReviewsRows()
    Console.WriteLine("  {0}", review.Rating)
Next
```

The code generator for the strongly typed DataSet will actually generate properly typed properties for each of the columns in a table. Here you can see that the example is printing out the value of Book.Title , rather than the following code or something similarly complex:

C#

```
Tables["Books"].Rows[0]["Title"] ;
```

VB.NET

```
Tables("Books").Rows(0)("Title")
```

Also note that instead of invoking the GetChildRows method, you are actually calling GetBookReviewsRows(), which the strongly typed DataSet implemented for you. This method returns an array of BookReviewsRow objects.

After you've placed all the code into your solution (or you've loaded the Example 6.7 project from the code download), rebuild it and then make sure that Show All Files is selected from the Project menu. You should see that the BookDataSet.xsd file has some child items. Expand them, and you'll see that a BookDataSet.Designer.cs/ BookDataSet.Designer.vb file has been generated for you. This is the actual class definition that the example uses as the strongly typed DataSet.

If you open the BookDataSet.xsd file in Notepad, you would immediately notice that the Books table is represented by the following element:

```
<xs:element name="Books">
 <xs:complexType>
   <xs:sequence>
     <xs:element name="BookID" type="xs:integer" minOccurs="1" />
     <xs:element name="Title" type="xs:string" minOccurs="1" />
     <xs:element name="Publisher" type="xs:string" minOccurs="1" />
   </xs:sequence>
 </xs:complexType>
</xs:element>
```

Also, you would note that the BookReviews table is defined by the following element:

```
<xs:element name="BookReviews">
 <xs:complexType>
   <xs:sequence>
     <xs:element name="BookID" type="xs:integer" minOccurs="1" />
     <xs:element name="Rating" type="xs:integer" minOccurs="1" />
     <xs:element name="Review" type="xs:string" minOccurs="0" />
   </xs:sequence>
 </xs:complexType>
```

Like so many properties, this is just a wrapper around a private member variable. The key thing to note here is the data type of the property: itÕs actually a nested class called BooksDataTable, which is another dynamically generated class deriving from the DataTable class.

Now letÕs take a look at another piece of ÒmagicÓ that the strongly typed DataSet is performing on your behalf: the invocation of GetChildRows() through the GetBookReviewsRows() method (a member of the BookDataSet.BooksRow class):

C#

```
public BookReviewsRow[] GetBookReviewsRows() {
    return
((BookReviewsRow[])(this.GetChildRows(this.Table.ChildRelations["KeyTitleRef"])));
}
```

VB.NET

```
    Public Function GetBookReviewsRows() As BookReviewsRow()
        Return CType(Me.GetChildRows( _
                    Me.Table.ChildRelations("KeyBookIDRef")), BookReviewsRow())
    End Function
```

This function locates a DataRelation object by pulling the KeyBookIDRef item out of the ChildRelations collection. It then pulls the child rows by passing the relation object to the GetChildRows() function, changing the type of the resulting array of DataRow objects to an array of BookReviewsRow objects. As with all of the items in the strongly typed DataSet, this function is visible through intellisense in Visual Studio .NET, dramatically reducing your chances of mistyping the name of the child rowÕs relation name.

One more useful feature of the typed DataSet is its strongly typed properties. LetÕs take a look at the Title property of the BooksRow class:

C#

```
public string Title {
        get {
            return ((string)(this[this.tableBooks.TitleColumn]));
        }
        set {
            this[this.tableBooks.TitleColumn] = value;
        }
}
```

VB.NET

```
    Public Property Title As String
        Get
            Try
                Return CType(Me(Me.tableBooks.TitleColumn), String)
            Catch e As InvalidCastException
                Throw New StrongTypingException( _
                            "Cannot get value because it is DBNull.", e)
```

```
    End Get
    Set
      Me(Me.tableBooks.TitleColumn) = value
    End Set
  End Property
```

This property is made possible by an override for the indexer (the `Item` property) in the `DataRow` class that `BooksRow` inherits from. It allows an actual column object to be supplied (specifically, a `DataColumn` object, or, as in this case, an object that inherits from `DataColumn`), rather than an ordinal or a string as a field identifier, on which to set and get values.

Building Typed DataSets Manually

Now that you have looked at how to make strongly typed `DataSet`s through Visual Studio .NET, which automatically makes them available to the rest of the project, let's look at how to do it the "hard" way. Of course, that's a misnomer, because there's really nothing hard about using command-line tools to create your own strongly typed `DataSet`s.

You shouldn't confuse "manually" with the process of writing all of the code for the `DataSet` class yourself. Even without Visual Studio .NET, there's still a tool called `XSD.exe` that automates generation of the class file. This approach is useful for batch code generation, typical in enterprise deployment or automated build environments. For the "manual" example, you will build the same example you have already built under Visual Studio .NET, though all of the compilation, code editing, and `DataSet` generation will be done from the command line and Notepad (or any other text editor).

The first thing that needs to be done is to create a new directory called Example 6.8. Into this, copy the `BookDataSet.xsd` file that you built containing the definitions for the `Books` table, the `BookReviews` table, the keys, and the `keyref` parent-child relationship.

Then, write a batch file to build a strongly typed `DataSet`. It's called `BuildDS.cmd`; the contents of this batch file are as shown:

C#

```
xsd /d /l:CS BookDataSet.xsd
```

VB.NET

```
xsd /d /l:VB BookDataSet.xsd
```

The batch file invokes a utility called `XSD.exe`, which is used to generate schema or class files from a given source. For this statement to work, you have to either run it from where `xsd` is in the same directory or where it is within the defined path. It can take a schema and either convert it to classes or create a typed `DataSet` from it (the `/d` argument). Also, it can create an XSD file from an XDR file to upgrade the schema definition. The `/l` version can specify the language to generate the strongly typed `DataSet` in.

Running the `BuildDS.bat` file creates a version of the strongly typed `DataSet` exactly the same as the one created through Visual Studio .NET. The same output for the Visual Basic version is as shown in Figure 6-7.

This is accomplished by supplying attributes from two XML namespaces provided by Microsoft: the first is the codegen namespace (defined by xmlns:codegen="urn:schemas-microsoft-com:xml-msprop") and the second is the msdata namespace (defined by xmlns:msdata="urn:schemas-microsoft-com:xml-msdata").

codegen

The codegen namespace contains a set of attributes that directly affect the code generation of a DataSet. You can apply the codegen attributes to various elements of an XSD file, providing fine-grained instructions to either XSD.exe or the VS.NET compiler on exactly how to generate the new DataSet. There will be a sample of a Visual Studio .NET-annotated DataSet using these new attributes at the end of this section.

All of this functionality can be controlled by programmatically modifying the properties of the DataSet later on. However, if the functionality and control is built into the schema and the typed DataSet in human-readable form, there can be no mistake about how the class creator intended it to function.

typedName

The typedName attribute indicates the name of an object, as it will appear in the new DataSet. This attribute can be applied to DataTables, DataRows, properties, and DataSet events.

typedPlural

The typedPlural attribute will indicate the name of the object when a plurality of the object is needed, as in the DataRowCollection or the DataTableCollection object.

typedParent

The typedParent attribute indicates the name of the object when it is referred to in a parent relationship. Typed DataSets automatically generate accessor functions for retrieving parents and children. For example, in the previous example, the GetBookReviewsRows() function was a child accessor.

typedChildren

The typedChildren attribute indicates the name of the object when it is referred to in a child relationship. As stated previously, typed DataSets generate both parent and child accessors, usually with confusing or unwieldy names. Providing the typedChildren and typedParent attributes generally makes for a much easier user experience in your typed DataSet.

nullValue

The nullValue attribute is an incredibly useful one. It allows you to define what action will be taken in the DataSet when a DBNull value is encountered. The following is a list of the valid values for the nullValue attribute:

- "replacement" : Rather than indicating a behavior, you can simply indicate what value your DataSet will store instead of DBNull.
- _throw: Throws an exception any time a DBNull is encountered on the related element

```
<xs:annotation>
  <xs:appinfo>
    <msdata:Relationship name="KeyBookIDRef"
                         msdata:parent="Books"
                         msdata:child="BookReviews"
                         msdata:parentkey="BookID"
                         msdata:childkey="BookID" />
  </xs:appinfo>
</xs:annotation>
```

Annotated Typed DataSet Example

Now that you've seen how to annotate a DataSet in order to control how its code is generated and to gain finer control over its behavior, constraints, and rules, let's take a look at one in action. For this example, we're going to create a Windows Forms application called Example 6.9. Add a new item to the project, choose DataSet as the type, and name it BookDataSet.xsd. Paste the following annotated XSD into your new DataSet class XSD file:

```
<?xml version="1.0" encoding="utf-8" ?>
<xs:schema id="BookDataSet"
           targetNamespace="urn:apress-proadonet-chapter6-BookDataSet.xsd"
           elementFormDefault="qualified"
           xmlns="urn:apress-proadonet-chapter6-BookDataSet.xsd"
           xmlns:mstns="urn:apress-proadonet-chapter6-BookDataSet.xsd"
           xmlns:xs="http://www.w3.org/2001/XMLSchema"
```

The next part is new. Visual Studio .NET puts the msdata namespace declaration into your DataSet schemas for you, but you have to enter the following one manually in order to gain access to the codegen namespace prefix:

```
           xmlns:codegen="urn:schemas-microsoft-com:xml-msprop"
           xmlns:msdata="urn:schemas-microsoft-com:xml-msdata">


  <xs:element name="BookDataSet" msdata:IsDataSet="true">
    <xs:complexType>
      <xs:choice maxOccurs="unbounded">
```

Here's a look at the first use of the codegen namespace. The following code is indicating that the typedName of the Books element (which matches <Books> tags in the XML instance document) is going to be called Book, and the typedPlural will be called Books:

```
        <xs:element name="Books" codegen:typedName="Book"
                                 codegen:typedPlural="Books">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="BookID" type="xs:integer" minOccurs="1" />
              <xs:element name="Title" type="xs:string" minOccurs="1" />
              <xs:element name="Publisher" type="xs:string" minOccurs="1" />
            </xs:sequence>
          </xs:complexType>
```

C#

```
foreach (BookDataSet.Book book in books.Books)
{
    foreach (BookDataSet.BookReview review in book.Reviews())
    {
        sum += Convert.ToInt32(review.Rating);
    }
}
MessageBox.Show(this, "Score Total: " + sum.ToString());
```

VB.NET

```
For Each Book as BookDataSet.Book In Books.Books
    For Each review as BookDataSet.BookReview In Book.Reviews()
        sum += CInt(review.Rating)
    Next
Next
MessageBox.Show(Me, "Score Total: " + sum.ToString())
```

Figure 6-8 shows just how much nicer everything looks when things have been annotated and given human- and programmer-readable names.

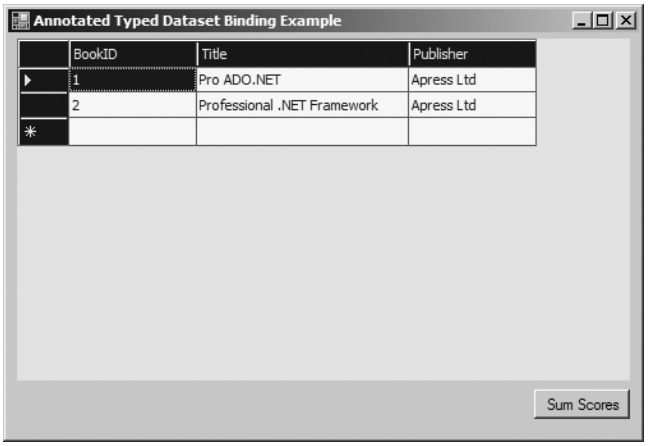


Figure 6-8. Annotated typed DataSet binding example

Summary

This chapter introduced you to the centerpiece of ADO.NET's disconnected architecture—the DataSet.

In this chapter, you then looked at various constituents of the DataSet object. You looked at DataTables, DataRows, DataColumns, DataRelations, etc. You saw many practical examples elucidating the usage of all of these objects. It was also noted that a lot of concepts mentioned

CHAPTER 7

Fetching Data: The DataAdapter

So far you've been introduced to the various parts that make up ADO.NET. You read about how ADO.NET splits up data access between connected and disconnected portions and why it chooses to do so. In Chapters 4 and 5 you read about the connected part of ADO.NET. You read about how a connection is established, and how you can execute commands and fetch results out of the database in a connected fashion using a data reader.

Chapter 6 introduced you to the various objects, such as `DataSet`, `DataTable`, `DataRow`, `DataRelation`, and so on that comprise the purely disconnected data access part within ADO.NET. Various examples were given how `DataSet`s can be used to hold a logical representation of your data model in both a strongly typed and non-strongly typed fashion.

Just as a data reader allows you to fetch data from a given data source, there exists an object in ADO.NET that acts as a bridge between the purely disconnected world of ADO.NET comprising the `DataSet`, `DataTable`, etc. and the connected world of the `DBCommand` objects that wrap various SQL statements: this object is the `DataAdapter`.

A `DataAdapter` not only lets you fill a `DataSet` or `DataTable` from the given data source, it also provides you with a convenient mechanism to persist, or save the changes back, into the database. This is usually achieved by specifying various command objects as properties on the `DataAdapter` object.

But before you look at updating the data source using a data adapter, first let's discuss fetching data using a data adapter.

What Is a DataAdapter?

As the name implies, a `DataAdapter` is the bridge between the connected and disconnected parts of ADO.NET. A `DataAdapter` is the object that lets you use connected objects, such as `DbCommand`, to fill and work with disconnected objects, such as the `DataSet`. As you have already seen in Chapters 4 and 5, in order to interrogate or interact with your data source, you need an object derived from `DbConnection`. Typically, this would be an instance of a `SqlConnection`, `OracleConnection`, or something similar. On an open and available object that is derived from `DbConnection`, you can send commands using an object that is derived from `DbCommand`. A `DbCommand` has the option of being able to retrieve the result set as a `DbDataReader` (such as `SqlDataReader` or `OracleDataReader`), or it may return a scalar value instead. Such commands may even be used to perform Data Manipulation Language (DML) on the database.

However, one of the things pointed out in Chapter 4 was the immense benefit of connection pooling in a multiuser, highly concurrent application environment. Since a data reader operates in a firehose read-only/forward-only manner, it ends up holding a connection open for its operation lifetime. Holding a connection open for an inordinately long time generally has a negative impact on connection pooling.

In Chapter 6, you read about various objects that form the disconnected architecture of ADO.NET. Objects such as DataSets, DataTables, DataRows, etc. help you hold a subset of the data in a relational form, similar to what you would find in the database. However, these objects, particularly the DataSet and DataTable, need to be filled using some mechanism.

ADO.NET employs the DbCommand object to abstract and hold the queries you wish to execute on the data source. The DbCommand object contains the information about what DbConnection it will employ in order to execute itself. The DataAdapter object is responsible for employing a DbCommand and acting as a gatekeeper/bridge between the connected world of DbCommand and DbConnection, and the disconnected world of DataTable and DataSet.

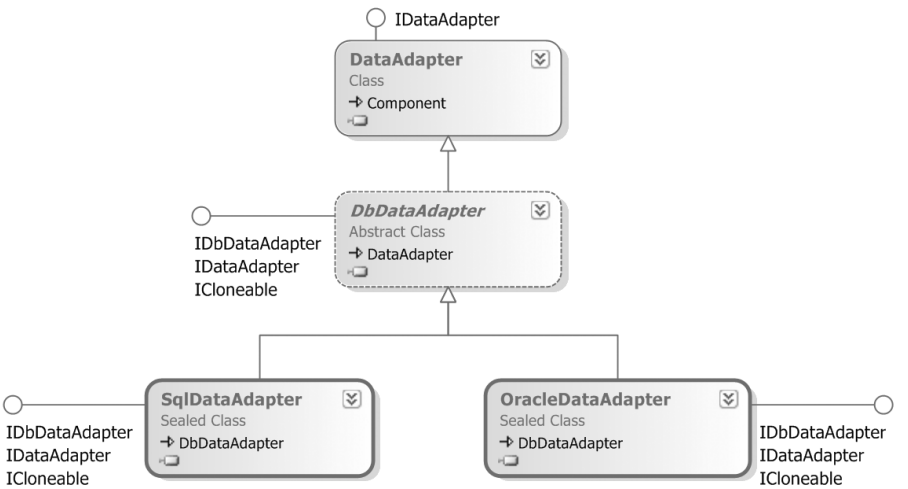
Structure of a DataAdapter

In Chapter 2 (specifically, Figure 2-7), you read about data adapters implementing the IDataAdapter and IDbDataAdapter interfaces without going into further detail. In fact, it's not the case that the various data adapters simply contain implementations of these interfaces' methods. Though that's effectively the outcome, there are a couple of stages in between.

The .NET Framework class hierarchy contains an abstract or MustInherit class that provider-specific data adapters inherit from called System.Data.Common.DbDataAdapter. It's the DbDataAdapter class that implements the IDbDataAdapter interface. The DbDataAdapter class provides the basic facilities for constructing a disconnected data-access mechanism to fill a DataSet or DataTable object and to update a data source.

The DbDataAdapter class in turn inherits from yet another class called the System.Data.Common.DataAdapter class, which in turn implements the IDataAdapter interface.

Figure 7-1 demonstrates the structure of two commonly used data adapters: the SqlDataAdapter and OracleDataAdapter in ADO.NET.



ataAdapter in ADO.NET

3. Once the Data Sources window is enabled, click the Add New Data Source icon on the top (or click the hyperlink in the window if available), and point the new database connection to the Test database on your local SQL Server 2005 instance. If you wish to use a SQL Server instance on the network, or an alternate data source such as Oracle, you need to use a different connection string and possibly different objects, but the overall pattern remains the same. Choosing the Test database on your local SQL Server 2005 instance is shown in Figures 7-4 and 7-5.

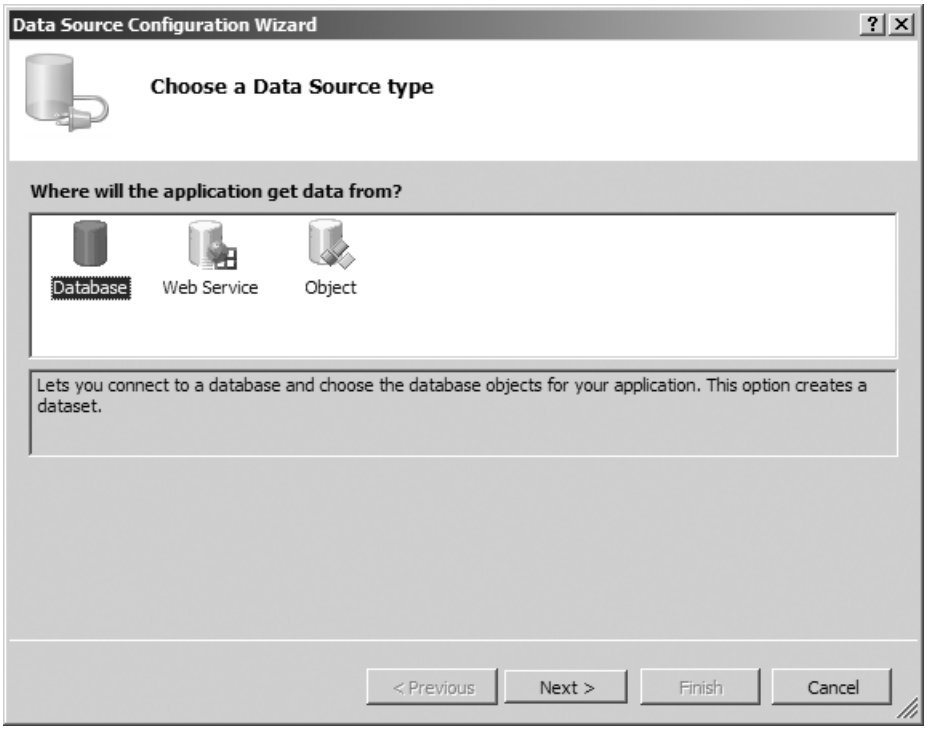


Figure 7-4.Choosing the appropriate data source (in our case ÒDatabaseÓ)

4. Now, go to the code for the form. To do this, right-click on the form and choose View Code. YouÕll see the partial class implementation for Form1.cs (or .vb). Over here, add using / Imports statements for System.Data and System.Data.SqlClient. Also, add a private variable to the class called userTable of type DataTable and instantiate it in the constructor, as shown here:

C#

```
partial class Form1 : Form
{
    private DataTable userTable;

    public Form1()
    {
        InitializeComponent();
        userTable = new DataTable();
    }
...
}
```

VB.NET

```
Public Class Form1
    Private userData As New DataTable
...
End Class
```

5. Now, back to the formÕs designer: double-click Fill Data (buttonFillData) to create an event handler for its click event. In this event handler, write code to populate the DataTable. This is shown in Listings 7-1 and 7-2.

Listing 7-1. Filling userTable Programmatically in C#

```
private void buttonFillData_Click(object sender, EventArgs e)
{
    // Never hard code connection strings.
    // Usually you would get this from a config file
    string connectionString =
    "Data Source=(local);Initial Catalog=Test;Integrated Security=SSPI;" ;

    using (SqlConnection testConnection = new SqlConnection(connectionString))
    {
        SqlCommand testCommand = testConnection.CreateCommand();
        testCommand.CommandText = "Select FirstName, LastName from userTable";
        SqlDataAdapter dataAdapter = new SqlDataAdapter(testCommand);
        dataAdapter.Fill(userTable);
    } // testConnection.Dispose called automatically.
```

Figure 7-12.The final running application

8. As an exercise, change theuserTable DataTable to a DataSet, and make suitable changes to the DataBind code. YouÕll note that theDataAdapter is able to Fill both a DataTable and a DataSet.

Unlike Exercise 7.1, this exercise involved writing some code. LetÕs dissect the code and try to understand what exactly was done.

The first part was obviously to set up the connection. This was done using the following code:

C#

```
using (SqlConnection testConnection = new SqlConnection(connectionString))
{
...
}
```

VB.NET

```
Using testConnection As SqlConnection = New SqlConnection(connectionString)
...
End Using
```

The next part was creating a SqlCommand for the underlying Microsoft SQL Server 2005 database (this has been extensively covered in Chapter 5):

C#

```
SqlCommand testCommand = testConnection.CreateCommand();
testCommand.CommandText = "Select FirstName, LastName from userTable";
```

VB.NET

```
Dim testCommand As SqlCommand = testConnection.CreateCommand()
testCommand.CommandText = "Select FirstName, LastName from UserTable"
```

The next step was to create a data adapter. This was done using the following statement:

C#

```
SqlDataAdapter dataAdapter = new SqlDataAdapter(testCommand);
```

VB.NET

```
Dim dataAdapter As New SqlDataAdapter(testCommand)
```

The SqlDataAdapter constructor supports four overloads. The overload used here (SqlDataAdapte(testCommand)) lets you specify a command with a connection setup in the constructor itself. The other three overloads available are as follows:

- SqlDataAdapter() : This sets up a bare instance of a SqlDataAdapter. The command with a valid connection now must be specified in the SelectCommand property. In Chapter 9, where updating data is discussed, you'll come across three more properties of a SqlDataAdapter object: the InsertCommand, DeleteCommand, and UpdateCommand properties, which also use SqlCommand type variables used for updating the data back to the data source. They cannot be specified via the constructor, but they can be specified as properties on the SqlDataAdapter object, once it is created.
- SqlDataAdapter(string commandText, SqlConnection connection) : This overload lets you specify the command being used to fill the DataSet/DataTable as a string and the connection it will be filled from.
- SqlDataAdapter(string commandText, string connectionString) : This overload lets you specify both the command and the relevant connection as strings; thus, you can specify all the necessary information in one line of code without instantiating any extra objects yourself.

And finally, with the data adapter ready, all the application had to do was fill the DataTable using the Fill method. This was done using the following code statement:

C#

```
dataAdapter.Fill(userTable);
```

VB.NET

```
dataAdapter.Fill(userTable)
```

Do note, however, that the data adapter's Fill method can be used to fill either a DataSet or a DataTable. So if, in a particular instance, your needs are met by a single DataTable, you should use that instead of a DataSet with one table in it because the single DataTable has a smaller memory footprint.

Another point to notice about this code is that the underlying SqlConnection was never explicitly opened. This is quite interesting because the SqlDataAdapter doesn't need an already open connection. As a matter of fact, as a part of the Fill operation, SqlDataAdapter would

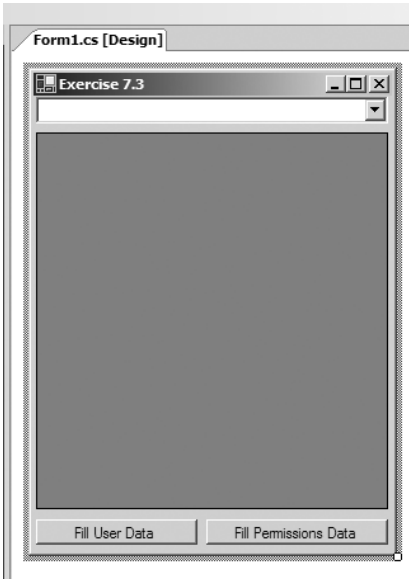


Figure 7-13.DataSet UI fill example in Design mode

The intention here is to be able to provide the user with a combo box full of various DataTables available in the DataSet. The DataGridView datagridView control shows the currently selected DataTable, and the two buttons allow you to fill in the DataSet one by one. LetÕs skip a Data Bind button here because the combo box will serve that purpose.

3. Add a private variable of DataSet type. Call it myData

C#

```
partial class Form1 : Form
{
    private DataSet myData;

    public Form1()
    {
        InitializeComponent();
        myData = new DataSet();
    }
...
}
```

VB.NET

```
Public Class Form1
      Private myData As New DataSet
...
```

This code loads the UserTable into the DataSet, and also calls a method named UpdateComboBox, which is responsible for keeping the DataSet and comboTables in sync. The code for UpdateComboBox is shown in Listings 7-9 and 7-10.

Listing 7-9. Code to Keep the Combo Box and DataSet in Sync in C#

```
private void UpdateComboBox()
{
    comboTables.Items.Clear();
    foreach (DataTable tbl in myData.Tables)
    {
        comboTables.Items.Add(tbl.TableName);
    }
}
```

Listing 7-10. Code to Keep the Combo Box and DataSet in Sync in Visual Basic .NET

```
Private Sub UpdateComboBox()
    comboTables.Items.Clear()
    For Each tbl As DataTable In myData.Tables
        comboTables.Items.Add(tbl.TableName)
    Next
End Sub
```

5. In the event handler for buttonPermData's click event, add code exactly the same as that for buttonUserData's click event, but change the CommandText to

   Select PermissionType from PermissionsTable

   and the dataAdapter.Fill statement to

   C#
   ```
   dataAdapter.Fill(myData, "PermissionsTable");
   ```

   VB.NET
   ```
   dataAdapter.Fill(myData, "PermissionsTable")
   ```

6. You're almost done. Now you need to hook up the combo box's SelectedIndexChanged event to toggle datagridView's current binding to a particular DataTable. This is shown in Listings 7-11 and 7-12.

Listing 7-11. Binding the DataGridView with the Right Table in C#

```
private void comboTables_SelectedIndexChanged(object sender, EventArgs e)
{
    datagridView.DataSource = myData.Tables[comboTables.SelectedIndex];
}
```

Listing 7-12. Binding the DataGridView with the Right Table in Visual Basic .NET

```
Private Sub comboTables_SelectedIndexChanged( _
   ByVal sender As System.Object, ByVal e As System.EventArgs)  _
   Handles comboTables.SelectedIndexChanged
    datagridView.DataSource = myData.Tables(comboTables.SelectedIndex)
End Sub
```

7. Compile and run the application. Click the buttonUserData button. YouÕll see the combo box populated with one DataTable: the UserTable.

8. Click the buttonPermData button. YouÕll see that the PermissionsTable is also loaded into the DataSet. The final running application can be seen in Figure 7-14.

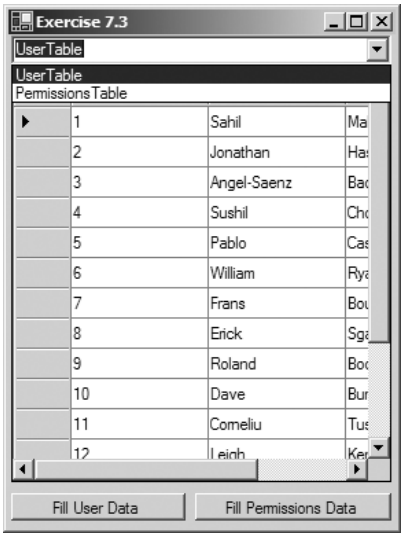


Figure 7-14.DataSet UI fill example with tables filled

One interesting thing to note in Exercise 7.3 is that if you click either of the two buttons repeatedly, the DataAdapter simply adds the rows over and over again. This is probably not what you would want in repeated ÒrefreshesÓ of theDataSet.

To understand how you can prevent this from happening, you need to examine what causes this behavior in the first place. When the DataAdapter looks to fill an already full DataSet, it has no way of identifying which rows being filled are already present in the DataSet. ThereÕs no primary key to look up and verify with. The DataSet is missing the schema information.

Aha!! Thus to solve this issue, all you need to do is specify the schema, right?

Yes, thatÕs all you need to do, but it gets easier than this; even though you could do it the hard way, you donÕt quite have to specify the full schema yourself. You can query the data source and fill the schema instead.

Querying Database Schema

In Exercise 7.3, you looked at an example that demonstrated filling a DataSet repeatedly from

epeated ÒrefreshesÓ of the

Figure 7-17.DataSet schema and data example without the schema loaded

8. Finally, run the application, click Fill Data twice (to load incorrect data), and now try and add a schema to the DataSet by clicking Fill Schema. You should see an exception as shown in Figure 7-18.

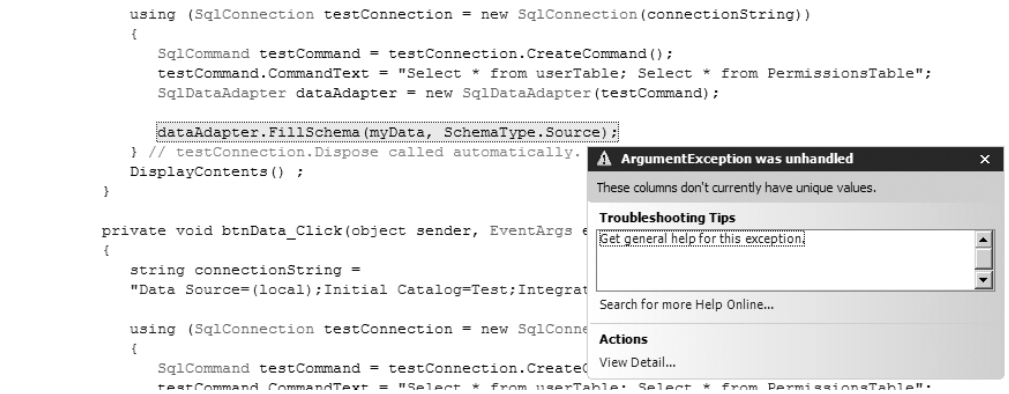


Figure 7-18.The ÒCannot load schema with incorrect dataÓ exception

```
        usersMapping.ColumnMappings.Add(lNameColumnMap)
        usersMapping.ColumnMappings.Add(cityColumnMap)
        usersMapping.ColumnMappings.Add(stateColumnMap)
```

The DataTableMapping object has a constructor that takes two strings. The first specifies the name of the source table and is case sensitive. This name must correspond to the table name used during the filling or updating process accomplished by the DataAdapter object. If you don't specify a source table name, you must use the default name assigned by the DataAdapter object: Table.

The second parameter is the DataTable object's name in the DataSet. The DataTableMapping object exposes the ColumnMappings collection property that must contain every column you want to map from the database to the DataSet.

Finally, we add the DataTableMapping object to the TableMappings property of the data adapter. Now when we fill the DataSet, the data adapter will find the DataTableMapping in its TableMappings collection and use it, as shown in Listings 7-26 and 7-27.

Listing 7-26. Using C#: The DoDataMappings Method, Adding the Table Mapping to the Data Adapter

```
        // Activate the mapping mechanism
        dataAdapter.TableMappings.Add(usersMapping);
    }
    catch (Exception ex)
    {
        // An error occurred. Show the error message
        Console.WriteLine(ex.Message);
    }
}
```

Listing 7-27. Using Visual Basic .NET: The DoDataMappings Method, Adding the Table Mapping to the Data Adapter

```
        ' Activate the mapping mechanism
        dataAdapter.TableMappings.Add(usersMapping)
    Catch ex As Exception
        ' An error occurred. Show the error message
        Console.WriteLine(ex.Message)
    End Try
End Sub
```

This has taken quite a bit more code than just using the SQL AS keyword, but this approach works much better with disconnected data since this is an ADO.NET solution, not a SQL Server or an Oracle database solution. The good news is that there is a shorter way to use the data mapping objects—you just looked at the longer version to get a better idea of what is happening. The bad news is that it's still not quite as short as using the AS keyword. Let's look at a shortened version of the DoDataMappings method in Listings 7-28 and 7-29.

```
        ' Activate the mapping mechanism
        dataAdapter.TableMappings.Add(usersTableMapping)
    Catch ex As Exception

        ' An error occurred. Show the error message
        Console.WriteLine(ex.Message)
    End Try
End Sub
```

This time you simply create an array of `DataColumnMapping` objects instead of declaring them all separately. You then use another `DataTableMapping` constructor that accepts an array of `DataColumnMapping` objects in the constructor, rather than adding each `DataColumnMapping` separately.

There are no substantial differences between using one method over the other. The first case requires more code, but itÕs slightly more readable than the second one; however, with a few clear comments, the shorter version is perfectly understandable.

MissingMappingAction and MissingSchemaAction

You just saw that when the data adapter fills a `DataSet`, it checks to see what table mappings have been specified. If none have been specified, then it uses the original names. However, this is only the default. You can also choose how you want the data adapter to react if it meets columns that you have no specified mappings for. The `MissingMappingAction` property of the data adapter has three settings:

- `Passthrough` (the default): If a mapping is not specified, the data adapter will assume the name is the same in the data source and the `DataSet`.
- `Error`: If a mapping is not specified, the data adapter will raise a `System.Exception`.
- `Ignore`: If a mapping is not specified, the `DataAdapter` object will ignore that column.

Moreover, when the `Fill()` and the `Update()` methods are used, you can choose what the data adapter should do when a `DataSet` schema doesnÕt meet expectations. The `MissingSchemaAction` property of the data adapter can accept four values:

- `Add` (default option): When the schema is missing for the current column, the data adapter will create it and add it to the `DataSet` object without creating information on primary keys or unique columns.
- `AddWithKey`: This is the same as the `Add` option, but with the difference that primary keys and unique columns will be created. Remember that the identity column will be created without identity seed and identity increment values. You should add them after the `Fill()` or the `Update()` calls.
- `Ignore`: When the schema is missing for the current column, the data adapter will ignore it and continue analyzing the other columns.
- `Error`: When the schema is missing for the current column, the data adapter will raise an exception.

These settings also apply when using the data adapter to update a data source, which will

CHAPTER 8

Sorting, Searching, and Filtering

In the last two chapters, you examined various objects that allow you to fetch data from a data source and store it in a disconnected manner within your application. You saw how `DataSets` allow you to represent a disconnected cache of relational data in memory. If a `DataSet` is a relational representation of data, then `DataTables` along with various constraints provide you with a representation of tabular data extracted out of a data source, and `DataRelations` allow you to create relational structures between `DataTables`.

The other relational representation of data you'll frequently come across is any common modern-day relational database management system (RDBMS). However, a `DataSet` is not a database. It's merely a surrogate that allows you to keep data in a portable cache while remaining disconnected from the underlying data source. It's like a mini in-memory database represented by an object instance. What this means to you as an application developer is that instead of relying on some flavor of the popular structured query language (SQL), you have to rely upon a method of working with disconnected data using an object model instead.

ADO.NET allows you to perform several operations on these structures, such as sorting, searching, and filtering on this in-memory relational structure, using an object representation rather than a database engine. This chapter will discuss the various objects and facilities ADO.NET provides that allow you to sort, search, and filter disconnected data held in a `DataSet` or `DataTable` object.

The discussion will first focus on the `DataTable` and `DataSet` objects themselves, and will elucidate with examples of what facilities these objects provide for you to work with the data contained within them. Then a new object will be introduced, the `DataView` object, which is a view on a `DataTable` but has varied uses in a typical Windows Forms or ASP.NET application.

Note It's tempting to think of the relationship between a `DataView` and a `DataTable` as the same as the relationship between a database view and a database table. That simile is close and helpful in understanding the concept, but it's not exactly the same. While a `DataTable` is simply tabular data which might be the result of a join between one or more tables from the underlying data source, a `DataView` lets you create a view on only one `DataTable` as a subset of the rows (and not columns), whereas a database view lets you create tabular data as a subset of both rows and columns from one or more tables. Therefore, a `DataTable` and a `DataView` aren't exactly like a Table or a View, but they are similar.

Figure 8-2. Sample data for this chapter

Now that the data source is set up, let's examine the various facilities the DataTable object provides that help you work with disconnected data easily.

Working in a DataTable

The DataTable object is an in-memory representation of tabular data. It can be filled as a result of a join between one or more tables in a database. What that means is that the data inside a DataTable is logically arranged with in-memory data structures that represent columns and rows. Also, there could be various constraints specified on the DataTable helping to keep the integrity of the data. Specifically, as you saw in Chapter 6, referential integrity is enforced using foreign-key constraints and data integrity is enforced using unique constraints.

Listing 8-9. Using Select to Find Multiple Rows Matching a Criterion in C#

```
static void Main(string[] args)
{
    DataTable myTable = DataSetFiller.FillDataset(dataFilePath).Tables[0];
    DataRow[] drs = myTable.Select("FirstName Like 'Jo%'");
    if (drs != null)
    {
        Console.WriteLine("Find rows using a  base DataSet");
        foreach (DataRow dr in drs)
        {
            ShowDataRow(dr);
        }
    }
}
```

Listing 8-10. Using Select to Find Multiple Rows Matching a Criterion in Visual Basic .NET

```
Sub Main()
    Dim myTable As DataTable = DataSetFiller.FillDataset(dataFilePath).Tables(0)
    Dim drs() As DataRow = myTable.Select("FirstName Like 'Jo%'")
    If drs IsNot Nothing Then
        For Each dr as DataRow In drs
            ShowDataRow(dr)
        Next
    End If
End Sub
```

Another thing that a database lets you do is return the results in a predefined sort order. You could query the data to find all matching rows where `LastName` ends with `OfJungle`, but say you want the results to be sorted by `FirstName`. This can be easily achieved by slightly modifying the code shown in Listings 8-11 and 8-12. All you need to do is replace the call to `Select` with a proper overload that allows you to specify a sort order. Also, you would have to replace the first parameter (the `where` clause passed to the `Select` method) with the appropriate search string. This can also be seen in Exercise 8.3 of the associated code download.

Listing 8-11. Using Select to Find Rows and Specify a Sort in C#

```
static void Main(string[] args)
{
    DataTable myTable = DataSetFiller.FillDataset(dataFilePath).Tables[0];
    DataRow[] drs = myTable.Select("LastName Like '%OfJungle'","FirstName ASC");
    if (drs != null)
    {
        foreach (DataRow dr in drs)
        {
            ShowDataRow(dr);
        }
    }
```

Thus, you pass in a DataViewRowState constant to identify the rows in a particular state that you're interested in querying. Even though DataViewRowState constants and DataRowState constants will be covered in greater depth in Chapters 9 and 10, for now it's sufficient to understand that the previous statement allows you to find the modified rows in the DataTable. You might also notice a call to AcceptChanges. This is important since when the DataSet or DataTable is read from XML, the default RowState for all those newly added rows is DataRowState.Added. By calling AcceptChanges, you effectively accept the suggested changes to the DataTable, and hence change all the RowStates to UnChanged. You can find the full code for this example in Exercise 8.4.

Frequently in a database, you might have a certain piece of information that can be derived definitively from a set of columns. In other words, you might need to calculate a certain column on the fly as a part or expression based on other columns in the result set.

Expressions: Calculating Columns on the Fly

Sometimes you may want to make calculations from columns of a tabular result set that produce values on the fly and appear as a column contained in the result. A real-world scenario would be to say that the Products table contains a price column and a tax column; what if you wanted to create a result set, with an additional column named "total price," as Price + Price * TaxPercent?

This can be easily achieved in the database with a SQL query that looks like

```
Select
    ProductID, Price, Tax, (Price + Price * TaxPercent) as TotalPrice
from Products;
```

Here, an expression is specified for TotalPrice. The database then calculates the TotalPrice for each row and includes that in the result set for you.

You can easily achieve the same result in a DataTable by specifying an expression on a new column you need to create. To emphasize the fact that this would be done on the fly for each column on the DataTable, this next example shall use a DataTable databound to a DataGridView on a Windows Forms application. This can be achieved in the following steps:

1. Start by creating a Windows Forms application. Name it Exercise 8.5. Also, change the text on the main form of the application to Exercise 8.5.

2. Drop a DataGridView on the form, make sure its Dock property is set to Dock.Fill. This will be named dataGridView1 by default.

3. In the Form1_Load event handler, which will be called when Form1 loads, add the code shown in Listings 8-13 and 8-14.

Listing 8-13. Creating a Calculated Column in C#

```
DataSet customerProducts =
CreateDataSet.DataSetFiller.FillStrongDataSet(dataFilePath);
DataTable productsTable = customerProducts.Tables[1];

DataColumn totalPrice = new DataColumn("Total Price");
totalPrice.Expression = "Price + Price * TaxPercent";
productsTable.Columns.Add(totalPrice);
```

Listing 8-14. Creating a Calculated Column in Visual Basic .NET

```
Dim customerProducts As DataSet = _
    CreateDataSet.DataSetFiller.FillStrongDataSet(dataFilePath)
Dim productsTable As DataTable = customerProducts.Tables(1)

Dim totalPrice As DataColumn = New DataColumn("Total Price")
totalPrice.Expression = "Price + Price * TaxPercent"
productsTable.Columns.Add(totalPrice)
dataGridView1.DataSource = productsTable
```

4. Compile and run the application. You should see output as shown in Figure 8-3.

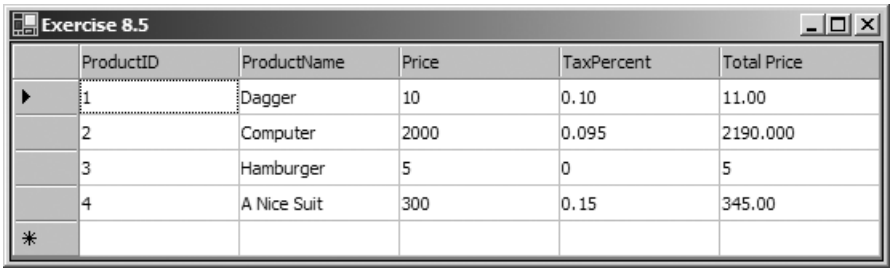


Figure 8-3. Calculated columns using the Expression property

As you can see in Figure 8-3, the column TotalPrice is calculated properly as per the expression you specified and is included in the DataTable. So what exactly is this code doing?

Well first of all, as you can see, the DataSet was filled using the FillStrongDataSet method instead of the FillDataSet method. This is critical since, for expressions to work, they need the data types set on the various columns. You cannot do a multiply operation using the * operand on two System.String column types, which is the default column type for a basic DataColumn. You could have instead chosen to specify the DataType property on a basic DataColumn; however, such an operation needs to be done before itÕs filled with data. This can be achieved easily using the following lines of code:

C#

```
productsTable.Columns["Price"].DataType = typeof(System.Int32);
// Fill Data here
```

VB.NET

```
productsTable.Columns("Price").DataType = GetType(System.Int32)
' Fill Data here
```

Or, you could simply implicit cast a basic DataSet back from a strongly typed DataSet, which is what is done in the following lines of code:

C#

```
DataSet customerProducts =
    CreateDataSet.DataSetFiller.FillStrongDataSet(dataFilePath);
```

VB.NET

```
Dim customerProducts As DataSet = _
    CreateDataSet.DataSetFiller.FillStrongDataSet(dataFilePath)
```

Note that implicit casting doesn't really require you to do anything, you can simply assign an inherited object instance back to the base. With the DataTable set up, the next three lines add a new column, totalPrice, set an expression on it, and add it to the DataTable:

C#

```
DataColumn totalPrice = new DataColumn("Total Price");
totalPrice.Expression = "Price + Price * TaxPercent";
productsTable.Columns.Add(totalPrice);
```

VB.NET

```
Dim totalPrice As DataColumn = New DataColumn("Total Price")
totalPrice.Expression = "Price + Price * TaxPercent"
productsTable.Columns.Add(totalPrice)
```

That's it! The next line simply data binds the DataTable to the DataGridView to display the results:

C#

```
dataGridView1.DataSource = productsTable;
```

VB.NET

```
dataGridView1.DataSource = productsTable
```

This approach works well for situations where you need to perform a calculation on every row in a DataTable. In certain situations, however, you might need to perform aggregate calculations on the entire DataTable. Luckily for such cases, the DataTable supports a method called Compute

Performing Aggregate Calculations

Say you needed to perform a calculation over the rows of a table. In our case, say you needed to add the Price column for all rows and come up with a total of prices for all products.

If this were a database, you could simply write a query that looks like this:

```
Select Sum(Price) from Products
```

This query would go over all the rows and sum up their Price columns one by one to return a total. In a DataTable, however, you can achieve the same result using the Compute method. This is demonstrated in Listings 8-15 and 8-16. In this code, btnSumPrices is a button which, when clicked, computes the aggregate sum function on the Price column and puts the calculated

5. Just as you were able to find ChildRows, you can easily find the ParentRow for any DataRow using the GetParentRow method. That is what the application does when you click the btnFilter2 button. For each identified row in the lbCustomerProducts listbox, it finds the relevant ParentRow for each such row, and populates those into the lbProducts listbox. The code for this can be seen in Listings 8-19 and 8-20.

   Listing 8-19. Finding Parent Rows for Various Rows in C#

```
private void btnFilter2_Click(object sender, EventArgs e)
{
    DataRow custProdRow;
    DataRow prodRow;
    DataTable custProdTable = customerProducts.Tables["CustomerProducts"] ;
    lbProducts.Items.Clear();
    foreach (object item in lbCustomerProducts.Items)
    {
        int custProdId = (int)item;
        custProdRow = custProdTable.Rows.Find(custProdId);
        prodRow = custProdRow.GetParentRow(customerProducts.Relations[0]);
        lbProducts.Items.Add(ProdRow["ProductName"]);
    }
}
```

   Listing 8-20. Finding Parent Rows for Various Rows in Visual Basic .NET

```
Private Sub btnFilter2_Click( _
  ByVal sender As System.Object, ByVal e As System.EventArgs) _
  Handles btnFilter2.Click
    Dim custProdRow As DataRow
    Dim ProdRow As DataRow
    Dim custProdTable As DataTable = _
        customerProducts.Tables("CustomerProducts")
    lbProducts.Items.Clear()
    Dim item As Object
    For Each item In lbCustomerProducts.Items
        Dim custProdId As Integer = CType(item, Integer)
        custProdRow = custProdTable.Rows.Find(custProdId)
        ProdRow = custProdRow.GetParentRow(customerProducts.Relations(0))
        lbProducts.Items.Add(ProdRow("ProductName"))
    Next
End Sub
```

6. Compile and run the application. Select ÒTarzan KingOfJungleÓ from lbCustomers and click btnFilter1 ; you should see output as shown in Figure 8-6.

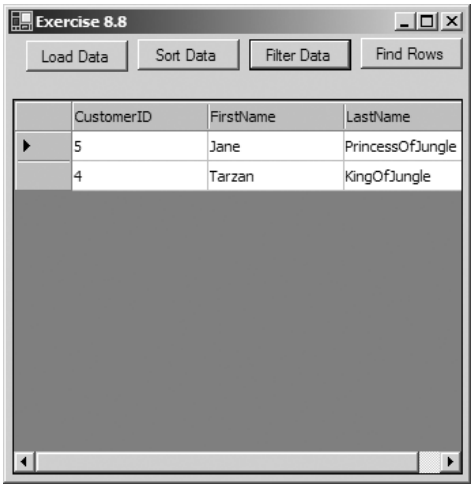


Figure 8-11.Data loaded, sorted, and filtered in CustomersView DataView and databound to a DataGridView control

6. Finally, the application will find the rows based on the sort key. Since our sort key is FirstName, letÕs try and find the names of customers who match the FirstName ÒTarzanÓ. This can be done using the following code snippet:

C#

```
private void btnFindRows_Click(object sender, EventArgs e)
{
    DataRowView[] drvs = CustomersView.FindRows("Tarzan");
    foreach (DataRowView drv in drvs)
    {
        MessageBox.Show(
            drv.Row["FirstName"] + " " + drv.Row["LastName"], "Selected Item");
    }
}
```

VB.NET

```
Private Sub btnFindRows_Click( _
   ByVal sender As System.Object, ByVal e As System.EventArgs) _
   Handles btnFindRows.Click
    Dim drvs() As DataRowView = CustomersView.FindRows("Tarzan")
    Dim drv As DataRowView
    For Each drv In drvs
        MessageBox.Show( _
            drv.Row("FirstName") & " " & drv.Row("LastName"), "Selected Item")
    Next
End Sub
```

Converting a DataView to a DataTable

A notable enhancement introduced in .NET 2.0 is that a method has been added to the DataViewobject that easily lets you create a new DataTable based on the rows available in a DataView This is incredibly helpful in extracting a smaller set of the rows in a DataTable as a brand new DataTable. The usage is fairly simple and is shown here:

C#

```
DataTable custProdTable = DataSetFiller.FillDataset(dataFilePath).Tables(2) ;
DataView view = new DataView[custProdTable];
view.RowFilter = "ProductID > 2";
DataTable subsetTable = view.ToTable[];
// or
DataTable subsetTable = view.ToTable["TableName"];
```

VB.NET

```
Dim custProdTable As DataTable = _
    DataSetFiller.FillDataset(dataFilePath).Tables(2)
Dim view as DataView = New DataView(custProdTable)
view.RowFilter = "ProductID > 2"
Dim subsetTable as DataTable = view.ToTable()
' or
Dim subsetTable as DataTable = view.ToTable("TableName")
```

Either of these result in a DataTable with the same number of columns as the original DataTable. There is, however, another overload that lets you limit the number of columns you get back in the result set.

Where this method is incredibly useful is in finding distinct result sets, akin to the DISTINCT keyword in SQL. For instance, if you wanted to find distinct ProductIDs in the CustomerProducts table, you could easily achieve that using the following code snippet (note that the first parameter to the ToTablemethod signifies that you want the resulting DataTableto contain distinct rows):

C#

```
DataTable custProdTable = DataSetFiller.FillDataset(dataFilePath).Tables(2) ;
DataView view = new DataView[custProdTable];
view.RowFilter = "ProductID > 2";
DataTable subsetTable = view.ToTable[true,"ProductID"];
```

VB.NET

```
Dim custProdTable As DataTable = _
    DataSetFiller.FillDataset(dataFilePath).Tables(2)
Dim view as DataView = New DataView(custProdTable)
view.RowFilter = "ProductID > 2"
Dim subsetTable as DataTable = view.ToTable(True,"ProductID")
```

Of course, as you can see, this overload can also be used in extracting aDataTable with fewer numbers of columns than the original. This can be done by changing the first parameter

ows, as shown here:

7. Now go ahead and make some change to the DataTable. For instance, change the name Bill to Chill; you'll see that the XmlDataDocument automatically understands the change and the resultant XML reflects the contents of the DataTable without any additional plumbing. This is shown in Figure 8-16.

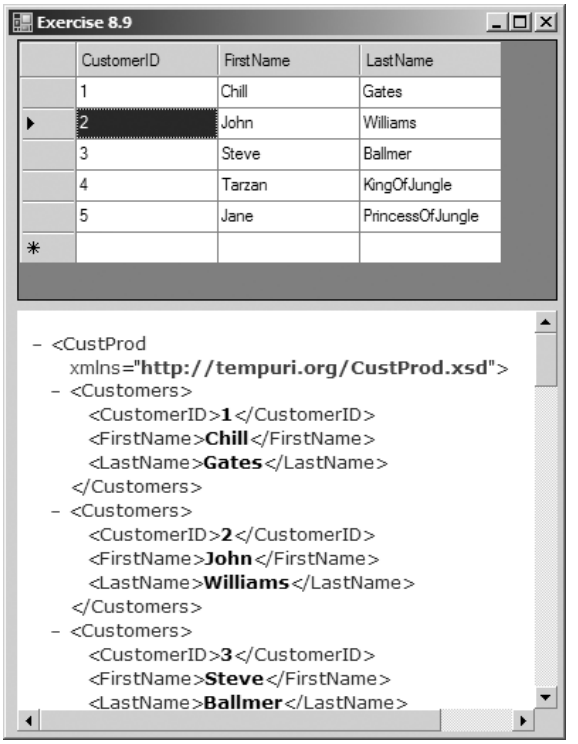


Figure 8-16. XmlDataDocument keeps the XmlDocument and DataSet in sync

In addition to this, you can leverage standard XML features such as XPath queries to query across the DataSet's data.

ADO.NET itself contains a lot of other XML-related functionality. Some of this functionality is around the new XML data type introduced in Microsoft SQL Server 2005, or there are functions such as ExecuteXmlReader on the SqlClient .NET data provider. Finally, there is a .NET data provider at System.Data.SqlXml, which is primarily used to leverage the XML features available in the Microsoft SQL Server database. These will be covered in further depth in Chapter 12, where XML and ADO.NET are discussed.

CHAPTER 9

Updating Data

Updating data back into the data source is probably one of the most interesting parts of any application. This is because updating data back into the data source requires you, as an architect, to make many decisions usually guided between a balance of effort, level of concurrency support, performance, and data sanctity. ItÕs also one of the most critical pieces of your application as far as efficiency and the right design are concerned. Before architecting the updating part of your application, you need to ask yourself a few questions.

These questions will basically be centered on what kind of data sanctity or integrity you require out of your data layer in a highly concurrent environment. Based upon that you could decide on the concurrency management scheme to use. When you do decide on the concurrency management scheme, youÕll then have to evaluate how exactly you wish to build the concurrency detection and conflict resolution features in your data layer. And to build such concurrency features, youÕll need to decide what built-in ADO.NET features you wish to leverage.

After you decide on the concurrency management schemes, the next big questions to ask yourself regard the performance you wish to garner out of the data layer. Do you wish to rely on autogenerated commands, which might be designed in a one-size-fits-all approach? Do you wish to cache commands? Or command parameters? Do you want to design specific supplied commands that take the specific table structures and business case into cognizance at the risk of a higher management hassle for all the commands? Or do you wish to use database-specific features and pass the data in bulk forms to the database server, in such a way that the solution is the least portable between databases, but the most efficient?

Finally, one of the most important questions you need to ask yourself while architecting the data layer is ÒWhat am I designing this data layer for, and how much effort am I willing to put in it?Ó Is coming up with the best and most efficient way to persist data really worth it for a single-user desktop application?

This chapter and the next donÕt give you a panacea for all your updating needs; I wish there were such a thing. The exact solution really depends on your needs. These chapters, however, do give you enough information for you to intelligently decide what might suit your needs the best.

As it turns out, just like in real life, ADO.NET has no freebies. Like anything else, the more effort you put in, the better your data layer will perform and the cleaner your data will be. Given that you might have many choices at each step as far the previous three questions go, it will pay off well if you know what choices you have, and how much work each one of those choices will entail.

Knowing about the various methods ADO.NET provides to update changes or new data into your database will allow you to make intelligent decisions regarding these three questionsÑand that is what this chapter and the next are all about.

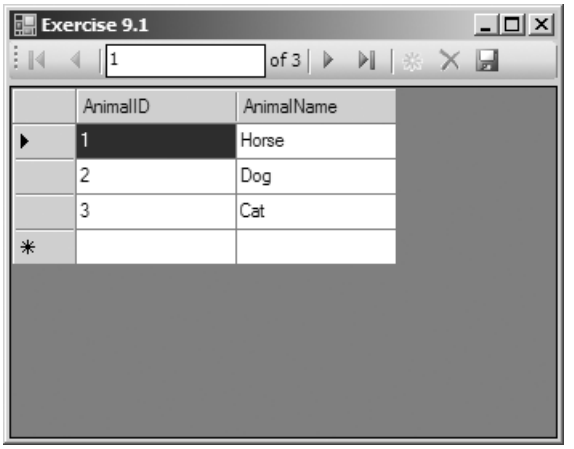


Figure 9-3.A few rows added and saved to the database

What just happened? How did a single line of code end up doing so much work? To discover the answers, letÕs examine the objects added for us one by one.

To get into the real meat of the matter, select Show All Files in your Visual Studio IDE, as shown in Figure 9-4.

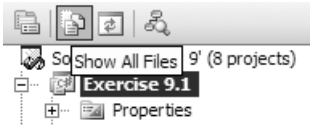
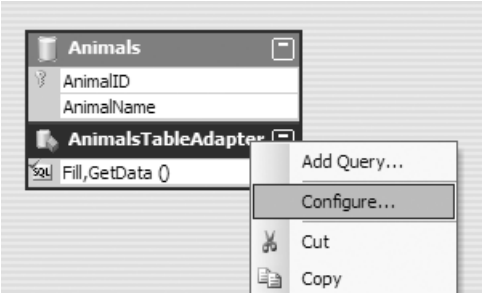


Figure 9-4.The Show All Files button

The first object added is the testDataSet . If you double-click it, youÕll see that it contains two objects: the Animals DataTable and the AnimalsTableAdapter TableAdapter. If you right-click the AnimalsTableAdapter and choose Configure, as shown in Figure 9-5, youÕll see a dialog box as shown in Figure 9-6.



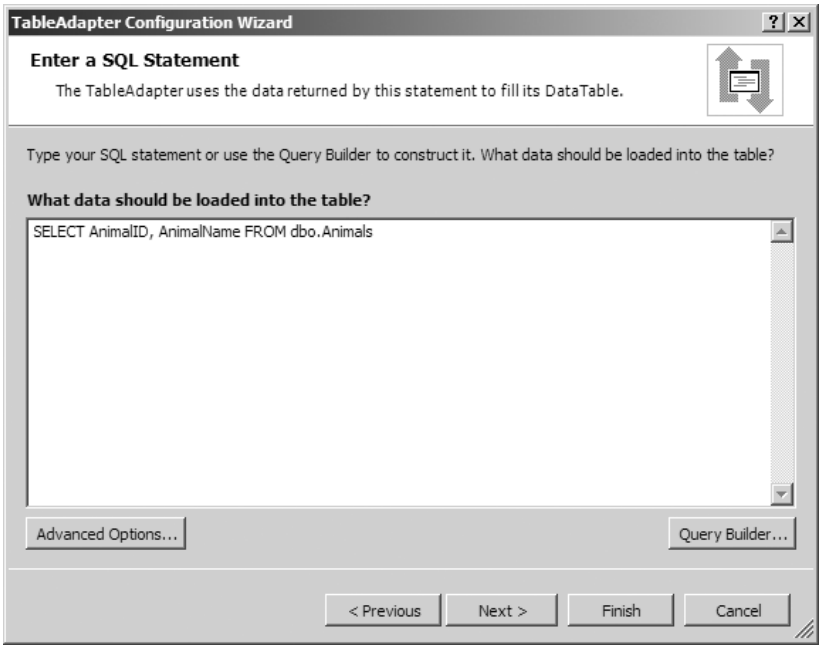


Figure 9-6. TableAdapter Configuration dialog box

In this dialog box, you can easily configure the SELECT query being used to fill the testDataSet at runtime. Not only that, but by clicking Advanced Options you can see a dialog box as shown in Figure 9-7.

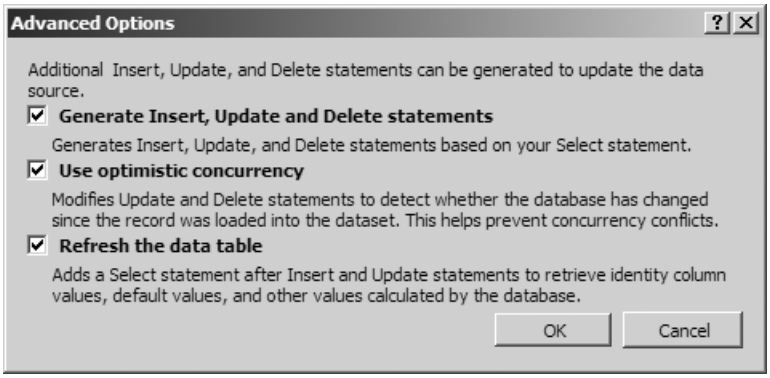


Figure 9-7. TableAdapter Configuration dialog boxÑAdvanced Options

In the Advanced Options dialog box, you are presented with three options. The exact underlying behavior of these three options will become clear as you progress through this and the next chapter, but for now it is sufficient to understand that these checkboxes allow you to do the following:

So basically, the second user cannot save his data unless he uses the correct keys. But how will the second user know what keys to useunless he refreshes the data?

This can be achieved by refreshing the data from the data source after an attempted update has failed. As a matter of fact, it makes sense to refresh the data from the underlying data source even if an update succeeds because the user might wish to add further rows or edit any existing rows, including the ones entered by another user. Also, by refreshing data you get the latest snapshot including the last generated primary key, and your next generated disconnected primary key in the DataTable has a lower probability of being incorrectÑthis way, you save yourself the pain of an exception and yet another refresh.

As it turns out, ADO.NET does give you enough control to automatically resolve such conflicts. This will become clearer as you read through the various examples in this chapter and the next. For now itÕs sufficient to understand that the disconnected model introduces some interesting new issues that usually wouldnÕt arise with a constantly connected application.

How Does It All Work?

This is such an obvious question especially since you didnÕt write even one line of code.

How could just dragging and dropping possibly take care of generating various queries for you for updates, inserts, and deletes; deciding between updates, inserts, and deletes on various rows in a DataTable; and, at the same time, even take care of concurrency issues?

Note What you are just about to do is degut a lot of autogenerated code. This comes with good news an bad news. The bad news first: there is a lot of code to look atÑalmost like a wall of ASCII, so my intention i this section is to give you a guided tour through this jungle of code. The good news is that you donÕt have memorize every little bit of this code, just view it conceptually to try and understand what it does functionall

---

Earlier in Figure 9-4 you enabled the Show All Files button. By doing so, under the Solution Explorer window, you would notice a file created for you by Visual Studio, which contains the actual code for the strongly typed DataSet. This file is the testDataSet.Designer.cs / testDataSet. Designer.vb file. In addition to testDataSet and AnimalsDataTable definitions, something you have already seen in Chapter 7, there lies a definition for AnimalsTableAdapter. The various methods in AnimalsTableAdapter are shown in Listings 9-1 and 9-2.

Listing 9-1. Various Methods in AnimalsTableAdapter in C#

```
public interface IAnimalsTableAdapter {
    int Fill(testDataSet.AnimalsDataTable dataTable);
    testDataSet.AnimalsDataTable GetData();
    int Delete(int Original_AnimalID, string Original_AnimalName);
    int Insert(int @AnimalID, string @AnimalName);
    int Update(int @AnimalID, string @AnimalName, int @Original_AnimalID);
    int Update(testDataSet dataSet);
    int Update(testDataSet.AnimalsDataTable dataTable);
    int Update(System.Data.DataRow[] dataRows);
    int Update(System.Data.DataRow dataRow);
```

Listing 9-2. Various Methods in AnimalsTableAdapter in Visual Basic .NET

```
Public Interface IAnimalsTableAdapter
    Function Fill(ByVal dataTable As testDataSet.AnimalsDataTable) As Integer
    Function GetData() As testDataSet.AnimalsDataTable
    Function Delete(ByVal Original_AnimalID As Integer, _
      ByVal Orignal_AnimalName as String) As Integer
    Function Insert(ByVal AnimalID As Integer, ByVal AnimalName As String) _
      As Integer
    Function Update(ByVal AnimalID As Integer, ByVal AnimalName As String, _
      ByVal Original_AnimalID As Integer) As Integer
    Function Update(ByVal dataSet As testDataSet) As Integer
    Function Update(ByVal dataTable As testDataSet.AnimalsDataTable) As Integer
    Function Update(ByVal dataRows() As System.Data.DataRow) As Integer
    Function Update(ByVal dataRow As System.Data.DataRow) As Integer
End Interface
```

Inside the AnimalsTableAdapter class, you have a definition for each one of these methods. First, letÕs focus our attention on the Fill method, which fills the AnimalsDataTable. The code for a sample Fill method is shown in Listings 9-3 and 9-4.

Listing 9-3. AnimalsTableAdapter.Fill Method in C#

```
public virtual int Fill(testDataSet.AnimalsDataTable dataTable) {
    this.Adapter.SelectCommand =
        ((System.Data.SqlClient.SqlCommand)(this.CommandCollection[0]));
    if ((this.m_clearBeforeFill == true)) {
            dataTable.Clear();
    }
    int returnValue = this.Adapter.Fill(dataTable);
    return returnValue;
}.
```

Listing 9-4. AnimalsTableAdapter.Fill Method in Visual Basic .NET

```
Public Overloads Overridable Function Fill( _
  ByVal dataTable As testDataSet.AnimalsDataTable) As Integer _
  Implements IAnimalsTableAdapter.Fill
    Me.Adapter.SelectCommand = _
      CType(Me.CommandCollection(0),System.Data.SqlClient.SqlCommand)
    If (Me.m_clearBeforeFill = true) Then
        dataTable.Clear
    End If
    Dim returnValue As Integer = Me.Adapter.Fill(dataTable)
    Return returnValue
End Function
```

Upon closer examination of this code, it looks very similar to the code you have already seen in various listings of Chapter 7. This is a simple case of specifying a SelectCommand to a

ut what is this.CommandCollection[0] ?

Interestingly, if you browse through the code for AnimalsTableAdapter a little bit, you'll find that this command is a SqlCommand representation of the command text you specified in Figure 9-5. The code for setting up such commands can be found in the InitCommandCollection method. This can be seen in Listings 9-5 and 9-6.

Listing 9-5. AnimalsTableAdapter.InitCommandCollection Method in C#

```
private void InitCommandCollection() {
    this.m_commandCollection = new System.Data.SqlClient.SqlCommand[1];
    this.m_commandCollection[0] = new System.Data.SqlClient.SqlCommand();
    this.m_commandCollection[0].Connection = this.Connection;
    this.m_commandCollection[0].CommandText =
        "SELECT AnimalID, AnimalName FROM dbo.Animals";
    this.m_commandCollection[0].CommandType = System.Data.CommandType.Text;
}
```

Listing 9-6. AnimalsTableAdapter.InitCommandCollection Method in Visual Basic .NET

```
Private Sub InitCommandCollection()
    Me.m_commandCollection = New System.Data.SqlClient.SqlCommand(0) {}
    Me.m_commandCollection(0) = New System.Data.SqlClient.SqlCommand
    Me.m_commandCollection(0).Connection = Me.Connection
    Me.m_commandCollection(0).CommandText = _
      "SELECT AnimalID, AnimalName FROM dbo.Animals"
    Me.m_commandCollection(0).CommandType = System.Data.CommandType.Text
End Sub
```

CommandCollection is an array because one TableAdapter can hold multiple commands. Next, look up the code for the definition of the Delete(int Original_AnimalID, string Original_AnimalName) overload. This is shown in Listings 9-7 and 9-8.

Listing 9-7. AnimalsTableAdapter.Delete(int Original_AnimalID, string Original_AnimalName) Method in C#

```
public virtual int Delete(int Original_AnimalID, string Original_AnimalName) {
    this.Adapter.DeleteCommand.Parameters[0].Value = ((int)(Original_AnimalID));
    if ((Original_AnimalName == null)) {
        throw new System.ArgumentNullException("Original_AnimalName");
    }
    else {
        this.Adapter.DeleteCommand.Parameters[1].Value =
            ((string)(Original_AnimalName));
    }
    System.Data.ConnectionState previousConnectionState =
        this.Adapter.DeleteCommand.Connection.State;
    this.Adapter.DeleteCommand.Connection.Open();
    try {
        return this.Adapter.DeleteCommand.ExecuteNonQuery();
```

```
    new System.Data.SqlClient.SqlParameter("@Original_AnimalID",
    System.Data.SqlDbType.Int,
    0, System.Data.ParameterDirection.Input,
    0, 0, "AnimalID", System.Data.DataRowVersion.Original,
    false, null, "", "", ""));
this.m adapter.DeleteCommand.Parameters.Add(
    new System.Data.SqlClient.SqlParameter("@Original_AnimalName",
    System.Data.SqlDbType.VarChar, 0,
    System.Data.ParameterDirection.Input, 0, 0,
    "AnimalName", System.Data.DataRowVersion.Original,
    false, null, "", "", ""));
```

Listing 9-10. DeleteCommand in Visual Basic .NET

```
Me.m_adapter.DeleteCommand = New System.Data.SqlClient.SqlCommand
Me.m_adapter.DeleteCommand.Connection = Me.Connection
Me.m_adapter.DeleteCommand.CommandText = _
    "DELETE FROM [dbo].[Animals] WHERE (([AnimalID] = @Original_AnimalID) " & _
    " AND ([Animal" & _
      "Name] = @Original_AnimalName))"
Me.m_adapter.DeleteCommand.CommandType = System.Data.CommandType.Text
Me.m_adapter.DeleteCommand.Parameters.Add( _
    New System.Data.SqlClient.SqlParameter("@Original_AnimalID",  _
    System.Data.SqlDbType.Int, 0, System.Data.ParameterDirection.Input, 0, 0,  _
    "AnimalID", System.Data.DataRowVersion.Original, false, Nothing, "", "", ""))
Me.m_adapter.DeleteCommand.Parameters.Add( _
    New System.Data.SqlClient.SqlParameter("@Original_AnimalName",  _
    System.Data.SqlDbType.VarChar, 0, _
    System.Data.ParameterDirection.Input, 0, 0, _
      "AnimalName", System.Data.DataRowVersion.Original,  _
    false, Nothing, "", "", ""))
```

As a matter of fact, if you look through the code of InitAdapter, the various table mappings, adapters, and commands are set up within this private method. The full code for InitAdapter is shown in Listings 9-11 and 9-12.

Listing 9-11. InitAdapter in C#

```
private void InitAdapter() {
    this.m_adapter = new System.Data.SqlClient.SqlDataAdapter();
    System.Data.Common.DataTableMapping tableMapping =
        new System.Data.Common.DataTableMapping();
    tableMapping.SourceTable = "Table";
    tableMapping.DataSetTable = "Animals";
    tableMapping.ColumnMappings.Add("AnimalID", "AnimalID");
    tableMapping.ColumnMappings.Add("AnimalName", "AnimalName");
    this.m_adapter.TableMappings.Add(tableMapping);
    this.m_adapter.DeleteCommand = new System.Data.SqlClient.SqlCommand();
```

```
this.m_adapter.DeleteCommand.CommandText =
    "DELETE FROM [dbo].[Animals] WHERE (([AnimalID] = @Original_AnimalID) AND"
    + "([Animal" + "Name] = @Original_AnimalName))";
this.m_adapter.DeleteCommand.CommandType = System.Data.CommandType.Text;
this.m_adapter.DeleteCommand.Parameters.Add(
    new System.Data.SqlClient.SqlParameter("@Original_AnimalID",
    System.Data.SqlDbType.Int, 0, System.Data.ParameterDirection.Input, 0, 0,
    "AnimalID", System.Data.DataRowVersion.Original,
    false, null, "", "", ""));
this.m_adapter.DeleteCommand.Parameters.Add(
    new System.Data.SqlClient.SqlParameter("@Original_AnimalName",
    System.Data.SqlDbType.VarChar, 0, System.Data.ParameterDirection.Input, 0,
     0, "AnimalName", System.Data.DataRowVersion.Original, false, null,
     "", "", ""));
this.m_adapter.InsertCommand = new System.Data.SqlClient.SqlCommand();
this.m_adapter.InsertCommand.Connection = this.Connection;
this.m_adapter.InsertCommand.CommandText =
"INSERT INTO [dbo].[Animals] ([AnimalID], [AnimalName]) " +
" VALUES (@AnimalID, @Animal" +
"Name);\r\nSELECT AnimalID, AnimalName FROM Animals WHERE " +
" (AnimalID = @AnimalID)";
this.m_adapter.InsertCommand.CommandType = System.Data.CommandType.Text;
this.m_adapter.InsertCommand.Parameters.Add(
    new System.Data.SqlClient.SqlParameter("@AnimalID",
    System.Data.SqlDbType.Int, 0, System.Data.ParameterDirection.Input, 0, 0,
    "AnimalID", System.Data.DataRowVersion.Current, false, null, "", "", ""));
this.m_adapter.InsertCommand.Parameters.Add(
    new System.Data.SqlClient.SqlParameter("@AnimalName",
    System.Data.SqlDbType.VarChar, 0, System.Data.ParameterDirection.Input, 0,
     0, "AnimalName", System.Data.DataRowVersion.Current,
     false, null, "", "", ""));
this.m_adapter.UpdateCommand = new System.Data.SqlClient.SqlCommand();
this.m_adapter.UpdateCommand.Connection = this.Connection;
this.m_adapter.UpdateCommand.CommandText =
"UPDATE [dbo].[Animals] SET [AnimalID] = @AnimalID, [AnimalName] = " +
" @AnimalName WHE" +
        "RE (([AnimalID] = @Original_AnimalID) AND ([AnimalName] = " +
        "@Original_AnimalName))" +
        ";\r\nSELECT AnimalID, AnimalName FROM Animals WHERE " +
        "(AnimalID = @AnimalID)";
this.m_adapter.UpdateCommand.CommandType = System.Data.CommandType.Text;
this.m_adapter.UpdateCommand.Parameters.Add(
    new System.Data.SqlClient.SqlParameter("@AnimalID",
    System.Data.SqlDbType.Int, 0, System.Data.ParameterDirection.Input, 0, 0,
    "AnimalID", System.Data.DataRowVersion.Current, false, null, "", "", ""));
this.m_adapter.UpdateCommand.Parameters.Add(
    new System.Data.SqlClient.SqlParameter("@AnimalName",
```

```
            0, "AnimalName", System.Data.DataRowVersion.Current,
            false, null, "", "", ""));
        this.m_adapter.UpdateCommand.Parameters.Add(
            new System.Data.SqlClient.SqlParameter("@Original_AnimalID",
            System.Data.SqlDbType.Int, 0, System.Data.ParameterDirection.Input, 0, 0,
            "AnimalID", System.Data.DataRowVersion.Original,
            false, null, "", "", ""));
        this.m_adapter.UpdateCommand.Parameters.Add(
            new System.Data.SqlClient.SqlParameter("@Original_AnimalName",
            System.Data.SqlDbType.VarChar, 0, System.Data.ParameterDirection.Input, 0,
            0, "AnimalName", System.Data.DataRowVersion.Original,
            false, null, "", "", ""));
    }
```

Listing 9-12. InitAdapter in Visual Basic .NET

```
Private Sub InitAdapter()
    Me.m_adapter = New System.Data.SqlClient.SqlDataAdapter
    Dim tableMapping As System.Data.Common.DataTableMapping = _
        New System.Data.Common.DataTableMapping
    tableMapping.SourceTable = "Table"
    tableMapping.DataSetTable = "Animals"
    tableMapping.ColumnMappings.Add("AnimalID", "AnimalID")
    tableMapping.ColumnMappings.Add("AnimalName", "AnimalName")
    Me.m_adapter.TableMappings.Add(tableMapping)
    Me.m_adapter.DeleteCommand = New System.Data.SqlClient.SqlCommand
    Me.m_adapter.DeleteCommand.Connection = Me.Connection
    Me.m_adapter.DeleteCommand.CommandText = _
    "DELETE FROM [dbo].[Animals] WHERE " & _
    " (([AnimalID] = @Original_AnimalID) AND ([Animal"& _
    "Name] = @Original_AnimalName))"
    Me.m_adapter.DeleteCommand.CommandType = System.Data.CommandType.Text
    Me.m_adapter.DeleteCommand.Parameters.Add( _
    New System.Data.SqlClient.SqlParameter("@Original_AnimalID", _
    System.Data.SqlDbType.Int, 0, System.Data.ParameterDirection.Input, 0, 0, _
     "AnimalID", System.Data.DataRowVersion.Original, _
    false, Nothing, "", "", ""))
    Me.m_adapter.DeleteCommand.Parameters.Add( _
    New System.Data.SqlClient.SqlParameter("@Original_AnimalName", _
    System.Data.SqlDbType.VarChar, 0, System.Data.ParameterDirection.Input, 0, _
     0, "AnimalName", System.Data.DataRowVersion.Original, _
    false, Nothing, "", "", ""))
    Me.m_adapter.InsertCommand = New System.Data.SqlClient.SqlCommand
    Me.m_adapter.InsertCommand.Connection = Me.Connection
    Me.m_adapter.InsertCommand.CommandText = _
    "INSERT INTO [dbo].[Animals] ([AnimalID], " & _
    " [AnimalName]) VALUES (@AnimalID, @Animal"& _
```

```
Console.WriteLine("Update Command = ");
Console.WriteLine(cmdBldr.GetUpdateCommand().CommandText);
```

VB.NET

```
Dim myDataAdapter As SqlDataAdapter = _
    New SqlDataAdapter("Select * from Animals2", connectionString)
Dim cmdBldr As SqlCommandBuilder = New SqlCommandBuilder(myDataAdapter)

cmdBldr.ConflictOption = ConflictOption.CompareRowVersion

Console.WriteLine("Update Command = ")
Console.WriteLine(cmdBldr.GetUpdateCommand().CommandText)
```

6. Compile and run the application, and examine the value of the update commandÕs text once again. You should receive results as shown here:

```
UPDATE [Animals2] SET [Animal2ID] = @p1, [AnimalName] = @p2
WHERE (([Timestamp] = @p3))
```

   Note that if you had not used a table with a `Timestamp` data type, you would have gotten an exception. The `Timestamp` data type column in SQL Server is a special column that changes its value every time the rowÕs value changes; therefore, itÕs a great way to check for concurrency conflicts in an efficient manner.

7. Now change the `ConflictOption` to `OverwriteChanges`, and examine the generated update commandÕs command text. It should look like as shown here:

```
UPDATE [Animals2] SET [Animal2ID] = @p1, [AnimalName] = @p2
WHERE (([Animal2ID] = @p3))
```

   In this case, youÕre instructing the command builder to not worry about concurrency and overwrite the previous values; therefore, the statement checks only for the primary key in its `WHERE` clause.

8. Thus, you can see how to leverage the `CommandBuilder` object to generate queries for you, plus you get some extra flexibility by using the `ConflictOption` property that you do not get in a pure drag-and-drop operation. This reinforces the notion that your data layer is as efficient and clean as the amount of effort you are willing to put into it. The more-effort approach of getting down and dirty and writing all your SQL yourself might be the best choice in a high-demand enterprise application anyway. Needless to say, 500 monkeys dragging and dropping 2,000 tables in an enterprise application will not necessarily be a good solution.

Now for a moment look at Listings 9-13 and 9-14 again. Specifically, this part:

C#

```
this.Adapter.Update(dataSet, "Animals");
```

VB.NET

```
Me.Adapter.Update(dataSet, "Animals")
```

As you have already seen, in Chapter 7, the DataAdapter object acts as the gatekeeper between the connected world and the disconnected world. Also you saw in Chapter 6 how the DataSet and DataTable objects not only hold data for you in a disconnected cache, but also maintain some history of the changes being done to the data. ItÕs that historical change information that lets the previous simple one-line statement decide what action needs to be taken on each row in each table of the DataSet.

State Management in a DataRow and Its Use in Updating Data

A DataSet allows you to store data fetched from an underlying data source or otherwise in a disconnected fashion. You have already been introduced to DataSets and fetching data into a DataSet.

But fetching data from the data source and holding it in a disconnected fashion is only half the story. Your application will probably make numerous changes to the data while keeping it disconnected from the underlying data source and finally reconnect using a data adapter via the Updatemethod call. Also, just by calling Updateon the data adapter, the application will also expect the underlying data adapter to magically understand which rows need to be updated, inserted, or deleted and expect the changes be persisted properly.

In this scenario of connect, fetch, disconnect, modify, reconnect, and persist changes, a number of things might need to be taken care of:

- You might need to figure out what rows were inserted, deleted, or updated.
- Other users could have saved their changes in the meantime, and the queried data you hold in a disconnected cache might not be valid anymore.
- The key values generated by you for newly inserted records might not be correct.
- After you do manage to do an update properly, you would probably need to fetch a refresh of the current data back to the application.

As it turns out, the DataSet, DataTable, and other objects that compose the disconnected part of ADO.NET do support a number of features that allow you to address all these issues appropriately.

At the heart of it all is the DataRow.RowStateproperty. The DataRow.RowStateproperty is of type DataRowStateenumeration. Table 9-1 shows you various values for the DataRowStateenumeration.

Table 9-1.The DataRowState Enumeration

Constant	Value	Description
Detached	1	A DataRowcan be connected to a maximum of one DataTableat any given time. If it is not connected to any DataTable, then the rowÕs state is detached.
UnChanged	2	This row was fetched from the data source and no changes have been

Constant	Value	Description
Deleted	8	This row was fetched from the data source and then deleted from the DataTable. During an Update, the data adapter will execute a DeleteCommand on it.
Added	4	This row was not fetched from the data source, it was added in the DataTable. During an Update, the data adapter will execute an InsertCommand on it.
Modified	16	This row was fetched from the data source but has been modified since. During an Update, the data adapter will execute an UpdateCommand on it.

As you can see from Table 9-1, ADO.NET can identify what happened to each row by examining its RowState. Also, you can see from Table 9-1 how the data adapterÕs Updatecommand is able to discern between what it needs to do to a row based upon the RowStateproperty of the DataRow

LetÕs examine this using an exercise that demonstrates all five row states. In this exercise, the database already contains some data as shown in Figure 9-10.

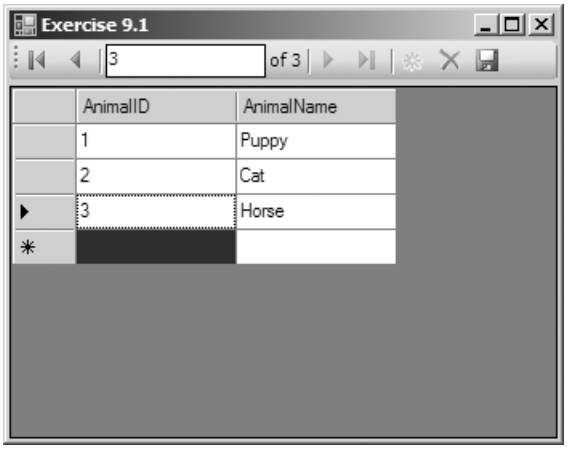


Figure 9-10.Current data in the database

This provides the application with enough use cases to simulate any of the row states in Table 9-1. In this exercise, the data shown in Figure 9-10 is fetched into a DataTable inside a Console application and various row states are examined. After that is done, the following changes will occur:

- The first row identifying a Puppy is changed to ÒDogÓ.
- The second row identifying a Cat is deleted.
- The third row identifying a Horse is left untouched.
- A fourth row is added that identifies a Camel.
- And a fifth row is created using the method, but itÕs not added to the DataTable.

6. Compile and run the application. You should see output that looks like Figure 9-12.

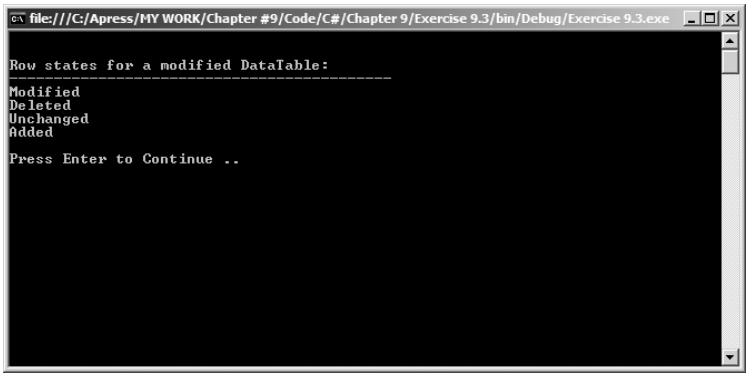


Figure 9-12.Row states for a modified DataTable

As you can see, the rows appear to have the correct row states clearly identifying which rows have been Modified, Deleted, and Added, and which ones that have been left Unchanged.

7. Now similar to the Camel row, create a new row, but don't add it to the animalsTable DataTable. Then check the row state of this new row:

C#

```
rowInQuestion = animalsTable.NewRow();
rowInQuestion["AnimalID"] = 5;
rowInQuestion["AnimalName"] = "Monkey";
```

VB.NET

```
rowInQuestion = animalsTable.NewRow()
rowInQuestion("AnimalID") = 5
rowInQuestion("AnimalName") = "Monkey"
```

When you compile and run the application, you can see the output the previous line produces as Monkey row's RowState Detached. Thus, when a DataRow is not associated or a part of any DataRowCollection as a Rows property of a DataTable, its RowState is Detached. Indeed, if you were to iterate through the various rows in animalsTable after this code executes, you would note that the rowInQuestion with AnimalName = "Monkey" is not a part of the animalsTable.

8. Next comes being able to update these changes back into the database. When the code calls sqlDa.Update and passes the animalsTable as a parameter, the data adapter will iterate through each row and call the InsertCommand for DataRowState.Added, UpdateCommand for DataRowState.Modified, and DeleteCommand for DataRowState.Deleted. The detached rows are not a part of the DataTable so they will be ignored, ed too.

For the rest of the rows, depending on the exact nature of the change, the RowState of the DataRow gets the appropriate value. Depending upon the RowState, the appropriate command as a property of the data adapter executes. The properties that hold the necessary commands are DeleteCommand, InsertCommand, and UpdateCommand.

But first you have to specify the DeleteCommand, UpdateCommand, and InsertCommand properties. Now you could specify any valid SQL command here, including stored procedures, but for the sake of simplicity, let's just go ahead and use the SqlCommandBuilder object covered in the previous section. The code can be seen in Listings 9-21 and 9-22.

Listing 9-21. Setting Various Commands in C#

```
// Update the changes back to the database.
SqlCommandBuilder cmbldr = new SqlCommandBuilder(sqlDa);

// Setup Update Command
sqlDa.UpdateCommand = cmbldr.GetUpdateCommand();
Console.WriteLine("Update Command: " + sqlDa.UpdateCommand.CommandText);

// Setup Insert Command
sqlDa.InsertCommand = cmbldr.GetInsertCommand();
Console.WriteLine("Insert Command: " + sqlDa.InsertCommand.CommandText);

// Setup Delete Command
sqlDa.DeleteCommand = cmbldr.GetDeleteCommand() ;
Console.WriteLine("Delete Command: " + sqlDa.DeleteCommand.CommandText);

sqlDa.Update(animalsTable);
```

Listing 9-22. Setting Various Commands in Visual Basic .NET

```
' Update the changes back to the database.
Dim cmbldr As SqlCommandBuilder = New SqlCommandBuilder(sqlDa)

' Setup Update Command
sqlDa.UpdateCommand = cmbldr.GetUpdateCommand()
Console.WriteLine("Update Command: " + sqlDa.UpdateCommand.CommandText)

' Setup Insert Command
sqlDa.InsertCommand = cmbldr.GetInsertCommand()
Console.WriteLine("Insert Command: " + sqlDa.InsertCommand.CommandText)

' Setup Delete Command
sqlDa.DeleteCommand = cmbldr.GetDeleteCommand()
Console.WriteLine("Delete Command: " + sqlDa.DeleteCommand.CommandText)

sqlDa.Update(animalsTable)
```

C#

```
DisplayRowStates("Final Row States:", animalsTable);
```

VB.NET

```
DisplayRowStates("Final Row States:", animalsTable)
```

The output should look like as shown in Figure 9-13.

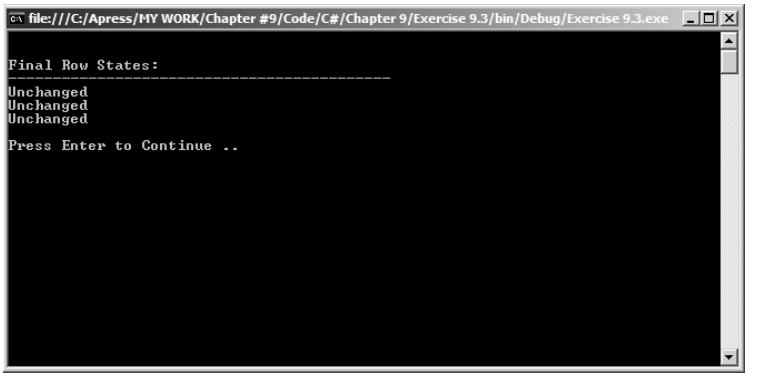


Figure 9-13.Final row states after a successful update

Thus, you can see from Figure 9-13 that after a successful update, the data adapter not only acts upon the right rows with the right commands, but it also changes the row states to Unchanged

12. There is one big drawback of this code. For three updates, you ended up making three database trips. There are multiple ways to get around this issue. One approach could be to use XML or a comma-delimited string passed directly into a stored procedure. But a rather easy quick-fix approach would be to use the UpdateBatchSizeproperty on the SqlDataAdapter. Just before the Updatestatement, add the UpdateBatchSizeas follows:

C#

```
sqlDa.UpdateBatchSize = 3;
```

VB.NET

```
sqlDa.UpdateBatchSize = 3
```

Now restore the Animals table in the database to the original three rows and rerun the application to view similar results. If you were, however, expecting to see one single batched SQL statement being sent to the database per the SQL Profiler trace, you might be in for a surprise. In fact, the SQL Profiler does not report any difference, but network roundtrips are still saved. Instead of implementing batched support in SQL, ADO.NET

Tabular Data Stream) protocol level.

C#

```
rowInQuestion = animalsTable.NewRow();
rowInQuestion["AnimalID"] = 4;
rowInQuestion["AnimalName"] = "Camel";
animalsTable.Rows.Add(rowInQuestion);
```

VB.NET

```
rowInQuestion = animalsTable.NewRow()
rowInQuestion("AnimalID") = 4
rowInQuestion("AnimalName") = "Camel"
animalsTable.Rows.Add(rowInQuestion)
```

If, instead, you're working with a DataView object attached to a DataTable, you can use the AddNew method on DataView to achieve the same results.

The second method to add a new DataRow into a DataTable is to use the LoadDataRow method. This method allows you to add a new DataRow and set its various values in one convenient method call. This is shown here:

C#

```
object[] rowVals = {"4", "Camel"} ;
animalsTable.LoadDataRow(rowVals, false);
```

VB.NET

```
Dim rowVals() As Object = {"4", "Camel"}
animalsTable.LoadDataRow(rowVals, False)
```

An interesting difference between these two methods is that the first method will add the new row and set its RowState property to DataRowState.Added. You can, at this point, call AcceptChanges to change the row state to DataRowState.Unmodified for all such rows in the DataTable or DataSet. The second method, however, allows you to pass in a second parameter, which if set to true, will not only add the row, but also set its RowState to DataRowState.Unmodified.

In addition, two methods exist for adding existing DataRows to a DataTable. The first is the ImportRow method, which simply takes a DataRow as a parameter, and the second is the Merge method, which merges two sets of disconnected data (DataSet, DataTable, DataRow array, etc.). The Merge method will be covered in detail later in this chapter.

Modify Existing Rows

Modifying existing rows is easy. You just find the right row and the right column and assign it a value. This is shown in the code snippet here:

3. Now with the form set up, let me present the logic for you before I delve too deeply into the actual code. The basic idea here is to generate dummy IDs for the AnimalID column when disconnected from the database. The real IDs will be generated in the database because the AnimalID column is an Identity column. Second, the AccountNumber column is not intended to be entered by the user, so the code can simply mark it as read only. The AccountNumber column is the column that I mentioned before—it's a column that has meaning in your business logic, therefore not a good candidate for a primary key in disconnected scenarios. Thus, the first thing you need to do is set up the animalsTable DataTable structure in Form1's constructor. This is shown in Listings 9-23 and 9-24. While you are setting up the DataTable's schema, also go ahead and data bind the animalsTable to dgView.

Listing 9-23. Setting Up the Empty Table's Schema in C#

```
public Form1()
{
    InitializeComponent();

    // Setup the schema for the Table.
    animalsTable = new DataTable("Animals");
    DataColumn dc = null;

    dc = new DataColumn("AnimalID");
    dc.Unique = true;
    dc.AutoIncrement = true;
    dc.AutoIncrementSeed = -1;
    dc.AutoIncrementStep = -1;
    dc.DataType = typeof(System.Int32);
    animalsTable.Columns.Add(dc);

    dc = new DataColumn("AnimalName");
    animalsTable.Columns.Add(dc);

    dc = new DataColumn("AccountNumber");
    dc.ReadOnly = true;
    animalsTable.Columns.Add(dc);

    // DataBind it, even though it has no rows in it yet.
    dgView.DataSource = animalsTable;
}
```

Listing 9-24. Setting Up the Empty Table's Schema in Visual Basic .NET

```
Private Sub Form1_Load( _
  ByVal sender As System.Object, ByVal e As System.EventArgs) _
  Handles MyBase.Load
    ' Setup the schema for the Table.
    animalsTable = New DataTable("Animals")
```

after the row has been updated. There's an equivalent event called `RowUpdating`, which is called before the row has been updated.

The `RowUpdated` event gets a `SqlRowUpdatedEventArgs` passed into it (in the case of OleDb or Oracle data providers, it is `OleDbRowUpdatedEventArgs` or `OracleRowUpdatedEventArgs`, respectively). Using the `RowUpdatedEventArgs` (the base class for `SqlRowUpdatedEventArgs`, `OleDbRowUpdatedEventArgs`, and `OracleRowUpdatedEventArgs`) you can easily identify the row that was updated last and execute an `ExecuteNonQuery` method to retrieve the key values.

As you might have guessed, due to the second database hit and event call, this method is significantly slower than the stored procedure or batched SQL methods. However, say if you were working with an Access database using the OleDb .NET data provider, you don't quite have the ability to create stored procedures with output parameters.

In that case, you would have to use the `RowUpdated` event on `OleDbDataAdapter` and use the `OleDbUpdatedEventArgs.Row` property to identify the row and act accordingly. You can also find the exact nature of the statement executed using `OleDbUpdatedEventArgs.StatementType` to differentiate between `INSERT`, `UPDATE`, or `DELETE`.

Now, again in Exercise 9.4, you might have noticed that at one point the code uses `@@IDENTITY` and at another point it uses `SCOPE_IDENTITY()`. Why is that?

As a footnote, I mentioned that there is a big difference between the two. The difference is more in the realm of SQL Server rather than ADO.NET, but since it might be used regularly in ADO.NET, I will mention the two briefly.

`@@IDENTITY` returns the last identity value generated on your connection. In other words, if your insert caused a trigger to fire that did another insert (say in an audit table), you'd end up getting the identity value of the audit table, not the table you did an insert in. That is obviously incorrect for our purposes. Thus, `SCOPE_IDENTITY()`, which is supported by SQL Server 2000 onward, should be used (which returns the last ID generated on the particular scope). Scope can be a stored procedure, trigger, function, or batch.

The only casewhere you still might have to use `@@IDENTITY` instead of `SCOPE_IDENTITY()` is where your scope is a batch or stored procedure, and you wish to retrieve the last generated value after the stored procedure or batch is done executing. In such a case, `SCOPE_IDENTITY()` will return a null.

Writing This Application in Oracle

Exercise 9.4 has been demonstrated using SQL Server 2005. It's notable, however, that the same ADO.NET concepts apply to any other database. The idea is to get the key values back and emulate the same behavior using database-specific features.

One key difference with Access was mentioned in which you would have to use the `RowUpdated` event to fill in key values. Similarly in Oracle, there are a few differences. You don't quite have to resort to the `RowUpdated` event for `OracleDataAdapter`, but since you don't have identity type columns in Oracle, you would need to use an Oracle-specific feature called sequences to generate and query identity values. An Oracle sequence is simply a database object that gives you a convenient way to generate sequential numbers for use in situations such as primary keys to a table. They aren't always strictly sequential, but they are unique (unless you reset a sequence).

A sequence can be created using the following PL/SQL statement:

```
Create Sequence seq_animals start with 1 increment by 1
```

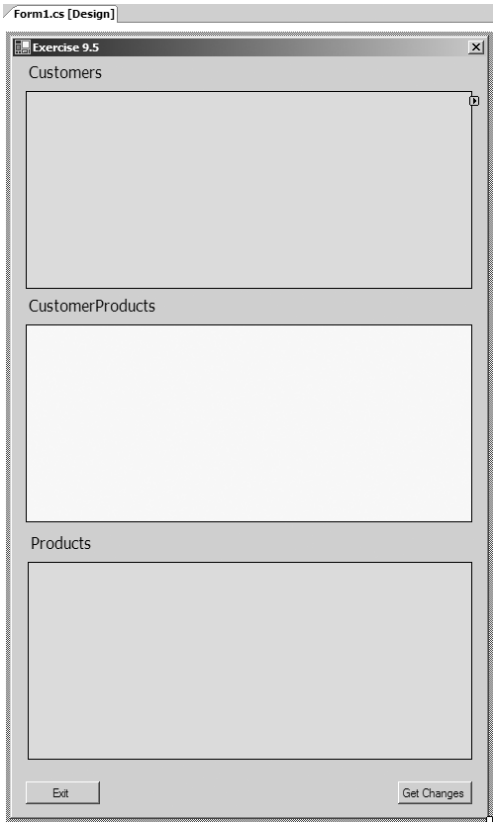


Figure 9-19.The form in Design view

4. In the constructor of the form, set up, load, and data bind the DataSet from an XML file sitting on the disk. This XML file can be found in the associated code download and it is the same XML file used in Chapter 8. The code for the constructor is shown in Listings 9-28 and 9-29.

Listing 9-28. Loading and Data Binding the DataSet in the Constructor in C#

```
public Form1()
{
    InitializeComponent();

    myDataSet = new CustProd();
    dgCustomers.DataSource = myDataSet.Customers;
    dgCustomerProducts.DataSource = myDataSet.CustomerProducts;
    dgProducts.DataSource = myDataSet.Products;

    myDataSet.ReadXml("Data.xml");
    myDataSet.AcceptChanges();
```

Listing 9-29. Loading and Data Binding the DataSet in the Constructor in Visual Basic .NET

```
Private Sub Form1_Load( _
   ByVal sender As System.Object,  _
   ByVal e As System.EventArgs) Handles MyBase.Load
    myDataSet = New CustProd()
    dgCustomers.DataSource = myDataSet.Customers
    dgCustomerProducts.DataSource = myDataSet.CustomerProducts
    dgProducts.DataSource = myDataSet.Products

    myDataSet.ReadXml("Data.xml")
    myDataSet.AcceptChanges()
End Sub
```

5. In the event handler for `btnClose`, add the following code to ensure the form is closed when the user clicks the Exit button.

C#

```
private void btnClose_Click(object sender, EventArgs e)
{
    this.Close();
}
```

VB.NET

```
Private Sub btnClose_Click( _
   ByVal sender As System.Object, ByVal e As System.EventArgs)( _
    Handles btnClose.Click
    Me.Close()
End Sub
```

6. Finally, in the `btnGetChanges` `OnClick` event handler, add the following code:

C#

```
private void btnGetChanges_Click(object sender, EventArgs e)
{
    CustProd changedDS = (CustProd)myDataSet.GetChanges();
    dgCustomers.DataSource = changedDS.Customers;
    dgCustomerProducts.DataSource = changedDS.CustomerProducts;
    dgProducts.DataSource = changedDS.Products;
}
```

VB.NET

```
Private Sub btnGetChanges_Click( _
```

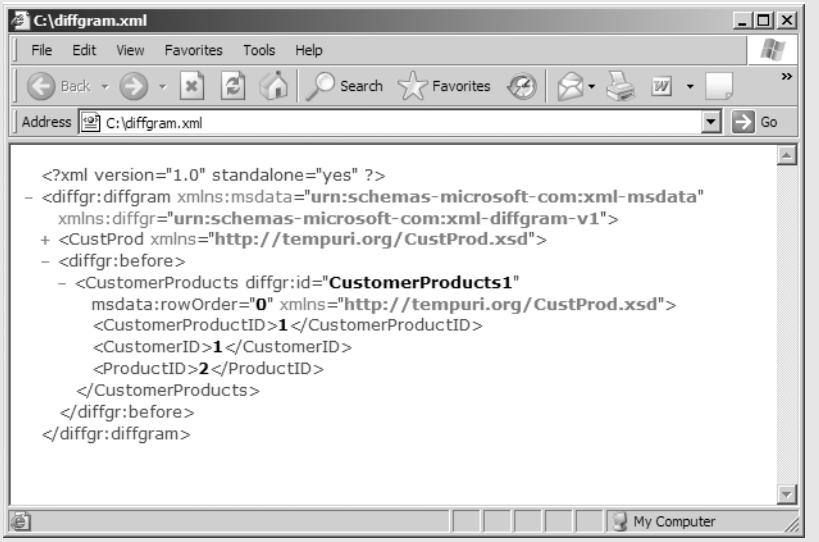


Figure 9-21.Viewing the changes as an XML diffgram

Now that you have the changes extracted out as a mini DataSet with the same structure as the original DataSet, you can use a second method called Merge to integrate those changes with an existing DataSet elsewhere. The use of Merge is rather simple. It looks something like this:

C#

```
Table1.Merge(Table2) ;
```

VB.NET

```
Table1.Merge(Table2)
```

A DataSet also has a Merge method on it, so this discussion can be applied to both a DataSet and a DataTable.

Now I wish I could tell you Merge was as simple as the previous single line of code; however, because the Merge method could be presented with a number of unique cases, it has to take decisions on all of those cases. This is best explained by presenting a few sample cases. The one common theme to remember is that whenever it comes to making a decision, Table1 will override Table2. I know this doesnÕt make much sense yet, but this will become clear in the following cases.

The code for this exercise can be downloaded in Exercise 9.6 in the associated code download. Since you have seen examples of DataSets and DataTables being created and filled many times over now, I will skip creating the example step by step and present only the pertinent parts. You can, however, see the entire code in the associated code download.

In this exercise, as usual, the strongly typed DataSet provides us with a convenient aid in creating the table structures needed to demonstrate various possible cases. This exercise con-

wn in Figure 9-22.

C#

```
VariousTables.Table1DataTable table1 = new VariousTables.Table1DataTable();
table1.LoadDataRow(new object[] { "1", "One" }, true);
table1.LoadDataRow(new object[] { "2", "Two" }, true);

VariousTables.Table2DataTable table2 = new VariousTables.Table2DataTable();
table2.LoadDataRow(new object[] { "2", "Monkey" }, true);
table2.LoadDataRow(new object[] { "3", "Donkey" }, true);

dgView1.DataSource = table1;
dgView2.DataSource = table2;
```

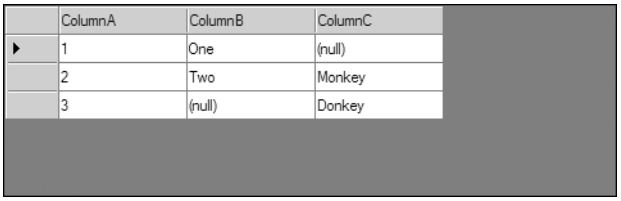
VB.NET

```
Dim table1 As VariousTables.Table1DataTable =  _
   New VariousTables.Table1DataTable()
table1.LoadDataRow(New Object() {"1", "One"}, True)
table1.LoadDataRow(New Object() {"2", "Two"}, True)

Dim table2 As VariousTables.Table2DataTable = _
    New VariousTables.Table2DataTable()
table2.LoadDataRow(New Object() {"2", "Monkey"}, True)
table2.LoadDataRow(New Object() {"3", "Donkey"}, True)

dgView1.DataSource = table1
dgView2.DataSource = table2
```

When you run the application, the data in the two tables is as shown in Figure 9-30.

Figure 9-30.Original data before Merge

e merged as shown in Figure 9-31.



	ColumnA	ColumnB	ColumnC
▶	1	One	(null)
	2	Two	Monkey
	3	(null)	Donkey

Figure 9-31.Data after Merge

So, with a primary key defined, and a different column, the Merge method will end up combining the common column and substituting null wherever it cannot find the appropriate values. But wherever there is a conflict, Table2Õs row is thrown out the window and Table1Õs row is kept. Therefore, if there were a third column in either table, which wasnÕt the primary key, and a differing value between Table1 and Table2 existed, you would see Table1 overriding Table2.

Merge Case 5: Absolutely Different Table Structures

In this case, the two table structures have nothing in common. All the columns differ. The code for loading such DataTables[2] can be seen here:

C#

```
DataTable table1 = new DataTable();
table1.Columns.Add(new DataColumn("ColumnA"));
table1.Columns.Add(new DataColumn("ColumnB"));
table1.LoadDataRow(new object[] { "1", "One" }, true);
table1.LoadDataRow(new object[] { "2", "Two" }, true);

VariousTables.Table3DataTable table2 = new VariousTables.Table3DataTable();
table2.LoadDataRow(new object[] { "3", "Monkey" }, true);
table2.LoadDataRow(new object[] { "4", "Donkey" }, true);

dgView1.DataSource = table1;
dgView2.DataSource = table2;
```

VB.NET

```
Dim table1 As DataTable = New DataTable()
table1.Columns.Add(New DataColumn("ColumnA"))
table1.Columns.Add(New DataColumn("ColumnB"))
table1.LoadDataRow(New Object() {"1", "One"}, True)
table1.LoadDataRow(New Object() {"2", "Two"}, True)
```

2. Why is Table1 not a strongly typed DataTable here? Because the final merge results do not satisfy the unique constraint. As you can see in Figure 9-33, two rows have ColumnA as null, which has a Unique-

- Merge(DataTable, Boolean) : Merges a DataTable into the calling DataTable, preserving changes according to the Boolean argument. A value of True indicates that any changes to the calling DataTable should be maintained. A value of False indicates that such changes should be discarded.
- Merge(DataTable, Boolean, MissingSchemaAction) : Merges a DataTable into the calling DataTable, preserving changes according to the Boolean argument and handling an incompatible schema according to the MissingSchemaAction argument. The MissingSchemaAction enumerator is discussed in just a moment.

Merging DataSets

The DataSet class also exposes a Merge() method that's used to merge one DataSet into another, a DataTable into a DataSet, or an array of DataRows into a DataSet. This type of action is useful when you have data for the same purpose coming from two separate sources—say, inventory data coming from multiple remote locations. Each location can pass a DataSet object to a centralized application where the DataSets can be merged. The critical difference between DataSet merging and DataTable merging is that DataSet merging takes into consideration the defined relations, as well as columns and data in various DataTables.

The DataSet class has the following overloaded Merge() methods:

- Merge(DataRow()) Merges an array of DataRow objects into the calling DataSet.
- Merge(DataSet): Merges a DataSet into the calling DataSet.
- Merge(DataTable): Merges a DataTable into the calling DataSet.
- Merge(DataSet, Boolean) : Merges a DataSet into the calling DataSet, preserving changes according to the Boolean argument. A value of True indicates that any changes to the calling DataSet should be maintained. A value of False indicates that such changes should be discarded.
- Merge(DataRow(), Boolean, MissingSchemaAction) : Merges an array of DataRow objects into the calling DataSet, preserving changes to the DataSet according to the Boolean argument, and handling an incompatible schema according to the MissingSchemaAction argument.
- Merge(DataSet, Boolean, MissingSchemaAction) : Merges a DataSet into the calling DataSet, treating the other two arguments as mentioned previously.
- Merge(DataTable, Boolean, MissingSchemaAction) : Merges a DataTable into the calling DataSet, treating the other two arguments as mentioned previously.

The MissingSchemaAction argument is an enumerator that specifies how to handle the merge operation if the object being merged has a different schema from the calling DataSet. This scenario would occur if, say, a new DataColumn were added to the schema in the merging DataSet. We'll have a closer look at this possibility in a moment.

Merging Two DataSets/DataTables with Different Schemas

As described previously, you can merge two DataSets with different schemas and specify how

```
{
    DataSet dsUsers = new DataSet("Users");

    try
    {
        // Define a  connection object
        OleDbConnection dbConn = new OleDbConnection(
            "Provider=Microsoft.Jet.OLEDB.4.0;" +
            "Password=;User ID=Admin;Data Source=db.mdb");

        // Create a  data adapter to retrieve records from DB
        OleDbDataAdapter daUsers =
            new OleDbDataAdapter("SELECT ID,fn,ln,cty,st" +
            " FROM tabUsers", dbConn);

        // Define each column to map
        DataColumnMapping dcmUserID =
            new DataColumnMapping("ID", "UserID");
        DataColumnMapping dcmFirstName =
            new DataColumnMapping("fn", "FirstName");
        DataColumnMapping dcmLastName =
            new DataColumnMapping("ln", "LastName");
        DataColumnMapping dcmCity =
            new DataColumnMapping("cty", "City");
        DataColumnMapping dcmState =
            new DataColumnMapping("st", "State");

        // Define the table containing the mapped columns
        DataTableMapping dtmUsers = new DataTableMapping("Table", "User");
        dtmUsers.ColumnMappings.Add(dcmUserID);
        dtmUsers.ColumnMappings.Add(dcmFirstName);
        dtmUsers.ColumnMappings.Add(dcmLastName);
        dtmUsers.ColumnMappings.Add(dcmCity);
        dtmUsers.ColumnMappings.Add(dcmState);

        // Activate the mapping mechanism
        daUsers.TableMappings.Add(dtmUsers);

        // Fill the dataset
        daUsers.Fill(dsUsers);

        // Declare a  command builder to create SQL instructions
        // to create and update records.
        OleDbCommandBuilder cb = new OleDbCommandBuilder(daUsers);

        // Insert a  new record in the DataSet
        DataRow r = dsUsers.Tables["User"].NewRow();
```

```
            r["LastName"] = "Robinson";
            r["City"] = "Houston";
            r["State"] = "Texas";
            dsUsers.Tables["User"].Rows.Add(r);

            // Insert the record in the database
            daUsers.Update(dsUsers.GetChanges());

            // Align in-memory data with the data source ones
            dsUsers.AcceptChanges();

            // Print successfully message
            Console.WriteLine("A new record has been"
                + " added to the database.");
        }
        catch (Exception ex)
        {
            // Reject DataSet changes
            dsUsers.RejectChanges();

            // An error occurred. Show the error message
            Console.WriteLine(ex.Message);
        }
    }
  }
}
```

Listing 9-31. Example of Adding a Record Using Visual Basic .NET

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

Module Module1
    Sub Main()
        Dim dsUsers As New DataSet("Users")

        Try
            ' Define a  connection object
            Dim dbConn As New _
      OleDbConnection("Provider=Microsoft.Jet.OLEDB.4.0;" & _
      "Password=;User ID=Admin;Data Source=db.mdb")

            ' Create a  data adapter to retrieve records from DB
            Dim daUsers As New OleDbDataAdapter("SELECT ID,fn,ln,cty,st" & _
                " FROM tabUsers", dbConn)

            ' Define each column to map
            Dim dcmUserID As New DataColumnMapping("ID", "UserID")
```

```
        Dim dcmLastName As New DataColumnMapping("ln", "LastName")
        Dim dcmCity As New DataColumnMapping("cty", "City")
        Dim dcmState As New DataColumnMapping("st", "State")

        ' Define the table containing the mapped columns
        Dim dtmUsers As New DataTableMapping("Table", "User")
        dtmUsers.ColumnMappings.Add(dcmUserID)
        dtmUsers.ColumnMappings.Add(dcmFirstName)
        dtmUsers.ColumnMappings.Add(dcmLastName)
        dtmUsers.ColumnMappings.Add(dcmCity)
        dtmUsers.ColumnMappings.Add(dcmState)

        ' Activate the mapping mechanism
        daUsers.TableMappings.Add(dtmUsers)

        ' Fill the dataset
        daUsers.Fill(dsUsers)

        ' Declare a  command builder to create SQL instructions
        ' to create and update records.
        Dim cb As New OleDbCommandBuilder(daUsers)

        ' Insert a  new record in the DataSet
        Dim r As DataRow = dsUsers.Tables("User").NewRow()
        r("FirstName") = "Eddie"
        r("LastName") = "Robinson"
        r("City") = "Houston"
        r("State") = "Texas"
        dsUsers.Tables("User").Rows.Add(r)

        ' Insert the record in the database
        daUsers.Update(dsUsers.GetChanges())

        ' Align in-memory data with the data source ones
        dsUsers.AcceptChanges()

        ' Print successfully message
        Console.WriteLine("A new record has been" & _
           " added to the database.")
    Catch ex As Exception
        ' Reject DataSet changes
        dsUsers.RejectChanges()

        ' An error occurred. Show the error message
        Console.WriteLine(ex.Message)
    End Try
End Sub
```

In the bolded snippet of code in Listings 9-30 and 9-31, a new row has been created and filled using the mapped column names. In Listings 9-32 and 9-33, a column is updated using a mapping mechanism.

Listing 9-32. Example of Updating a Record Using ColumnMappings Using C#

```
#region Using directives

using System;
using System.Collections.Generic;
using System.Text;
using System.Data.OleDb;
using System.Data.Common;
using System.Data;

#endregion

namespace Exercise_9_8
{
    class Program
    {
        static void Main(string[] args)
        {
            DataSet dsUsers = new DataSet("Users");
            try
            {
                OleDbConnection dbConn =
                    new OleDbConnection("Provider=Microsoft.Jet.OLEDB.4.0;" +
                    "Password=;User ID=Admin;Data Source=db.mdb");

                // Create a  data adapter to retrieve records from db
                OleDbDataAdapter daUsers =
                    new OleDbDataAdapter("SELECT ID,fn,ln,cty,st" +
                    " FROM tabUsers", dbConn);

                // Define each column to map
                DataColumnMapping dcmUserID =
                    new DataColumnMapping("ID", "UserID");
                DataColumnMapping dcmFirstName =
                    new DataColumnMapping("fn", "FirstName");
                DataColumnMapping dcmLastName =
                    new DataColumnMapping("ln", "LastName");
                DataColumnMapping dcmCity =
                    new DataColumnMapping("cty", "City");
                DataColumnMapping dcmState =
                    new DataColumnMapping("st", "State");
                // Define the table containing the mapped columns
```

```
            new DataTableMapping("Table", "User");
        dtmUsers.ColumnMappings.Add(dcmUserID);
        dtmUsers.ColumnMappings.Add(dcmFirstName);
        dtmUsers.ColumnMappings.Add(dcmLastName);
        dtmUsers.ColumnMappings.Add(dcmCity);
        dtmUsers.ColumnMappings.Add(dcmState);

        // Activate the mapping mechanism
        daUsers.TableMappings.Add(dtmUsers);

        // Fill the dataset
        daUsers.Fill(dsUsers);

        DataColumn[] dcaKey = { dsUsers.Tables["User"].Columns["UserID"] };
        dsUsers.Tables["User"].PrimaryKey = dcaKey;

        // Declare a  command builder to create SQL instructions
        // to create and update records.
        OleDbCommandBuilder cb = new OleDbCommandBuilder(daUsers);

        // Update an existing record in the DataSet
        DataRow r = dsUsers.Tables["User"].Rows.Find(8);

        if (r != null)
        {
            r["FirstName"] = "Venus";
            r["LastName"] = "Williams";
            r["City"] = "Houston";
            r["State"] = "Texas";

            // Update the record in the database
            daUsers.Update(dsUsers.GetChanges());

            // Align in-memory data with the data source ones
            dsUsers.AcceptChanges();

            // Print success message
            Console.WriteLine("The record has been updated " +
                "successfully.");
        }
        else
        {
            Console.WriteLine("No record found...");
        }
    }
    catch (System.Exception ex)
    {
```

CHAPTER 10

Updating Data: Advanced Scenarios

In the last chapter, you were introduced to the concept of updating in ADO.NET. This chapter takes the same discussion a step further by introducing more real-world scenarios and other real-world problems.

ADO.NET is different from previous data access architectures in the sense that it allows you to retrieve a purely disconnected cache of records in an object representation of a database called a `DataSet`. The `DataSet` resembles the database, but isn't quite a database. It allows you to search, sort, and filter through the records it contains, works with various relations, and leverages the power of both an object representation and XML representation to allow you to easily update its contents.

But a `DataSet` isn't a database, and it shouldn't be abused as one either; this statement will become very clear in the contents of this chapter. In a fully connected environment, you can easily update, insert, or delete various relational data; however, when you are working with disconnected data, the mundane tasks of fetching the latest generated keys and managing concurrency issues on the same hierarchical relational data can get quite complex.

Because a `DataSet` is completely disconnected, all the changes done to it while it's disconnected are then saved back to the database, typically when the user clicks the Save button, or when some other part of the application architecture does the equivalent of a save. When this happens, the operation ultimately calls a data adapter's `Update` method or something similar. In most cases, data adapters support updating single tables or row arrays; if you need anything more complex, it's generally up to you to write the commands for that.

Now you could modify the `DataSet` as much as you wish, the only thing that would have been changed so far would be the in-memory disconnected cache. Until you click Save (by virtue of which you will start a process that will insert, modify, or delete various changes to the database), the database is untouched. And even when you do click Save, because of the fact that you had queried a snapshot of the database as a `DataSet` some time in history, that snapshot might be out of date when you do a save. There could be a number of other users who might be working on the same data, and you need to be mindful of the changes they might have done between your querying for the data and your attempting to save the changes.

Between querying and saving, a number of things could have happened that you might need to take care of at save time, including the following:

anyone who wishes to update that row, while you still haven't updated it, simply waits, times out, or is given an error. That sounds like a great idea, but it has some serious drawbacks.[1]

First, by locking rows you are creating a serious contention with the one central piece of architecture in your system—the database. This goes against ADO.NET's general philosophy, which emphasizes connecting as late as possible and disconnecting as soon as you can. Now, not only does the database have to do the "row protection" (or page or table protection) for you, but it also has to constantly listen to your request if you decide to unlock that row.

Second, this approach goes against another ADO.NET philosophy, which encourages connection pooling by reducing the actual time you need and keeping the connection open and reserved for yourself. ADO.NET will allow you to do this should you choose this as your solution, but it's not a recommended approach. Rightfully so, ADO.NET encourages you to use a disconnected architecture, which just doesn't go hand in hand with resource locking at the database level. It's true, however, that you could execute a database-specific command using the `DbCommand` object, like the one shown here (a SQL Server command):

```
Select * from Customers HOLDLOCK where CustomerID = 1093
```

By using such a command in a transaction, you'd essentially end up locking the rows and prevent any other user from updating those rows until you commit or abort the transaction.

This creates some interesting problems. The end user could have locked the row by one of his actions and then, let's say, that he left for lunch and left the screen open leaving that row locked. Now while he is out to lunch, nobody else can update that row. Not only that, but you have the serious contention issue of a lock causing another lock. If it gets bad enough, you might end up with deadlocks in the database, which will require the intervention of a database administrator to clean up and kill those transactions for everyone else in a concurrent system—clearly something you'd rather not deal with in a highly concurrent application. So now you have to worry about implementing a timeout mechanism on the user interface, but what if the application crashes?

Let's say the application didn't crash, but given that application logic can get complex, what if you had Resource A waiting for Resource B, Resource B waiting for Resource C, and Resource C waiting for Resource A? You'd have a deadlock, and even though some databases can detect such a situation, it's still quite expensive for the one central resource involved.

Since ADO.NET lets you execute commands directly at the database level (using `DbCommand`), another way of implementing a solution slightly better than transaction-based row locking is to create a scrollable, updateable server-side cursor. This can be implemented in ADO.NET by simply wrapping the `CREATE CURSOR` command inside a `DbCommand`. The server-side cursor can then be positioned to the particular row you wish to work with—hence, apprising your application of the latest changes at any given point in time. This does not, however, solve the problem of having to create and keep an open connection for the life of the cursor, but if you must do pessimistic locking this might be a better approach.

Yet another method could be to build in a check-in/check-out functionality of rows in the application. Thus, using some clever application logic, a checked-out row is left alone by other

---

1. As you will see shortly, a better way to implement locking, without explicitly using pessimistic concurrency, is to implement locking in the logic that surrounds the database. This way, a user is informed in advance that the 10 minutes he is about to spend updating a particular screen of information shouldn't be done now because another user is busy working with the data at this very moment. Also, in this approach, you don't keep a database connection open or overload the central policeman more than

Table 10-3. Final Updated Row Per Last-In Wins

AnimalID	AnimalType	AnimalWeight
1	Mutt	50 lbs

Typically, this is done by using a sequence of SQL queries that looks like this:

```
-- Erick's Select query
Select AnimalID, AnimalType, AnimalWeight from Animals
-- Frans'ss Select query
Select AnimalID, AnimalType, AnimalWeight from Animals
-- Erick's update query
Update Animals Set AnimalType = 'Dog', AnimalWeight = '50 lbs' where AnimalID = 1
-- Frans's update query
Update Animals Set AnimalType = 'Mutt', AnimalWeight = '50 lbs' where AnimalID = 1
```

Since the row lookup was done using only the primary key, which is probably indexed, and only the columns changed were updated, this performs fairly well because Frans never checked for any concurrency/conflicts and saved the time to run those queries.

Check All Columns Before an Update

The problem with the previously mentioned approach is that Erick thinks he's still working with a Dog. It's not until Erick fully refreshes his user interface that he'll see that Frans changed the row in the database to Mutt. And when he does do a refresh, by then he might have updated another pets table using the Dog animal row just because he didn't know the latest name for the animal in the database. (Prevent such accidents: using the right database design, you would obviously not copy the name of the animal, but only the key, i.e., `AnimalID`).

Perhaps a better approach in this scenario would have been for Frans to check for Erick's or any other user's updates between the time he fetched the data and the time he wished to update his changes back to the database. So, instead of issuing an `UPDATE` command that contains only the primary key in the `WHERE` clause, the `UPDATE` command's `WHERE` clause could instead contain a check for all values queried in the first place.

In other words, the sequence of commands the database will work with would look like this:

```
-- Erick's Select query
Select AnimalID, AnimalType, AnimalWeight from Animals
-- Frans's Select query
Select AnimalID, AnimalType, AnimalWeight from Animals

-- Erick's update query
Update Animals
   Set AnimalType = 'Dog', AnimalWeight = '50 lbs'
where
   AnimalID = 1 and AnimalWeight = '3 lbs' and AnimalType = 'Puppy'
-- Frans's update query
Update Animals
   Set AnimalType = 'Mutt', AnimalWeight = '50 lbs'
where
```

These queries take a slightly safer approach; instead of issuing only the primary key as part of the `WHERE` clause to identify the one row to be changed, a `WHERE` clause is constructed out of all the values in the disconnected data cache that were queried in the first place. By doing so, at the end of Erick's `UPDATE` query, the data in the table looks like as shown in Table 10-4. Now at this time, when Frans comes and queries for a row based on his original data, he will simply not update any rows. This can be easily caught using the return value of the `ExecuteNonQuery` method on the `DbCommand` object which will tell you that no rows changed, when you had expected one row to be changed.

Table 10-4. Updated Row

AnimalID	AnimalType	AnimalWeight
1	Dog	50 lbs

This is a one-size-fits-all approach and is what the previous data access architectures, such as `Recordsets` in ADO classic, used.

Obviously, you don't corrupt data using this approach. In this approach, once a conflict has been detected because `ExecuteNonQuery` returned 0 modified rows (whereas the expected number was 1), Frans will know that a conflict has occurred. At this point the application should inform Frans of the conflict by refilling his data cache and asking him to save his changes again if he wishes to.

This approach is used by classic data access platforms such as the `Recordset` in ADO classic and `CommandBuilder` in .NET Framework 1.1. As a matter of fact, as you saw in Chapter 9, to ensure backward compatibility, the `CommandBuilder` object still uses this approach by default.

But, what if one of the columns involved in this comparison was a blob? It would take a lot to compare a blob, which could even be in megabytes of data. Thankfully, the query generation engine used in `SqlCommandBuilder` or `OracleCommandBuilder` is smart enough to exclude the blob data types, which has a negative side that blobs will not be checked for this kind of concurrency check done by various common command builder objects. The query generation engine still, however, must query the database for the structure of the table and various data types involved. That query could not only be expensive, but also it might not work if you do not have the proper access rights. Even if you were able to run this query, and generate a query that contains every single column in the `WHERE` clause, simply executing such a generated query might be quite a bit more expensive than other approaches.

The biggest advantage of this approach, however, is that setting up this kind of optimistic concurrency model requires very little effort (for a single table at least). If your performance needs aren't that severe, given the lesser code maintenance you would have to do in this solution, maybe this would be the better approach for you.

Check Only Modified Columns and Primary Keys Before an Update

Checking all columns has its downsides: it will not work with blobs, it's too expensive to generate such a query, and it takes too long to execute. Plus, if you decide that the programmatic ease of generating such a query is not worth the performance you might need out of the system, you may choose a midway approach.

The midway approach involves checking only the modified columns plus the primary key. So let's say that Frans and Erick have already queried the table as per Table 10-1. Now rans agrees that he doesn't wish to

rename Puppy to Mutt; rather, he wishes to update the Puppy to Dog. But let's say Frans has a heavier dog; his dog is 60 lbs not 50 lbs. So the sequence of queries would look somewhat like this:

```
-- Erick's Select query
Select AnimalID, AnimalType, AnimalWeight from Animals
-- Frans's Select query
Select AnimalID, AnimalType, AnimalWeight from Animals

-- Erick's update query
Update Animals
   Set AnimalType = 'Dog', AnimalWeight = '50 lbs'
where
   AnimalID = 1 and AnimalWeight = '3 lbs' and AnimalType = 'Puppy'
-- Frans's update query, this will now fail.
Update Animals
   Set AnimalWeight = '60 lbs'
where
   AnimalID = 1 and AnimalWeight = '3 lbs'
```

This query sequence will succeed for Erick, but it will fail for Frans. At this point, however, Frans will have the ability to refresh the data back from the database. He will intelligently decide that, indeed, Erick has already done half the work by updating Puppy to Dog, but Frans doesn't quite agree that the dog is only 50 lbs. So Frans will reexecute a set of queries as shown here, to update only the DogÕs weight

```
-- Frans's Select query
Select AnimalID, AnimalType, AnimalWeight from Animals

-- Frans's update query
Update Animals
   Set AnimalWeight = '60 lbs'
where
   AnimalID = 1 and AnimalWeight = '50 lbs'
```

This query will now succeed. So in this approach, Frans was prevented from making changes to a row that had changed. He was rightfully informed that another user had modified the row. Given such a meaningful message and the latest data on his screen, he could now easily update the `Animal` row and change only its weight.

Obviously, this approach has the plus of not having a very complex `WHERE` clause. Also, such a `WHERE` clause would generally be specified by you in advance, instead of being queried from the database. This would generally lead to better performance in comparison with a query that compares every single column in the `WHERE` clause, though query plans could change much too frequently to be effectively cached.

There are still a few problems with this approach, though.

Checking for Timestamps

In our pursuit to find the best concurrency management scheme, the last approach discussed

y key and the modified columns.

- A concert ticket booking website can implement functionality to lock your seats for 2 minutes. This way, the seats are checked out to you for 2 minutes, so no explicit clean-up logic is needed. If another user requests the same seats after 2 minutes, you lose the seats; that is, your records are implicitly checked in due to time lapse.
- A payment application to an insurance account would need you to first open a batch, enter checks, and then close a batch before you can apply the batch. Of course, the user would also request the identity keys in advance for the checks they are about to enter.
- An e-commerce site may decide that checking inventory every time an order is placed for a popular item is too expensive to do at every order placement. Instead, it may choose to inform the last five or ten users that their orders cannot be fulfilled or, when the inventory drops below a certain number, it may show a message informing that their orders may be delayed.
- When editing a complex screen full of information about a patient, you would first need to take it out of read-only mode (check out), then do your changes, and finally save (check in your changes).

The common underlying theme is to avoid situations that lock too many resources for too long at a given point in time and implement as many optimistic checks to keep the system more maintainable. But in a real-world application, that is never completely unavoidable. In those instances, you need to decide on the right combination of approaches that have been mentioned here.

Implementing Concurrency: Practical Concerns

Per the previous academic discussion, you could probably come up with a concurrency mechanism that suits you best. In most cases, you would probably want to go with optimistic concurrency. As a matter of fact, even where you need strict row locking, you could use alternate mechanisms such as check-in/check-out logic and not create actual row locks in the database.

When you have selected an appropriate concurrency management strategy, there are a few things you need to consider when implementing concurrency in your data layer.

Null Values

.NET 1.x doesn't allow nullable value types like databases or languages such as C++ may allow you to have. .NET 2.0 did introduce nullable value types, but they are not quite the same as the nullable value types that may exist in the context of databases or C++; thus, those don't go all the way in helping you deal with the null mismatches between databases and .NET. It's for this purpose that .NET provides you with various facilities to work with nullable columns in the database.

A database null is represented by the `System.DBNull` structure in .NET. Usually, this can be easily retrieved using the `System.Convert.DBNull` constant to assign null values to a column. Alternatively, on reads you can use the `DbDataReader.IsNull` method to check if a particular column is null or not. As a better performing alternative, you could also compare the result of `GetValue` with `DBNull.Value` inside a data reader loop.

Strongly typed `DataSets` make it even easier for you by allowing you to set a particular col-

```
(1 row(s) affected)
(1 row(s) affected)
```

This output is correct since the first row is due to the insert you did and the second is probably an insert into the log table—thanks to the trigger. The problem is this: when you run `ExecuteNonQuery` on such a command, `ExecuteNonQuery` will tell you that it ended up modifying two rows. Since concurrency management schemes depend on checking how many rows did get modified, such an approach would create a problem.

The work-around for this is to use SQL Server's `NOCOUNT` setting. By setting `NOCOUNT ON` and `NOCOUNT OFF` at appropriate places inside the trigger, you can ensure that you do not confuse your application logic by reporting changed rows that might not be important from a concurrency check point of view.

Similarly, if you have a batched SQL statement, or a stored procedure which has multiple statements modifying rows, then you need to execute `SELECT @@ROWCOUNT` to find the exact number of rows modified at each specific given statement.

Multiple Rows Being Updated

This is a tricky one. You have a `DataTable` with three rows changed. One is inserted, another is modified, and the third is deleted. Now, assuming that you have specified an `InsertCommand`, `UpdateCommand`, and `DeleteCommand` on the `DataAdapter`, the appropriate commands will get called one by one based upon the various `RowStates`.

Say the `InsertCommand` succeeds, but the modified (`UpdateCommand`) fails; should you then

- Go ahead and execute the `DeleteCommand`?
- Stop execution right there?
- Roll back the `InsertCommand`?

Well, technically per your application logic, you should be able to choose any of these three. The exact situation might demand that you handle each scenario per its specific requirements. It's for this reason that you have the `ContinueUpdateOnError` property on a `DataAdapter`. The behavior is actually quite simple: By leaving `ContinueUpdateOnError` on `false` (the default value), if an error occurs during the execution of a command for a particular changed row, you should get a `DbConcurrencyException`. At that point, the `DataAdapter` will stop executing any further commands and return.

If you set the `ContinueUpdateOnError` property to `true`, the `DataAdapter` would continue executing commands. Its default value is `false` because by setting this property to `false`, you complete or fail an entire update operation and report success only on a complete success. That is, if one row fails the whole operation fails; thus, making the update atomic in nature if indeed you did wrap it in a transaction.

However, it's important that, in this case, ADO.NET should somehow inform us of the rows that have errors in them. This can be easily found out using the `DataRow.HasErrors` property to verify if a particular row has errors in it or not. If it does have errors, then you can get the exact error description using the `DataRow.RowError` property. Similarly, the `DataSet` and `DataTable` also have the `HasErrors` property, which lets you easily identify any errors at a global level. If a `DataRow` does have an error when it's databound to a `DataGridView` control, the particular row shows an exclamation mark indicating an error, and the tool tip indicates the error description

As you can see in the code download, to make our lives easier the exercise uses a strongly typed `DataSet`. The strongly typed `DataSet` contains three tables as shown in Figure 10-2, which mimic the database table structure.

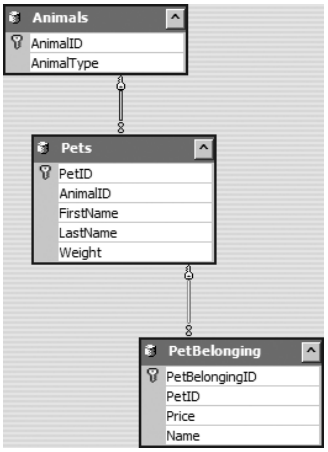


Figure 10-2. The strongly typed DataSet

Because the strongly typed `DataSet` is intended to store a disconnected cache, some of the concepts learned in this chapter are used to set up primary keys, seed, and increment values in the three tables. Thus, in Exercise 10.1 you can see `Animals.AnimalID`, `Pets.PetID`, and `PetBelonging.PetBelongingID` have been set up as primary keys with `AutoIncrementSeed = 0`, and `AutoIncrementStep = -1` in the strongly typed `DataSet`.

Finally, again to mimic the database, two relationships are set up between the three `DataTables` representing the foreign keys that exist in the database. The first is `FK_Animals_Pets` between `Animals.AnimalID` and `Pets.AnimalID`, and the second is `FK_Pets_PetBelonging` between `Pets.PetID`, and `PetBelonging.PetID`. These relationships have their update rule and delete rule set to cascade by default.

With the `DataSet` set up, the three tables can be easily bound to three `DataGridViews` to generate columns at design time. Also, just to make sure the user doesn't meddle with the autogenerated keys, the primary columns in all three tables, `Animals.AnimalID`, `Pets.PetID`, and `PetBelonging.PetBelongingID`, have been marked as read only.

The final UI in design time looks like as shown in Figure 10-3.

```
   dgAnimals.DataSource = animalsData.Tables["Animals"];
   dgPets.DataSource = animalsData.Tables["Pets"];
   dgPetBelonging.DataSource = animalsData.Tables["PetBelonging"];
}
```

Listing 10-2. Filling the DataSet and Data Binding It to the Right DataGridViews in Visual Basic .NET

```
Private Sub FillData()
   animalsData = New AnimalsDataSet()
   Dim sqlDA As SqlDataAdapter = New SqlDataAdapter( _
      "Select * from Animals; Select * from Pets;" & _
      "Select * from PetBelonging", _
   connectionString)
   sqlDa.TableMappings.Add("Table", "Animals")
   sqlDa.TableMappings.Add("Table1", "Pets")
   sqlDa.TableMappings.Add("Table2", "PetBelonging")
   sqlDa.Fill(animalsData)

   dgAnimals.DataSource = animalsData.Tables("Animals")
   dgPets.DataSource = animalsData.Tables("Pets")
   dgPetBelonging.DataSource = _
   animalsData.Tables("PetBelonging")
End Sub
```

Now if you run the application, you should see the data populated as shown in Figure 10-4.

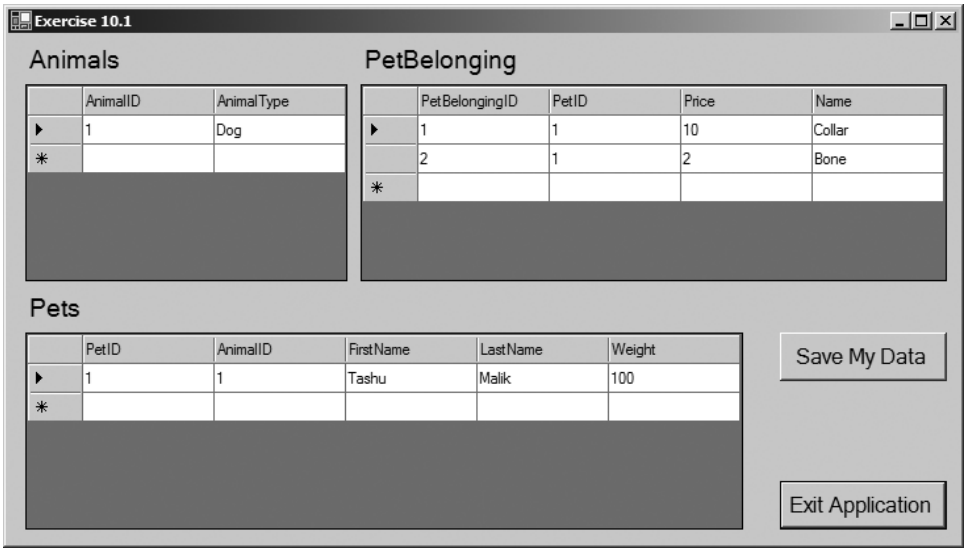


Figure 10-4. The application allowing the user to edit data at runtime

e the fun begins!

```
// This parameter will be retreived from the first command insertPetCommand
param = new SqlParameter("@PetID", SqlDbType.Int, 4, "PetID");
param.Direction = ParameterDirection.Input;
insertPetBelongingCommand.Parameters.Add(param);

insertPetBelongingCommand.Transaction = trans;
#endregion
```

Listing 10-7. Setting Up the insertPetCommand and insertPetBelongingCommand in Visual Basic .NET

```
' Moving down the hierarchy - the Pets Table
'  insertPetCommand
Dim insertPetCommand As SqlCommand = New SqlCommand()
insertPetCommand.Connection = testConnection

insertPetCommand = New SqlCommand("UP_PETSINSERT")
insertPetCommand.CommandType = CommandType.StoredProcedure

param = New SqlParameter("@PetID", SqlDbType.Int, 4, "PetID")
param.Direction = ParameterDirection.Output
insertPetCommand.Parameters.Add(param)

param = New SqlParameter("@FirstName", SqlDbType.VarChar, 50, "FirstName")
param.Direction = ParameterDirection.Input
insertPetCommand.Parameters.Add(param)

param = New SqlParameter("@LastName", SqlDbType.VarChar, 50, "LastName")
param.Direction = ParameterDirection.Input
insertPetCommand.Parameters.Add(param)

param = New SqlParameter("@Weight", SqlDbType.Int, 4, "Weight")
param.Direction = ParameterDirection.Input
insertPetCommand.Parameters.Add(param)

' This parameter will be retreived from the first command insertAnimalCommand
param = New SqlParameter("@AnimalID", SqlDbType.Int, 4, "AnimalID")
param.Direction = ParameterDirection.Input
insertPetCommand.Parameters.Add(param)

insertPetCommand.Transaction = trans

' Finally moving to the end of the hierarchy - PetBelonging
'  insertPetBelongingCommand
Dim insertPetBelongingCommand As SqlCommand = New SqlCommand()
insertPetBelongingCommand.Connection = testConnection
```

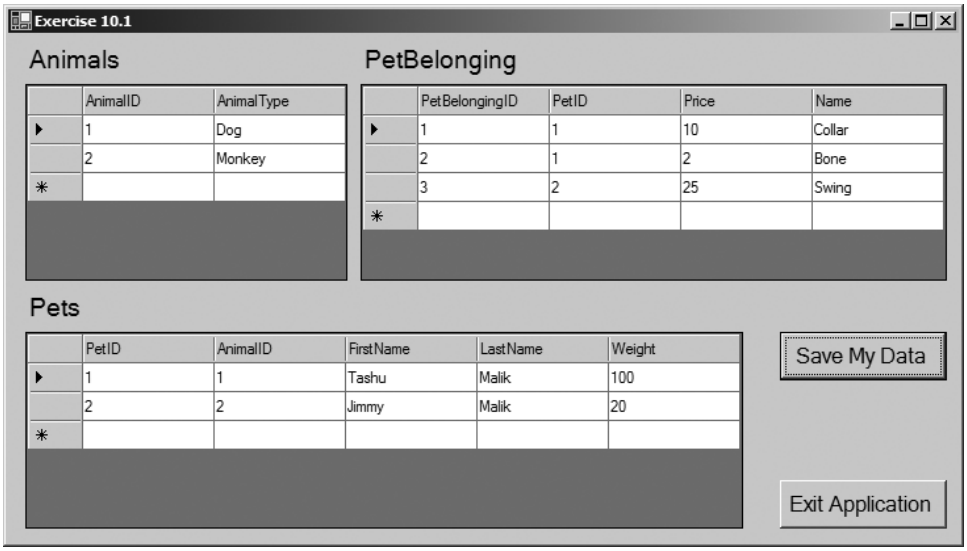


Figure 10-6. Added data, after clicking the Save button. Notice the Key values.

As you can see in Listings 10-9 and 10-10, a special overload of the `DataAdapter`'s `Update` method was used. `DbDataAdapter.Update` supports a few other overloads as shown in Table 10-5.

Table 10-5. DbDataAdapter.Update Overloads

Constant	Description
`DbDataAdapter.Update(DataRow[])`	Calls `INSERT`, `UPDATE`, `DELETE` on various changed rows specified in the data row array.
`DbDataAdapter.Update(DataSet)`	Calls `INSERT`, `UPDATE`, `DELETE` on various changed rows in the first table of the `DataSet`. Note that previous MSDN documentation versions specify that the commands will be called on all rows in the `DataSet`; that is incorrect.
`DbDataAdapter.Update(DataTable)`	Calls `INSERT`, `UPDATE`, `DELETE` on various changed rows in the specified `DataTable`.
`DbDataAdapter.Update(DataRow[]. DataTableMapping)`	Calls `INSERT`, `UPDATE`, `DELETE` on various changed rows in the specified `DataRow` array, while taking care of various mappings.
`DbDataAdapter.Update(DataSet, String)`	Calls `INSERT`, `UPDATE`, `DELETE` on various changed rows in one `DataTable` of a `DataSet` identified by the second parameter.

The overload used is `DbDataAdapter.Update(DataRow[])`. The `DataRow` array is filtered out for only the added rows using the `DataTable.Select` method. The reason this was necessary was because, in hierarchical updates, inserted rows might need a different logic than updated (`DataViewRowState.ModifiedCurrent`) or deleted (`DataViewRowState.Deleted`) rows.

- Do inserts first in a top-down approach.
- Then do updates in a top-down approach.
- Finally, do deletes in a down-top approach.

Note These are not cardinal, unbreakable rules. They just make sense in most cases, though depending on the exact structure of your `DataSet` you might come up with a different approach. ItÕs important to understand how a `DataAdapter` works, and then understand the previous discussion that concludes with these three points, rather than just memorizing them.

The final "Save My Data" code for the given `DataSet` in Exercise 10.1 would look somewhat like Listings 10-11 and 10-12. Obviously, this is just an outline. Generally, you would put such transactional code in either `using` blocks or in try...catch constructs.

Listing 10-11. Insert, Update, Delete Code for a Hierarchical DataSet in C#

```
// First inserts and updates in a top-down approach.
sqlDa = new SqlDataAdapter("Select * from Animals", testConnection);
sqlDa.InsertCommand = insertAnimalCommand;
sqlDa.InsertCommand.Connection = testConnection;
sqlDa.Update(animalsData.Animals.Select("", "", DataViewRowState.Added));

sqlDa = new SqlDataAdapter("Select * from Animals", testConnection);
sqlDa.UpdateCommand = updateAnimalCommand;
sqlDa.UpdateCommand.Connection = testConnection;
sqlDa.Update(animalsData.Animals.Select("", "", DataViewRowState.ModifiedCurrent));

sqlDa = new SqlDataAdapter("Select * from Pets", testConnection);
sqlDa.InsertCommand = insertPetCommand;
sqlDa.InsertCommand.Connection = testConnection;
sqlDa.Update(animalsData.Pets.Select("", "", DataViewRowState.Added));

sqlDa = new SqlDataAdapter("Select * from Pets", testConnection);
sqlDa.UpdateCommand = updatePetCommand;
sqlDa.UpdateCommand.Connection = testConnection;
sqlDa.Update(animalsData.Pets.Select("", "", DataViewRowState.ModifiedCurrent));

sqlDa = new SqlDataAdapter("Select * from PetBelonging", testConnection);
sqlDa.InsertCommand = insertPetBelongingCommand;
sqlDa.InsertCommand.Connection = testConnection;
sqlDa.Update(animalsData.PetBelonging.Select("", "", DataViewRowState.Added));

sqlDa = new SqlDataAdapter("Select * from PetBelonging", testConnection);
sqlDa.UpdateCommand = updatePetBelongingCommand;
```

```
sqlDa.Update(animalsData.PetBelonging.Select("", "",
   DataViewRowState.ModifiedCurrent));

// Finally deletes in a  down-top approach.
sqlDa = new SqlDataAdapter("Select * from PetBelonging", testConnection);
sqlDa.DeleteCommand = deletePetBelongingCommand;
sqlDa.DeleteCommand.Connection = testConnection;
sqlDa.Update(animalsData.PetBelonging.Select("", "", DataViewRowState.Deleted));

sqlDa = new SqlDataAdapter("Select * from Pets", testConnection);
sqlDa.DeleteCommand = deletePetCommand;
sqlDa.DeleteCommand.Connection = testConnection;
sqlDa.Update(animalsData.Pet.Select("", "", DataViewRowState.Deleted));

sqlDa = new SqlDataAdapter("Select * from Animal", testConnection);
sqlDa.DeleteCommand = deleteAnimalCommand;
sqlDa.DeleteCommand.Connection = testConnection;
sqlDa.Update(animalsData.Animal.Select("", "", DataViewRowState.Deleted));
```

Listing 10-12. Insert, Update, Delete Code for a Hierarchical DataSet in Visual Basic .NET

```
' First inserts and updates in a  top-down approach.
sqlDa = New SqlDataAdapter("Select * from Animals", testConnection)
sqlDa.InsertCommand = insertAnimalCommand
sqlDa.InsertCommand.Connection = testConnection
sqlDa.Update(animalsData.Animals.Select("", "", DataViewRowState.Added))

sqlDa = New SqlDataAdapter("Select * from Animals", testConnection)
sqlDa.UpdateCommand = updateAnimalCommand
sqlDa.UpdateCommand.Connection = testConnection
sqlDa.Update(animalsData.Animals.Select("", "", DataViewRowState.ModifiedCurrent))

sqlDa = New SqlDataAdapter("Select * from Pets", testConnection)
sqlDa.InsertCommand = insertPetCommand
sqlDa.InsertCommand.Connection = testConnection
sqlDa.Update(animalsData.Pets.Select("", "", DataViewRowState.Added))

sqlDa = New SqlDataAdapter("Select * from Pets", testConnection)
sqlDa.UpdateCommand = updatePetCommand
sqlDa.UpdateCommand.Connection = testConnection
sqlDa.Update(animalsData.Pets.Select("", "", DataViewRowState.ModifiedCurrent))

sqlDa = New SqlDataAdapter("Select * from PetBelonging", testConnection)
sqlDa.InsertCommand = insertPetBelongingCommand
sqlDa.InsertCommand.Connection = testConnection
sqlDa.Update(animalsData.PetBelonging.Select("", "", DataViewRowState.Added))
```

CHAPTER 11

Transactions

Databases hold information. ADO.NET helps you interact with that information. Information is critical in todayÕs information age, and its sanctity is required to be guarded closely.

In a multiuser scenario, or even within a single-user scenario, as multiple processes, threads, and entities make changes to the database, there is a danger of stepping over someoneelseÕs changes. In addition, in certain situations, even within the same logical process, you may wish to execute a number of statements together.

For example, when you walk to an ATM (automated teller machine), pop in your card, and request some cash to be withdrawn, there are a distinct number of steps that occur in such a transaction. A request is made to check if you have the necessary funds, the funds are disbursed, and the account balance is reduced accordingly.

ItÕs necessary that no other operation be done between checking the funds and disbursing them, because if that werenÕt the case, someone could hack the system while checking the funds once and withdraw a number of times on the same reported balance.

Also it is critical that the account balance be reduced accordingly as cash is dispensed, and vice versa if a deposit is made. For example, not only would it be really bad if you didnÕt get the cash, but also if your account balance was reduced.

Thus, all these steps must happen together as if they were one unified stepÑthat is, they must all commit or all roll back together. The operation that will commit or roll back all of these steps together is commonly referred to as a transaction.

There are a number of systems involved in this transaction: the database, the ATM, and who knows how many computers in the middle.

Since a transaction could lock contentious resources for a finite time and also because it doesnÕt come for free, as is logical to expect, wrapping these distributed steps into one transaction costs system resources, and at times you might decide not to implement a transaction but rather implement checks and boundary conditions around a failed condition. This is, in fact, what most disconnected ATMs would doÑadd sanity checks at the beginning and ending of various distributed commands, and limit the maximum damage that can be done. Thus, if a command sequence fails, you have a recourse action and limited damage.

Therefore, the next time you use an ATM in a shady bar that is not actively connected over a connection to the bankÕs computer, you will have a daily withdrawal limit set on your card (limiting damage), and there will be a camera watching you (recourse action). Also, the bankÕs computers will maintain enough information for a recourse action to correct the data, if need be.

Thus, itÕs important to understand where you can and should leverage transactions to avoid the need for such checks and recourse, and where you should rely on a potential recourse action

where a transaction mishap might occur. In most cases, however, it makes sense to use the rich framework provided to you to wrap up transactions, but as it turns out, even within that realm there are many choices and flavors to pick from.

This chapter delves into the basics of transactions as it applies to ADO.NET and databases, while giving sufficient coverage to distributed transactions using the new System.Transactions namespace in .NET 2.0. YouÕll see the various pros and cons of each approach that will help you make wise choices applicable to your system architecture.

Also discussed in this chapter is a SQL Server 2005Ðspecific featureÑMARS (Multiple Active Resultsets) and its support in ADO.NET. In short, MARS lets you execute multiple statements and maintain their results on the same connection. ItÕs important to note that, in reality, the statements are not executing in parallel, they are only executing on the same connection. However, simultaneously, active result sets are being maintained.

This makes me sit back and scratch my head and wonder, what if those two commands include a select command and an update command? What if there is another insert command in the mix? What happens if there is a transaction in the mix? MARS gets especially interesting when viewed within transactional semantics and deserves sufficient attention in this context, as will be discussed in this chapter.

But first things first, what is a transaction?

What Is a Transaction?

A transaction is a set of operations where either all of the operations must be successful or all of them must fail to ensure consistency and correct behavior within a system. LetÕs look at the traditional example of a transaction.

Suppose you need to transfer $1,000 from account A to account B. This operation involves two steps:

1. $1,000 should be deducted from account A.

2. $1,000 should be added to account B.

Say that you successfully completed step 1, but due to some error step 2 failed. If you do not undo step 1, then the entire operation will be faulty. Transactions help to avoid this. Operations in the same transaction will only make changes to the database if all the steps are successful. So in this example, if step 2 fails, then the changes made by step 1 will not be committed to the database.

Transactions usually follow certain guidelines known as the ACID properties, which ensure that even complex transactions will be self-contained and reliable.

ACID Properties

Transactions are characterized by four properties popularly called ACID properties. To pass the ACID test, a transaction must be Atomic, Consistent, Isolated, and Durable. While this acronym is easy to remember, the meaning of each word is not obvious. Here is a brief explanation:

- Atomic: All steps in the transaction should succeed or fail together. Unless all the steps from a transaction complete, a transaction is not considered complete.

- Consistent: The transaction takes the underlying database from one stable state to another.

Many software packages (such as MSDTC) are available to assist with programming distributed transactions, which help ensure integrity by controlling commit and rollback behavior across all data resources, using mechanisms such as a two-phase commit and rollback.

MSDTC can only be used with applications that have compatible interfaces for transaction management. MSMQ, SQL Server, Oracle, Sybase, and several others are such applications, referred to as resource managers, which are currently available.

Manual and Automatic Transactions

Another way to slice and dice different kinds of transactions is as manual or automatic transactions.

A manual transaction model

- Allows you to use explicit instructions to begin and end the transaction to control the transaction boundary.
- Allows you to start a new transaction from within an active transaction (it supports nested transactions).

A set of objects, such as SqlConnection, SqlTransaction , etc., are provided by the ADO.NET data providers to help create a connection to the data source; begin, commit, or roll back the transaction; and finally close the connection manually.

An automatic transaction model wraps around a statement or a number of statements implicitly. In other words, additional statements simply enlist automatically in the current running transaction. If your transaction spans multiple transaction-aware resource managers (such as SQL Server or MSMQs), an automatic transaction is your best option. System.Transactions or COM+ will do all of the coordination work, causing extra overhead, but the application design will be much simpler, reducing coding requirements.

MSDTC is used to manage transactions in a distributed environment, which allows a .NET application to run a transaction that combines a number of diverse activities such as the following:

- Retrieving data from a SQL Server database
- Inserting an order into an Oracle database
- Writing a message to a Microsoft Message Queue

Transaction Vocabulary

There are some commands that are used frequently in the context of database transactions; they are BEGIN, COMMIT, SAVE, and ROLLBACK. These are the basic building blocks used in implementing transactions. Before going any further, let's take a quick look at what these commands do:

- BEGIN: Before executing any statements under a transaction, a transaction must be initiated; to do this, you use BEGIN.
- COMMIT: A transaction is said to be committed when all the changes that occurred during the transaction are written successfully to the underlying persistent store such as command.

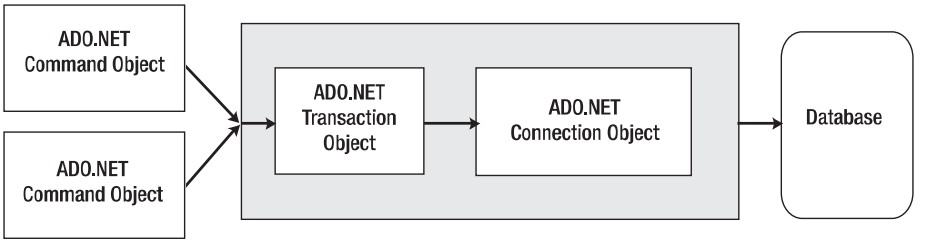
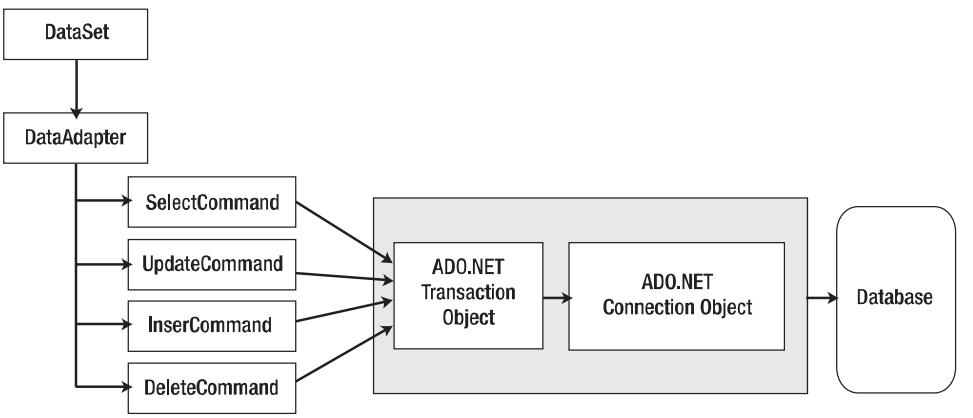


Figure 11-1.Transactions in a connected mode

In disconnected mode, generally, data is fetched first (usually one or more tables) into a DataSet object, the connection with the data source is closed, the data is manipulated as required, and then the data is updated back into the database. In this mode, the typical sequence of operations will be as follows:

1. Open a database connection.
2. Fetch the required data in a DataSet object.
3. Close the database connection.
4. Manipulate the data in the DataSet object.
5. Again, open a connection with the database.
6. Start a transaction.
7. Assign the transaction object to the relevant commands on the data adapter.
8. Update the database with changes from the DataSet.
9. Close the connection.

Figure 11-2 illustrates this sequence of events.



Implementing transactions in connected mode is relatively simple, as we have everything happening live; however, in disconnected mode, while updating the data back into the database, some care should be taken to account for concurrency issues. Also, depending on your architecture, it might be necessary to roll back any changes made to the DataSet that might have been done in a partially successful, but rolled back update. [1] An example demonstrating that can be seen in Chapter 10's Exercise 10.1.

The following section will discuss the transaction class. It will also look at the commonly used methods of the transaction class and typical ways of using these methods.

Transaction Class

The various .NET-managed providers available in the .NET Framework, like OleDb, SqlClient, OracleClient, and ODBC, etc., each has its own implementation of the transaction class: the OleDb data provider has the OleDbTransaction class, which resides in the System.Data.OleDb namespace; the ODBC data provider has the OdbcTransaction class, which resides in the System.Data.Odbc namespace, and so on for the other providers (SqlTransaction class and OracleTransaction class).

All of these classes implement the IDbTransaction interface from the System.Data namespace. Most of the properties and methods of these classes are identical; however, each has some specific methods of its own, as shall be discussed shortly.

This would be a good time to flip back to Chapter 2 and look at Figure 2-3 to examine the parallelism that various IDbTransaction implementations exhibit, such as SqlTransaction and OracleTransaction.

Methods of the Transaction Class

Commit and rollback were discussed earlier in this chapter. The transaction classes have two methods that will be used frequently that reflect these:

- Commit: This method identifies a transaction as successful. Once you call this method, all the pending changes are written permanently to the underlying database once this method returns without an error. The exact implementation depends on the data provider, but typically this translates to executing a COMMIT on the underlying database.
- Rollback: This method marks a transaction as unsuccessful, and pending changes are discarded. The database state remains unchanged.

Typically, both of these methods are used together. The following code snippet shows how they are used in the most common way:

C#

```
using (SqlConnection myConnection = new SqlConnection(connectionString))
{
    myConnection.Open();
    myTransaction = myConnection.BeginTransaction();
```

1. Of notable mention here are the AcceptChangesDuringUpdate and ContinueUpdateOnError properties

Writing Transactional Database Applications

Implementing a basic transaction using ADO.NET in an application can be fairly straightforward. The most common sequence of steps that would be performed while developing a transactional application is as follows:

1. Open a database connection using the `Open` method of the connection object.
2. Begin a transaction using the `BeginTransaction` method of the connection object. This method provides a transaction object that can be used later to commit or roll back the transaction. Note that the changes caused by any queries executed before calling the `BeginTransaction` method will be committed to the database immediately after they execute.
3. Set the `Transaction` property of the command object to the transaction object mentioned in step 2.
4. Execute the SQL commands using the command object. More than one command object may be used for this purpose as long as the `Transaction` property of all the objects is set to a valid transaction object.
5. Commit or roll back the transaction using the `Commit` or `Rollback` method of the transaction object.
6. Close the database connection.

Note that once you have started a transaction on a certain connection, all the queries that use that particular connection must fall inside the boundary of the transaction. For example, two `INSERT` queries and one `UPDATE` query can be executed inside a transaction. In the midst, if you were to execute a `SELECT` query without the transaction, you'd get an error indicating that a transaction is still pending. Also, one connection object can have only one pending transaction at a time.

In other words, once you call the `BeginTransaction` method on a connection object, you cannot call `BeginTransaction` again, until you commit or roll back that transaction. In such situations, you will get an error message stating that parallel transactions are not supported. To overcome this error, you may use another connection to execute the query.

There is, however, a new SQL Server 2005–specific feature called MARS that lets you maintain multiple active result sets on one connection, but not multiple transactions. Now because you can execute multiple interleaved command batches at the same time, it gets especially interesting in transactional scenarios. Essentially in MARS, nondeterministic transaction scenarios that involve multiple interleaved commands are disallowed. But that will be discussed in detail later in this chapter.

Implementing Transactions

So without much further ado, let's get our hands dirty and write an actual application demonstrating single database/nondistributed transactions using ADO.NET.

For the purposes of this application, a small Console application will suffice. As usual, let's begin with setting up the database for various examples in this chapter.

The database is a simple customers/products database with three tables: Customers, which holds the customer details; Products, which holds the products details; and CustomerProduct, which holds the many-to-many mapping between the Customers table and the Products table. The schema for the database is shown in Figure 11-3, or it may be created using the CreateDatabase.SQL file in the associated code download for this chapter (see the Downloads section of the Apress website at http://www.apress.com).

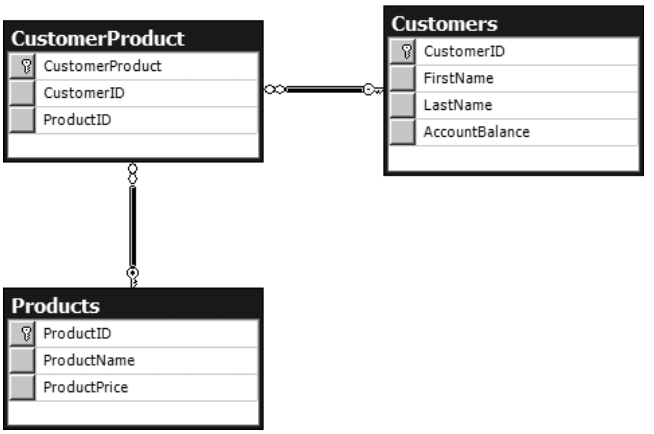


Figure 11-3.Sample database for a transactional application

To get started, go ahead and populate the Customers table with the rows as shown in Figure 11-4.

	CustomerID	FirstName	LastName	AccountBalance
1	1	Capt.	Kirk	100
2	2	Super	Man	100

Figure 11-4.Sample rows for the Customers table

Also, go ahead and add the rows for the Products table as shown in Figure 11-5.

	ProductID	ProductName	ProductPrice
1	1	Cape	4
2	2	Phasor Gun	9

Figure 11-5.Sample rows for the Products table

As you can see, both of our customers start with an account balance of 100 dollars. What this application will try to do is place an order for Superman. Superman will purchase a cape, and because a cape is worth 4 dollars, his account balance will reduce from 100 to 96 dollars. This operation will involve two SQL queries. The first query to place an order for Superman

```
Insert into CustomerProduct
    (CustomerID, ProductID)
Values
    (2, 1)
```

The second query to update his account balance to 96 will look like this:

```
Update Customers
    Set AccountBalance = 96 [2]
Where
    CustomerID = 2
```

One way to make this work would be to wrap up the two queries in a single transaction at the database level. This can be done easily using a group of statements that look like this:

```
Begin Transaction
Insert into CustomerProduct
    (CustomerID, ProductID)
Values
    (2, 1)
Update Customers
    Set AccountBalance = 96
Where
    CustomerID = 2
Commit
```

This can be wrapped as inline SQL or as a stored procedure call. What is interesting in this approach is that it doesn't let you have fine-grained control (at least not in .NET code) between the insert and update. That kind of control is especially important when dealing with disconnected DataSets. For instance, if between the insert and update someone else placed an order for Superman for a product that costs 100 dollars, then per the previous commands, without a concurrency check built in, your issued commands would result in a negative balance for Superman (which should not be allowed). That is obviously not the right thing to do. It corrupts your database and makes Superman mad at you—you don't want either to happen.

Thus, a better way to implement this scenario would be to have concurrency checks and transactions based upon individual ADO.NET commands wrapped up in DbTransaction objects. In code, this would look like Listings 11-1 and 11-2. (You can find the code for Listings 11-1 to 11-6 in the associated code download for this chapter under Exercise 11.1.)

Listing 11-1. Transactional Command Execution in C#

```
static void Main(string[] args)
{
    using (SqlConnection testConnection = new SqlConnection(connectionString))
```

2. Bonus points for you if you immediately jumped in your seat and thought, "A better way to write this query would be Set AccountBalance = AccountBalance … 4." You are right, but I wanted to exemplify the query as is because what if the query was the result of a disconnected DataSet with the value "96"

Examining the Effect of Isolation Levels

Now that you know how to implement basic transactions, letÕs go into some more detail and look at how you can control the behavior of a transaction, and how a transactional batch of queries affects other queries running on the same resource. This is typically done by specifying an isolation level on your transactions.

This section will cover isolation levels and how they affect your application. In doing so, youÕll encounter some terms that are frequently used while discussing isolation levels. In addition, youÕll see how to set isolation levels in your code.

What Are Isolation Levels?

An isolation level is a measure of the extent to which changes made outside a transaction are visible inside that transaction. It determines how sensitive your transaction is to changes made by other transactions.

For example, by default in SQL Server, if two transactions are running independently of one another, then records inserted by one transaction are not visible to the other transaction, unless the first transaction is committed.

The isolation level concept is closely related to the locks concept, because by determining isolation levels for a given transaction you can determine how long the given transaction must hold locks on resources to protect them against changes that may be made by others.

You may wish to alter the behavior of the transactions so that the second transaction can view records inserted by the first one. This may amount to reading invalid data (because it isnÕt committed yet), which is typically referred to as a *dirty read*. But, if you wish to shoot yourself in the foot by reading invalid data, ADO.NET does give you the facility to do so. [3] You can achieve this through setting isolation levels appropriately. For an ADO.NET programmer, the isolation level can be set via isolation level enumerated values.

Some Related Terms

Before going any further, itÕs important to understand some terms that are used frequently when discussing isolation levels:

- Dirty read: A dirty read is a condition when a transaction reads data that has yet to be committed. Consider a case where transaction A inserts some records into a table, but is pending. Transaction B reads these records. Now, if transaction A rolls back, transaction B will refer to data that is invalid.

- Nonrepeatable read: Consider a case where transaction A reads a record from a table. Transaction B then alters or deletes the records and commits the changes. Now, if transaction A tries to re-read the record, it will either be a different version or it will not be available at all. Such a condition is called a nonrepeatable read.

---

3. ÒShoot yourself in the footÓ may be a very strong term, but in certain instances, it might be necessary to do this. If you have a rather busy table (lots of inserts and updates), and some part of the application doesnÕt care about 100 percent data accuracy, but does care about response time for its selects, then this would be a possible way out of that scenario. Though, the *Snapshot*isolation takes care of many of those scenarios.

- Phantom read: Suppose that transaction A has some criteria for record selection. Initially, transaction A has, say, 100 rows matching these criteria. Now transaction B inserts some rows that match the selection criteria of transaction A. If transaction A executes the selection query again, it will receive a different set of rows than in the previous case. The rows added in this way are called phantom rows.

Possible Isolation Levels in ADO.NET

Let's now look at the different isolation level values that you can implement with ADO.NET. These values are accessible via the `IsolationLevel` enumeration:

- `Chaos`: The pending changes from more highly isolated transactions cannot be overwritten. This setting is not supported by SQL Server or Oracle.

- `ReadUncommitted`: In this case, a dirty read is possible. This means that no shared locks are placed, and no exclusive locks are honored. This type of isolation level is appropriate when you want to work with all the data matching certain conditions, irrespective of whether it's committed or not.

  Generally, you would use this isolation level where the application doesn't require guaranteed accuracy in the data it queries, and where the priority is performance over accuracy. If performance is the chief criterion in your application design, you should prefer this isolation level over others, but do understand that it comes at the cost of possibly inaccurate data. Note that the `Snapshot` isolation level described shortly provides a much better alternative than `ReadUncommitted` in doing nonblocking `SELECT` queries. Even though you don't get the latest data, at least you get committed data. A good, practical use of `ReadUncommitted` is data monitoring by an administrator. I recommend that you use this isolation level sparingly.

  In this scenario, there is no concurrency model being followed, unless you implement your own homegrown concurrency check model such as `Timestamps`, `RowVersions`, etc. These have been discussed in depth in the last chapter. This isolation level is not supported in Oracle.

- `ReadCommitted`: Shared locks are held while the data is being read by the transaction. This avoids dirty reads, but the data can be changed before a transaction completes. This may result in nonrepeatable reads or phantom rows. This type of isolation level is appropriate when you want to work with all the data that matches certain conditions, and is committed.

  Shared locks are locks that are placed when a transaction wants to read data from the database, and no exclusive lock is already held on that data item. No other transactions can modify the data while shared locks exist on a table or tables. Exclusive locks are the locks that prevent two or more transactions modifying data simultaneously. An exclusive lock is issued when a transaction needs to update a table or tables, and no other locks are already held on the respective tables. Because an exclusive lock is needed, this would qualify as pessimistic locking on the database.

  This isolation level can be used where you need transactional consistency for long running queries, but you do not care if you may get nonrepeatable reads. This isolation level is supported in Oracle. Also, this is the default isolation level for both `SqlTransaction`

Listing 11-8. Setting Up Two Transactions with Isolation Levels ReadCommitted and ReadUncommitted in Visual Basic .NET

```
Dim connection1 As SqlConnection = New SqlConnection(connectionString)
Dim connection2 As SqlConnection = New SqlConnection(connectionString)

Dim command1 As SqlCommand = connection1.CreateCommand()
Dim command2 As SqlCommand = connection2.CreateCommand()

connection1.Open()
connection2.Open()
Dim transaction1 As SqlTransaction = _
    connection1.BeginTransaction()
command1.Transaction = transaction1
Dim transaction2 As SqlTransaction = _
    connection2.BeginTransaction(IsolationLevel.ReadUncommitted)
command2.Transaction = transaction2
```

Also note that in this code, two commands are created that will enlist in the specified transactions. The next step is to set up command text for both commandsÑthe first that will insert a row and the second that will check for the existence of the inserted row.

These commands are then executed and results are shown. The transaction is then rolled back and the results are shown. This can be seen in Listings 11-9 and 11-10.

Listing 11-9. Working with Two Commands on the Same Table, in Two Transactions on Different Isolation Levels in C#

```
SqlDataReader myReader;
try
{
    command1.CommandText =
        "INSERT INTO CUSTOMERS (FIRSTNAME, LASTNAME, ACCOUNTBALANCE) "
              + "VALUES ('Bat', 'Man', 100)";
    command1.ExecuteNonQuery();

    command2.CommandText =
        "SELECT FIRSTNAME, LASTNAME from CUSTOMERS where FIRSTNAME = 'Bat'";
    myReader = command2.ExecuteReader();

    Console.WriteLine("Results when the transaction is midway:");
    if (!myReader.HasRows)
    {
        Console.WriteLine("No Rows Found");
    }
    while (myReader.Read())
    {
        Console.WriteLine(
             "FirstName: " + myReader[0] + " and LastName: " + myReader[1]);
```

If you were to make a slight modification and change the isolation level of the first transaction to Serializable and then run the code again, you'd notice that the code will simply hang at the SELECT command execution. This is because isolation level Serializable will block any further reads from the underlying data source (row, page, or table) until the first transaction has either rolled back or committed.

Now try another experiment. Set the isolation level on both transactions to Snapshot and run the application again. You should probably see an exception as shown in Figure 11-7.

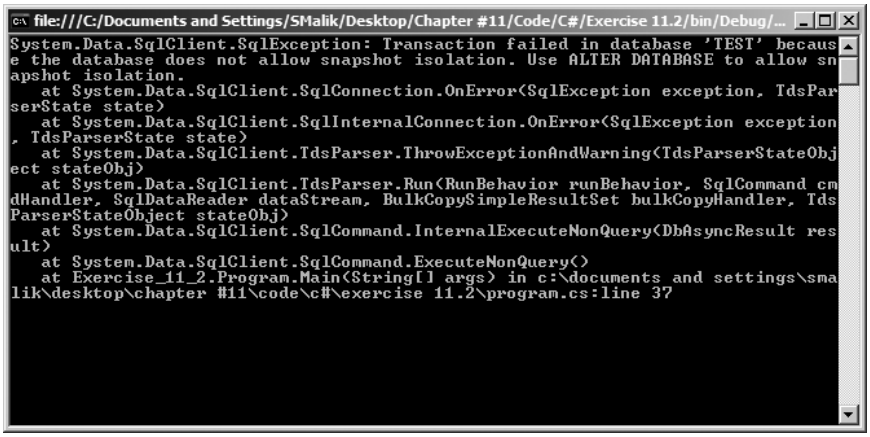


Figure 11-7.Running the example with IsolationLevel = Snapshot without enabling them in the database first

You get this error because this isolation level is disabled by default. To enable it, simply run the following command:

```
ALTER DATABASE TEST SET ALLOW_SNAPSHOT_ISOLATION ON
```

Run the application again and you should see the results shown in Figure 11-8.

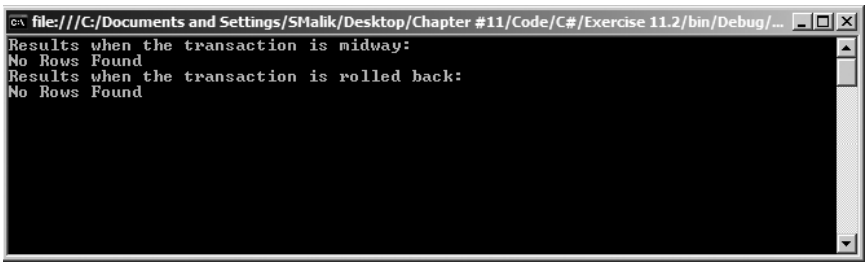


Figure 11-8.Running the example with IsolationLevel = Snapshot after enabling them in the database

C#

```
cmd1.Connection = myConnection ;
cmd2.Connection = myConnection ;

SqlDataReader firstReader = cmd1.ExecuteReader() ;
SqlDataReader secondReader = cmd2.ExecutReader() ;
```

VB.NET

```
cmd1.Connection = myConnection
cmd2.Connection = myConnection

Dim firstReader As SqlDataReader = cmd1.ExecuteReader()
Dim SqlDataReader As SqlDataReader = cmd2.ExecutReader()
```

The implications of MARS on a transactional scenario are quite profound, but before I discuss those, let me quickly discuss a short primer on how MARS works. ItÕs important to note that MARS in SQL Server works only in SQL Server 2005[4] and above, and in SQL Server 2000 and below you are still restricted to running one command per connection.

SQL Server is queried using the SQL SELECT command. By executing the SELECT command, results are copied into prereserved network buffers, which are then sent to the caller. Network write operations will succeed and free up used buffers as long as the client is reading from the buffers. If, for some reason, the client is unable to read, the network buffers will fill up and be unusable, and either the server will force a timeout or the client will catch up. This is typically referred to as Òdefault result setsÓ or firehose cursors, a.k.a. data readers.

Now that you have the ability to read the results of a SQL query, you could write up code as shown in Listings 11-11 and 11-12 that attempt to run two firehose cursors at the very same time.

Listing 11-11.Attempting to Run Two Data Readers on the Same Open Connection Concurrently in C#

```
SqlCommand cmd = conn.CreateCommand();
SqlCommand cmd2 = conn.CreateCommand();

cmd.CommandText=
    "select * from customers";
cmd2.CommandText=
    "Update Customers set FirstName='Tarzan' where CustomerId = @CustomerID";

SqlParameter custID=cmd2.Parameters.Add("@CustomerID", SqlDbType.Int);
```

4. As youÕll see later in this chapter, you also need to be using the SqlClient from ADO.NET 2.0 or the

MarsConn=yes

Or, if you are connecting using OleDb service components, add

MARS Connection=True

For ODBC, you need to add

MARS_Connection=yes

If you forget to add these keywords, the behavior will revert to what it was like for previous versions of OleDb providers and ODBC drivers. That is, you will start using implicit connections.

MARS and Transactions

In a MARS-enabled world, you can run multiple interleaved commands and maintain parallel result sets, but you cannot run multiple transactions at the same time on the same connection.

This means that you have to be careful when mixing older versions of OleDb with transactions of a high isolation level. As mentioned earlier, it might appear on the surface with older versions of OleDb that multiple commands are, indeed, working in the same transaction and same connection; however, previous versions of OleDb will simply use a new connection and you might end up in a deadlock. Older versions of ODBC, on the other hand, respond with a "connection busy" error.

Let's say that you tried doing the same in a MARS-enabled world. This could include SqlClient connecting with SQL Server 2005, or OleDb or ODBC connecting with the SQL Native Client for SQL Server 2005.

In a MARS-enabled world, a transaction on multiple commands including a transaction with a higher isolation level can be made to work. This is not because MARS can handle multiple parallel transactions; in fact, you cannot have multiple parallel transactions on one connection that enables multiple interleaved commands and parallel result sets. But because all those commands are indeed working on one single connection, and maintaining their result sets in parallel while remaining on one single transaction at the same isolation level on the same connection, is how you are able to run multiple interleaved commands on the same transaction and on the same connection. And once you achieve multiple interleaved commands in one transaction on one single connection, the isolation level could be whatever you wish.

This raises interesting questions and possibilities about how transactions work in a MARS-enabled world.

If you attempt to [6] run two commands on the same transaction, then the regular interleaving rules apply. So what are the interleaving rules?

If the command being executed is reading results and belongs to one of the commands that allows interleaving (SELECT, FETCH, READTEXT), and another operation attempts to modify the data (INSERT, UPDATE, etc.), then, as per the interleaving rules, the DQL (Data Query Language) statement will need to yield to the DML statement. Once the read operation has been blocked, the read operation ensues after the changes have been completed by the DQL command.

---

ministic transactions.

MARSÑA LITTLE PRACTICAL WORD OF ADVICE

You might wonder why I chose to mention the very specific details of MARS such as interleaving rules, serialized commands versus interleaved commands, and their effect on transactions. It is extremely important that you understand all of these rules before you decide to use MARS in your application architecture.

As mentioned previously, whether a command will be interleaved or serialized cannot be accurately determined. This is because it depends on a lot of features, stemming from the fact that you cannot accurately predict when the full result set for a query has been put on the wire for the client to consume. So what does this mean? Well, consider the code shown here:

C#

```
using (SqlConnection testConnection = new SqlConnection(marsConnectionString))
{
    testConnection.Open();
    SqlCommand myCommand = testConnection.CreateCommand();
    myCommand.CommandText = "exec sp_who";
    SqlDataReader myDataReader =
     myCommand.ExecuteReader(CommandBehavior.Default);
    Boolean flag = myDataReader.Read();
    try
    {
        SqlTransaction myTran = testConnection.BeginTransaction();
    }
    catch(Exception e){
        Console.WriteLine(e);
    }
}
```

VB.NET

```
Using testConnection As SqlConnection = _
      new SqlConnection(marsConnectionString)
    testConnection.Open()
    Dim myCommand As SqlCommand =  testConnection.CreateCommand()
    myCommand.CommandText = "exec sp_who" 'this does not repro the problem.
    Dim myDataReader As SqlDataReader = _
      myCommand.ExecuteReader(CommandBehavior.Default)
    Dim flag As Boolean =  myDataReader.Read()
    Try
        Dim myTran As SqlTransaction =  testConnection.BeginTransaction()
    Catch e As Exception
        Console.WriteLine(e)
    End Try
End Using
```

This code will work fine because there is just one result set being maintained and a transaction in this scenario should not cause a problem. But now go ahead and change the text of myCommand to

would roll back the first and second executed commandÕs changes, in the DataSet itself, the row states for the first two rows would have changed back to UnModified. Thus, at this point, you would have no easy recourse of setting the row states back to what they should have been.

In .NET 1.1, the solution for this problem would be to extract a smaller DataSet with only the changes using the GetChanges method, and work with that smaller DataSet instead. In the event of a successful transaction, you could then refresh the data and Merge the new and fresh data back from the database into the DataSet.

There are, however, numerous problems with this approach. One big problem is that Merge and GetChanges are potentially expensive commands to run. In .NET 1.1, the time required to run Merge or GetChanges increases exponentially with the increase in complexity in the DataSet. In .NET 2.0, the internal logic is vastly improved and the decrease in performance is linear, but they are still expensive commands to run. A bigger problem is that if you were using any of the events, such as RowUpdated, to set error values in your DataSet (which in this case would be the smaller DataSet extracted using GetChanges), then all your user-friendly errors will now have to be Òhand mergedÓ back to the original DataSet, or they will simply be lost. Then there are other issues surrounding the fact that you might have specified autoincrement primary keys in your original DataSet, and an automated merge will simply mess up all the primary-key values extracted out of the database. In fact, I am sure there are even more scenarios you could think of where this approach would cause problems.

It is for this purpose that .NET 2.0 has introduced a new property, called AcceptChangesDuringUpdate, on the DataAdapter object. Setting AcceptChangesDuringUpdate to true instructs the data adapter that it should not change row states on rows as it is executing the commands on various rows. Instead, you will have to call AcceptChanges on the DataSet as a whole at the end of your update. This can be seen in the associated code download under the AccChanges code sample, or in the following code snippet:

C#

```
testConnection.Open();
SqlTransaction myTransaction = testConnection.BeginTransaction();
sqlDA.UpdateCommand.Transaction = myTransaction;
sqlDA.AcceptChangesDuringUpdate = false;

try
{
    sqlDA.Update(ds);
    myTransaction.Commit()
    ds.AcceptChanges();
}
catch (Exception ex)
{
    MessageBox.Show(ex.ToString());
}
```

typically referred to as the Distributed Transaction Coordinator (DTC) [7] also referred to as the transaction manager.

The transaction coordinator that ships with Windows is the Microsoft Distributed Transaction Coordinator (MSDTC). MSDTC is a Windows service that provides transactional coordination for distributed applications.

Here is a typical flow of a distributed transaction:

1. An application begins a transaction requesting one from the MSDTC. This application is commonly also referred to as the initiator .
2. The application then asks the RM to do its work as a part of the same transaction (this is actually configurable), and the RMs register with the transaction manager as a part of the same transaction. This is commonly referred to as enlisting in a transaction.
3. If all goes well, the application commits the transaction.
4. If something fails, either step can issue a rollback.
5. In either case, the transaction coordinator coordinates with all the RMs to ensure that they all either succeed and do the requested work, or they all roll back their work.

MSDTC is used by MTS/COM+, System.EnterpriseServices , and, starting with .NET 2.0, in the new System.Transactions namespace. In fact, you can bypass both COM+ and .NET and simply use the MSDTC proxy msdtcprx.dll .

Two-Phase Commits

In a distributed transaction scenario, various RMs need to implement a reliable commit protocol, the most common implementation of which is known as a two-phase commit.

In a two-phase commit, the actual commit for the work is split into two phases. The first phase involves preparing the changes required for the commit. At this point, the RM communicates to the transaction coordinator that it has its changes prepared and ready to be committed, but not actually committed yet.

Once all the RMs give the green flag to the transaction coordinator, the transaction coordinator then lets everyone know that it is okay to go ahead and commit their changes.

This picture is obviously oversimplified, and entire books have been written on transaction coordinators and RMs, but that discussion is beyond the scope of this book.

This brings up an interesting question: Who can participate in a distributed transaction? ItÕs not just the database, or a number of databases that can participate in a distributed transaction. In fact, anything that has the ability to enlist itself in an MSDTC transaction can enlist itself in a distributed transaction managed by the MSDTC. For instance, MSMQ can enlist in a transaction that has two other SqlConnection objects that connect to two different databases.

Implementing a Distributed Transaction: The .NET 1.1 Way

To appreciate the beauty of .NET 2.0Õs distributed transactions implementation, itÕs first necessary to become familiar with the ugly duckling of .NET 1.1Õs implementation. So letÕs look at that

first. If you are already familiar with this, or not interested in becoming friends with the ugly duckling, you may jump directly to the ÒImplementing a Distributed Transaction: The .NET 2.0 WayÓ section.

In .NET 1.1, distributed transactions can be implemented using the System.EnterpriseServices namespace, which provides you with a convenient method to access MSDTC functionality.

To do so, your project must be a class library. This is so because Component Services is designed to run without a UI. The class that will enlist itself in an MSDTC transaction needs to inherit from ServicedComponent and it will need to be in the Global Assembly Cache (GAC). Because it needs to be in the GAC, you will also need to strongly name it, using the sn.exe tool that comes with the .NET SDK.

Finally, you need to specify the TransactionAttribute at the top of the class specifying its behavior when it encounters an MSDTC transaction. This is done using the TransactionOption enumeration. The various TransactionOption enumeration values are shown in Table 11-1.

Table 11-1.TransactionOption Enumeration Values

Constant	Description
Disabled	This component does not participate in transactions. This is also the default value.
NotSupported	This component runs outside the context of a transaction.
Supported	This component participates in a transaction if one exists, but does not require a transaction or creates one for itself.
Required	This component must have a transaction. If no transaction exists, one will be created. If one exists, it will participate in the transaction.
RequiresNew	This component requires a transaction and will create a brand new transaction for itself.

Thus, in this manner, you could create a number of components that enlist themselves in an MSDTC transaction via Enterprise Services. Each of these calls SetCompleteupon success or SetAbort upon failure. By calling SetComplete, you are notifying the transaction coordinator that you are ready and willing with your changes, pending others. If anyone else in the chain calls SetAbort, then all bets are off and no changes are committed.

You can see example code for such a component in Listings 11-15 and 11-16.

Listing 11-15.ODP.NET Component with the Ability to Enlist Itself in a Distributed Transaction Using Enterprise Services in C#

```
using System;
using System.EnterpriseServices;
using System.Data;
using Oracle.DataAccess.Client;
using Oracle.DataAccess.Types;
using System.Runtime.CompilerServices;
using System.Reflection;

// returns the name of Assembly, that is used by CLR to bind the Assembly
```

```
// Name of a  file containing the strong name key
[assembly: AssemblyKeyFile(@"DistributedTransaction.snk")]

namespace DistributedTransactionExample
{
    // Apply a  Transaction Attribute class to this class
    [Transaction(TransactionOption.Required)]

    // Inherit from ServicedComponent class.
    public class DistributedTransaction : ServicedComponent
    {
        public static void updateAmounts(int id, float amount)
        {
            OracleCommand debitCommand ;
            OracleCommand creditCommand ;
            try
            {
                string cmdTxt =
        "UPDATE Credit SET CreditAmount =" + amount + " WHERE CreditID =" + id;

                // OracleCommand for credit database
                creditCommand =
                    new OracleCommand(cmdTxt, creditConnectionString);

                // OracleCommand for debit database
                debitCommand = new
                    OracleCommand(cmdTxt, debitConnectionString);

                // Executes the Oracle Commands
                creditCommand.ExecuteNonQuery();
                debitCommand.ExecuteNonQuery();
            }
            finally
            {
                // Release all resources held by OracleCommand objects
                creditCommand.Dispose();
                debitCommand.Dispose();
            }
         }
    }
}
```

Listing 11-16.ODP.NET Component with the Ability to Enlist Itself in a Distributed Transaction Using Enterprise Services in Visual Basic .NET

```
Imports System
Imports System.EnterpriseServices
```

The application you are going to see involves two databases. The first database is the Credits database, with one table ÒCreditsÓ. Credits simply has a CreditID and a CreditAmount. Similarly, there is a second table called the ÒDebitsÓ table. The Debits table has a DebitID and a DebitAmount. You can find the relevant SQL script to set up these databases in the associated code download under the CreateDistributedDatabase.sql file in Exercise 11.4.

The code you are going to see attempts to wrap the two matching insert operations into one transaction. In other words, if a debit fails, the previously inserted credit must also be rolled back.

The code for this example is really simple. It can be seen in Listings 11-17 and 11-18. There is no need for strong names, segregating your code into separate assemblies, or putting them in the GAC. All of that mess is simply replaced by the code you see here:

Listing 11-17. Implementing a Distributed Transaction in .NET 2.0 Using C#

```
static void Main(string[] args)
{
    try
    {
        using (TransactionScope myTransaction = new TransactionScope())
        {
            using (SqlConnection connection1 = new SqlConnection(connectionString1))
            {
                connection1.Open();
                SqlCommand myCommand = connection1.CreateCommand();
                myCommand.CommandText =
                    "Insert into Credits (CreditAmount) Values (100)";
                myCommand.ExecuteNonQuery();
            }

            Console.WriteLine(
               "The first connection transaction has done its work" +
               ", moving on to the second.");
            Console.ReadLine();

            using (SqlConnection connection2 = new SqlConnection(connectionString2))
            {
                connection2.Open();
                SqlCommand myCommand = connection2.CreateCommand();
                myCommand.CommandText = "Insert into Debits(DebitAmount) Values (100)";
                myCommand.ExecuteNonQuery();
            }
            myTransaction.Complete() ;
        }
    }
    catch (System.Exception ex)
    {
        Console.WriteLine(ex.ToString());
    }
```

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED
BEGIN TRANSACTION
SELECT CREDITAMOUNT FROM CREDITS
COMMIT
```

youÕll notice that SQL Server 2005 reports that one row has been inserted so far, as shown in Figure 11-10. Do note, however, that since this insert is contingent upon the second connectionÕs insert, this is a dirty read. But because, in this instance, you are interested in peeking beneath the action and you do intend to do a dirty read, you can use the uncommitted isolation level to actually read the halfway modified row. Just for fun, can you guess what would happen if you were to remove the isolation level and then run only the Select statement? Try it!! Here is a hint: the distributed transaction is executing the transaction at a high isolation level, such as Serializable .

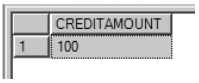


Figure 11-10.Dirty read as a result of the first part of a distributed transaction

Another thing to note here is that if you go to the Control Panel Administrative Tools Component Services applet and navigate through the tree on the left side to view the transaction list (as shown in Figure 11-11), nothing appears under the Distributed Transaction CoordinatorÕs Transaction List, as long asSqlConnection connection2 doesnÕt come into the picture.[8]

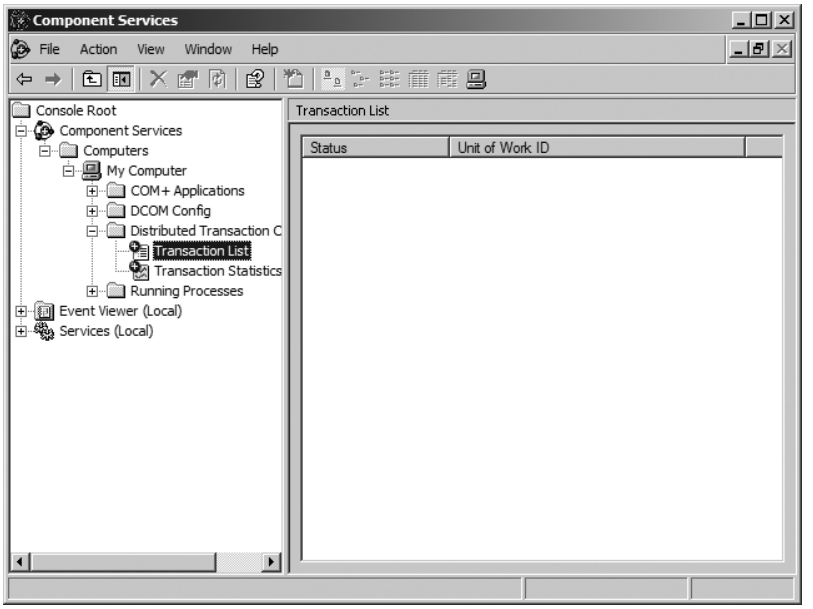


Figure 11-11.The Transaction List does not show any transactions until a second RM comes into the picture.

---

erver 2005.

Now, step through the code in debug mode. As soon as you trip over the connection2.Open line in code, youÕll notice that a transaction appears in the DTCÕs list of transactions. This is shown in Figure 11-12.

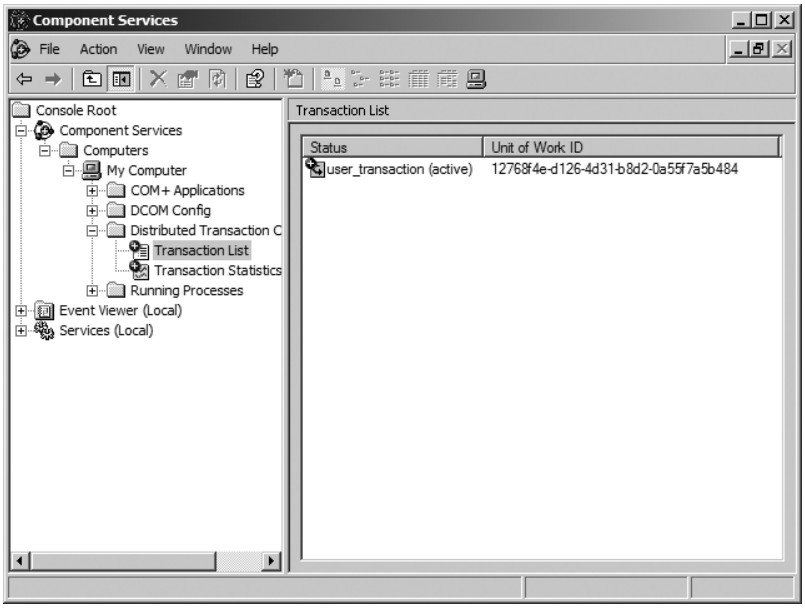


Figure 11-12.The Transaction List shows a transaction only when a second RM comes into the picture.

This is quite an important concept that is typically referred to as promotable enlistment of transactions. Do note, however, that of the currently implemented .NET data providers shipping with .NET 2.0, promotable enlistment is supported only by SqlConnection connecting with SQL Server 2005. Other scenarios, such as SqlClient connecting with SQL Server 2000, OracleConnection, etc., do work with distributed transactions, but they are handled exclusively by MSDTC.

Promotable Enlistment: A Quick Primer

MSDTC is expensive. It generates a lot of network traffic [9] and it possibly could run into network issues, such as firewalls, etc. Thus, itÕs unfair that every single transaction, whether or not it needs full distributed support, should pay the price of MSDTC. For instance, depending on your logic, your database transaction might be limited to a single database. In that instance, it would be overkill to involve MSDTC.

ItÕs for situations like these that a new facade layer called the Lightweight Transaction Manager (LTM) has been introduced in .NET 2.0. The LTM is what manages a transaction, as long as it doesnÕt need MSDTC.

---

9. To be specific, MSDTC generates a lot of network roundtrips. The actual data transferred isnÕt that mance.

- At least one durable resource that doesn't support single-phase notifications is enlisted in the transaction
- At least two durable resources that support single-phase notifications are enlisted in the transaction
- A request to marshal the transaction to a different appdomain or different process is done

Being able to limit transactions within the LTM significantly reduces the overload of creating and managing a transaction. The good news is that as long as the RM is promotable enlistment–aware, you (as a user of that RM) get this performance benefit for free. You do, however, have to understand how promotable enlistment works, should you choose to implement your own RMs.

This, as you can tell, has a significant positive impact on performance; however, in the case of SQL Server 2005, it gets even better.

Usually in distributed transactions involving databases, the isolation level of such transactions is `Serializable`. This ensures the best database consistency, at the cost of expensive locks and poor concurrent performance. However, with SQL Server 2005, as long as there is only one RM involved, you can still get away with a lower isolation level, such as `ReadCommitted`. As soon as the second RM comes into picture, the isolation level could be [10] bumped up to `Serializable`.

Do note, however, that by using one of the overloads of `TransactionScope`'s constructor, you can specify the desired or suggested isolation level that the RMs should use. Which isolation level the RMs choose to use depends on their implementation.

System.Transactions: Manually Enlisting and Multithreaded Environments

So how would you use `System.Transactions.Transaction` in a multithreaded environment? An instance of `System.Transactions.Transaction`, which can be accessed using `Transaction.Current`, is stored at thread-level storage. This means if you had, say, two instances of `SqlConnection` on different threads (or any other RM for that matter), in order for them to enlist within the same distributed transaction, you'd need to come up with a thread-safe approach.

Let me kill two birds with one stone with an example that demonstrates manual enlisting in a distributed transaction and a thread-safe approach of letting RMs in multiple threads enlist in the same transaction (see Exercise 11.5 in the associated code download).

Here are the two tricks I am going to use. In order for a `SqlConnection` instance to enlist itself in a transaction, it needs to use the following code snippet:

C#

```
connection2.EnlistTransaction(tran);
```

VB.NET

```
connection2.EnlistTransaction(tran)
```

---

10. Why "could be"? Because the final decision rests on the RM's `SqlConnection`. In certain instances, it in a distributed transaction.

```
        // Do something here - this connection is manually enlisted.
        tran.Rollback(); // ok to do
    }
    isThreadRunning = false ;
}
```

VB.NET

```
Private Sub ThreadEntryPoint(ByVal transactionInstance As Object)
    isThreadRunning = true
    Dim tran As Transaction = CType(transactionInstance, Transaction)
    Using connection2 As SqlConnection = New SqlConnection(connectionString2)
        connection2.Open()
        connection2.EnlistTransaction(tran)
        ' Do something here - this connection is manually enlisted.
        tran.Rollback() ' ok to do
    End Using
    isThreadRunning = false
End Sub
```

Take note in this code that one of the comments says that it is okay to `Rollback`. This means that a cloned transaction can `Rollback`, but it cannot `Commit`The full code can be seen in Listings 11-19 and 11-20.

Listing 11-19.Thread-safe and Manual Enlistment in a System.Transactions.Transaction Using C#

```
private static string connectionString1 = "...";
private static string connectionString2 = "...";
private static bool isThreadRunning = false;
static void Main(string[] args)
{
    try
    {
        using (TransactionScope myTransaction = new TransactionScope())
        {
            Thread myThread;
            myThread =
                new System.Threading.Thread(new
                    ParameterizedThreadStart(ThreadEntryPoint));
            Transaction tran = Transaction.Current.Clone();
            myThread.Start(tran);

            using (SqlConnection connection1 = new SqlConnection(connectionString1))
            {
                connection1.Open();
                // Do something here - this connection will auto-enlist
            }
```

```
            // Wait for the other thread to finish
            while (isThreadRunning)
            {
                Console.Write("\rWaiting for thread to finish .. ");
            }
            myTransaction.Complete();
        }
    }
    catch (System.Exception ex)
    {
        Console.WriteLine(ex.ToString());
    }
}

private static void ThreadEntryPoint(object transactionInstance)
{
    isThreadRunning = true;
    Transaction tran = (Transaction)transactionInstance;
    using (SqlConnection connection2 = new SqlConnection(connectionString2))
    {
        connection2.Open();
        connection2.EnlistTransaction(tran);
        // Do something here - this connection is manually enlisted.
        tran.Rollback(); // ok to do
    }
    isThreadRunning = false;
}
```

Listing 11-20.Thread-safe and Manual Enlistment in a System.Transactions.Transaction Using Visual Basic .NET

```
Private connectionString1 As String = "..."
Private connectionString2 As String = "..."
Private isThreadRunning As Boolean = False

Sub Main()
    Try
        Using myTransaction As TransactionScope = New TransactionScope()
            Dim myThread As Thread
            myThread = _
                New System.Threading.Thread( _
                    New ParameterizedThreadStart(AddressOf ThreadEntryPoint))
            Dim tran As Transaction = Transaction.Current.Clone()
            myThread.Start(tran)

            Using connection1 As SqlConnection = _
                New SqlConnection(connectionString1)
```

```
            ' Do something here - this connection will autoenlist
        End Using
        ' Wait for the other thread to finish
        While (isThreadRunning)
            Console.Write("\rWaiting for thread to finish ..")
        End While
        myTransaction.Complete()
    End Using
  Catch ex As System.Exception
    Console.WriteLine(ex.ToString())
  End Try
End Sub

Private Sub ThreadEntryPoint(ByVal transactionInstance As Object)
  isThreadRunning = True
  Dim tran As Transaction = CType(transactionInstance, Transaction)
  Using connection2 As SqlConnection = New SqlConnection(connectionString2)
    connection2.Open()
    connection2.EnlistTransaction(tran)
    ' Do something here - this connection is manually enlisted.
    tran.Rollback() ' ok to do
  End Using
  isThreadRunning = False
End Sub
```

Now, much like Exercise 11.4, if you put a breakpoint on the following line:

C#

```
connection2.EnlistTransaction(tran);
```

VB.NET

```
connection2.EnlistTransaction(tran);
```

you would note that the transaction appears in the DTC as soon as this line is executed. This confirms that, indeed, the second connection is now listed in the transaction and the transaction has been promoted to the DTC; whereas before, this lineÕs execution (connection2) was not enlisted, thus manual enlistment is working. Also, because the transaction has executed a roll-back in the subsequent line of code, youÕll note that in the original thread the transaction will not successfully commit and youÕll get an exception as shown here:

```
System.Transactions.TransactionAbortedException: The transaction has aborted.
```

Thus, in this case, the application creates a clone of a `Transaction` object instance and you can use that to enlist the `SqlConnection` instance in a distributed transaction when you consider it necessary to do so. Once you have a`Transaction` instance, you can then enlist `SqlConnection` objects inside the current transaction.

There is yet another class that you could use for manual enlistment: `System.Transactions.`

new instance of `CommittableTransaction`,

only the creator of the distributed transaction can commit the transaction. In other words, copies of a CommittableTransaction obtained through System.Transactions.Transaction.Clone cannot be committed.

Note that in .NET 1.1, you could achieve the same using System.EnterpriseServices using the EnlistDistributedTransaction method. Also, it is important to note that once a connection is enlisted, it cannot be unenlisted.

These examples using System.Transactions will also work for other databases, such as Oracle. However, promotable enlistment works only for SqlConnection connecting with SQL Server 2005.

Judicious Use of Transactions

Even though ADO.NET provides good support for transactions, itÕs not always necessary to use transactions. A more accurate statement perhaps could be that you should use transactions when you can, but not overuse them. Every time you use a transaction, you carry some overhead. Plus, transactions may involve some kind of locking of table rows. Thus, unnecessary use of transactions will cause performance penalties. As a rule of thumb, use a transaction only when your operation requires one. For example, if you are simply selecting records from a database, or firing a single query, then most of the time you will not need an explicit transaction because your statement is already wrapped in an implicit transaction. However, as mentioned previously, itÕs important to note that in multistatement updates, transactions can actually make the operation faster, rather than slower. Also, if it comes down to a choice between a few milliseconds saved versus compromising your data sanctity, the right answer is to not worry about the milliseconds and keep your data clean.

This effect is even more pronounced when using distributed transactions involving MSDTC. Because true distributed transactions (involving more than one RM in different appdomains or durable RMs) involve running the underlying ADO.NET transactions in IsolationLevel. Seri alizable , and because you could run into network issues, and finally because MSDTC generates a lot of network traffic, [11] distributed transactions are typically extremely expensive and problematic.

As a matter of fact, in a database scenario or ADO.NET world, you should prefer to use distributed transactions only when you have a nondatabase entity, such as an RM that you wrote yourself enlisting itself in the transaction, or when you cannot wrap the multidatabase transaction inside the database using SQL. For instance, in many scenarios, you can get away with using linked tables or linked servers.

If you do have nondatabase RMs involved, or you cannot get away with wrapping up distributed transactions inside the database itself, then you could apply sanctity checks at the end of a transaction so incorrectly committed transactions can be recovered from in a nontransactional form.

But if that is not a suitable solution, then you should try and use the System.Transactions namespaceÐbased transactions. However, always remember to limit the scope of your transaction to as little as possible.

---

11. Actually MSDTC will cause a lot of roundtrips with small payloads, and a lot of roundtrips is what really kills performance. (See ÒQuerying the Database for Multiple Result SetsÓ in Chapter 5.) Even though MSDTC isnÕt querying the database, the same principles applyÑtoo many roundtrips affect

CHAPTER 12

XML and ADO.NET

Lingua franca : A common language used by speakers of different languages

It wouldn't be a stretch of imagination to say that if English is the lingua franca of the world, XML is probably the lingua franca of the computer world. XML by the very nature of it allows you to create a structured and hierarchical representation of your data. Because it is text-based, it can flow freely from one platform to another; and because parsers and standard schema exist on most platforms for its validation, it is the almost de facto choice for transferring data between different applications on different platforms.

Since ADO.NET is a data access architecture, and XML is a popular method to hold and move data, it is quite reasonable to expect that ADO.NET has significant interoperability with XML built in it. As a matter of fact, in the .NET Framework itself, there is the `System.Xml` namespace that is entirely devoted to working with data in XML form.

In Chapter 6, you have already seen some interoperability within the `DataSet` object and XML. You saw how easy it is to convert both a `DataSet` and a `DataTable` to XML and vice versa. You also saw how XSD schemas, which are XML, dictate the structure and data validity of a strongly typed `DataSet`.

This chapter and the next present concepts specific to SQL Server. This chapter carries that discussion forward and introduces you to the ability of leveraging the XML features built in SQL Server. This chapter also looks into the new XML data type in SQL Server 2005 and how you can use that to build better architected applications.

Finally, this chapter looks at the `SqlXml` data provider that has been included as a part of the .NET Framework 2.0, but was also available in a previous version as a part of Microsoft Data Access Components (MDAC) and could be used in .NET 1.1 and before.

But first, let's start with the basics and examine how you can leverage the XML features built in the Microsoft SQL Server database.

SQL Server Native XML Support

Given that XML lends you the ability to hold islands of data without a dedicated database, and given the fact that most of your data will reside in a database somewhere, it's critical that there should be a method to extract XML data out of a relational database such as SQL Server. Also, when programming with such data islands of XML, frequently you'll be manipulating, transmitting, or receiving them using a language such as C# or VB.NET. It is critical that the data access block of such platforms, ADO.NET, should provide you with an easy way to leverage

FOR XMLÕs Optional Arguments

The following optional arguments can be used in conjunction with a FOR XML query:

- ELEMENTS is only applicable to a FOR XML AUTO query, and specifies that the value of each column returned will be represented as an element within the XML document, rather than as an attribute (the default).
- BINARY BASE64 causes any binary data within the XML document to be represented in base-64 encoding. Such data is found in columns of BINARY, VARBINARY, or IMAGE type. The BINARY BASE64 option must be specified in order for FOR XML RAW and FOR XML EXPLICIT queries to retrieve binary data.

  By default, a FOR XML AUTO query handles binary data by creating a reference within the XML document to the location of the binary data. The disadvantage of doing this, however, is that it limits an XML documentÕs portability. When BINARY BASE64 is specified for a FOR XML AUTO query, the generated XML document will contain the binary data.
- XMLDATA generates a schema for the XML document generated by the FOR XML query. This schema is placed at the start of the XML document.

The following SQL is identical to one of our earlier examples, save that it contains the optional XMLDATA argument:

```
SELECT
    Humanresources.Employee.Loginid, Humanresources.Employee.Title,
    Humanresources.Department.Departmentid, Humanresources.Department.Name
FROM
    Humanresources.Department
    INNER JOIN Humanresources.Employee On
    Humanresources.Department.Departmentid = Humanresources.Employee.Departmentid
WHERE Humanresources.Department.Departmentid = 7
FOR XML AUTO, XMLDATA
```

A portion of the XML document generated by this query (including the schema) is as follows:

```
<Schema name="Schema1" xmlns="urn:schemas-microsoft-com:xml-data"
xmlns:dt="urn:schemas-microsoft-com:datatypes">
  <ElementType name="HUMANRESOURCES.EMPLOYEE" content="eltOnly" model="closed"
order="many">
    <element type="HUMANRESOURCES.DEPARTMENT" maxOccurs="*" />
    <AttributeType name="LOGINID" dt:type="string" />
    <AttributeType name="TITLE" dt:type="string" />
    <attribute type="LOGINID" />
    <attribute type="TITLE" />
  </ElementType>
  <ElementType name="HUMANRESOURCES.DEPARTMENT" content="empty" model="closed">
    <AttributeType name="DEPARTMENTID" dt:type="i2" />
    <AttributeType name="NAME" dt:type="string" />
    <attribute type="DEPARTMENTID" />
```

```
<PRODUCTION.PRODUCTPHOTO ProductPhotoID="70" THUMBNAILPHOTO=
"dbobject/PRODUCTION.PRODUCTPHOTO[@ProductPhotoID='70']/@ThumbNailPhoto" />
<PRODUCTION.PRODUCTPHOTO ProductPhotoID="72" THUMBNAILPHOTO=
"dbobject/PRODUCTION.PRODUCTPHOTO[@ProductPhotoID='72']/@ThumbNailPhoto" />
...
```

Referencing XML data is clearly more readable than including binary data. Since the binary data itself is not contained in the XML document, copying the document to another location results in references that can no longer be resolvedÑreferences to binary data are not portable. Microsoft-specific references to binary data should be avoided unless the deployment environment is Microsoft-homogeneous.

Moving on, the `Production.ProductPhoto` table of the `AdventureWorks` database contains a primary key called `ProductPhotoID`, which is also the name of a foreign key in the `Production.ProductProductPhoto` table (mapping table between product and product photo). An example of a `FOR XML AUTO` query that accesses the `ProductProductPhoto` and `ProductPhoto` is shown here:

```
SELECT
    PP.Productphotoid, PP.Thumbnailphoto, PPP.Modifieddate
FROM
    Production.Productphoto PP
INNER JOIN
    Production.Productproductphoto PPP On
    PP.Productphotoid = PPP.Productphotoid
FOR XML AUTO
```

A portion of the output generated by this query is as follows:

```
...
<PP PRODUCTPHOTOID="111" THUMBNAILPHOTO=
"dbobject/PRODUCTION.PRODUCTPHOTO[@ProductPhotoID='111']/@ThumbNailPhoto">
   <PPP MODIFIEDDATE="2001-06-01T00:00:00" />
   <PPP MODIFIEDDATE="2001-06-01T00:00:00" />
   <PPP MODIFIEDDATE="2001-06-01T00:00:00" />
   <PPP MODIFIEDDATE="2001-06-01T00:00:00" />
   <PPP MODIFIEDDATE="2001-06-01T00:00:00" />
</PP>
<PP PRODUCTPHOTOID="113" THUMBNAILPHOTO=
"dbobject/PRODUCTION.PRODUCTPHOTO[@ProductPhotoID='113']/@ThumbNailPhoto">
   <PPP MODIFIEDDATE="2003-06-01T00:00:00" />
   <PPP MODIFIEDDATE="2003-06-01T00:00:00" />
   <PPP MODIFIEDDATE="2003-06-01T00:00:00" />
   <PPP MODIFIEDDATE="2003-06-01T00:00:00" />
</PP>
...
```

This document exploits the relationship between `ProductPhoto` and `ProductProductPhoto`. Each `ProductPhotoID` (element `<PP>`) contains subelements corresponding to itself (element `<PPP>`). Instead of this, you could also specify the `ELEMENTS` option and modify the query as

```
  </GrandParent>
  <GrandParent GrandParentID="2">
    <Son SonName="Darth" />
    <Son SonName="Luke" />
  </GrandParent>
</XML>
```

Entity Encoding

Before delving further into the tantalizing world of FOR XML EXPLICIT directives, let's look at an equally important concept: entity encoding. Entity encoding is the means by which XML special characters can be included in data. What are special characters? The "less-than" character (<) is special because it's used to start each element within an XML document. For example, how would an XML parser handle data of the following form?

```
<CompareThis> MassOfEarth < MassOfJupiter </CompareThis>
```

This is not actually well-formed XML, because the < character inside the element's data leads to a parsing error. In this XML-like snippet, the less-than character indicates the start of each tag, and is also part of the data associated with the element: MassOfEarth < MassOfJupiter. The previous snippet could be made well-formed by using entity encoding to change how the less-than character is represented:

```
<CompareThis> MassOfEarth &lt; MassOfJupiter </CompareThis>
```

< is the entity encoded form of the less-than character, so there is no ambiguity for XML parsers here. The characters deemed as special by XML include &, ', >, <, and ". When you need them, they should by written within an XML document using the alternative representations shown in Table 12-2.

Table 12-2. Various Entity Encoding Representations

Character Name	Character Literal	Entity Encoding Representation
Ampersand	&	&
Apostrophe	'	'
Greater-than	>	>
Less-than	<	<
Quotation mark	"	"

This concept of entity encoding is pertinent to the next section's discussion of the FOR XML EXPLICIT directives.

Directives

The short SQL query you just saw demonstrated how the FOR XML EXPLICIT clause in T-SQL can be used to generate a specific hierarchy. However, real-life situations will present you with loftier goals where you might need to exercise what you will see next: "directives" for the FOR XML EXPLICIT clause. The FOR XML EXPLICIT directives presented in this section include what

```
        '' AS [Daughter!3!OrderByDaughterName!hide],
        '' AS [Daughter!3!DaughterName!element],
        '' AS [Daughter!3!SomeData!element],
        0 AS [GrandKid!4!ChildOfSonID!element],
        '' AS [GrandKid!4!OrderByChildOfSonName!hide],
        '' AS [GrandKid!4!ChildOfSonName!element],
        '' AS [GrandKid!4!GrandKidData!xml]

FROM GrandParent

UNION ALL

-- Generated the Son level of the hierarchy
SELECT 2 AS Tag,
        1 AS Parent,
        0, -- GrandParent.GrandParentID
        G.GrandParentName AS [GrandParent!1!OrderByGrandParentName!hide],
        '', -- GrandParent.Name
        '', -- GrandParent.Diary
        SonID,
        RTRIM(SonName),
        RTRIM(SonName),
        PermanentRecord,
        0, -- Daughter.DaughterID
        '', -- Daughter.OrderByDaughterName
        '', -- Daughter.DaughterName
        '', -- Daughter.SomeData,
        0, -- ChildOfSon.ChildOfOnID,
        '', -- ChildOfSon.OrderByChildOfSonName
        '', -- ChildOfSon.ChildOfSonName
        '' -- ChildOfSon.GrandKidData
FROM GrandParent AS G, Son AS S
WHERE G.GrandParentID = S.GrandParentID

UNION ALL

-- Generate the Daughter level of the hierarchy
-- that is in the same level as the Son's data
SELECT 3 AS Tag,
        1 AS Parent,
        0, -- GrandParent.GrandParentID
        G.GrandParentName AS [GrandParent!1!OrderByGrandParentName!hide],
        '', -- GrandParent.Name
        '', -- GrandParent.Diary
        0, -- Son.SonID
        '', -- Son.SonName (hidden)
        '', -- Son.SonName
```

SQL Server 2005 and FOR XML PATH

So the query used in Listing 12-1 was huge. Specifically, it reads from four tables, and based upon the relationships between those four tables, you had to split the query into four separate SQL statements tied together with a UNION ALL clause.

Actually, itÕs more complex than that. A UNION ALL clause in regular T-SQL (not using FOR XML) would simply require you to make sure that the column sets match, both logically and physically. However, when working with XML data, and FOR XML EXPLICIT queries, you have to logically make sure that all your ÒtagsÓ match, and that you specify the correct ORDER BY clause in the very end.

Obviously, this is a fairly complex routine to achieve something rather straightforward. Luckily for this very reason, SQL Server 2005 has introduced the new FOR XML PATH syntax.

The usage is fairly simple. Consider the following query:

```
SELECT 1 as Tag,
       NULL as Parent,
       G.GrandParentID as [GrandParent!1!GrandParentID],
       NULL       as [Son!2!SonName]
FROM   GrandParent G
WHERE  G.GrandParentID IN (Select GrandParentID from Son)
UNION ALL
SELECT 2 as Tag,
       1 as Parent,
       S.GrandParentID,
       LTRIM(RTRIM(S.SonName))
FROM   GrandParent G, Son S
WHERE  G.GrandParentID = S.GrandParentID
ORDER BY [GrandParent!1!GrandParentID], [Son!2!SonName]
FOR XML EXPLICIT , ROOT('XML')
```

If you wanted to write this same query using the FOR XML PATH syntax, it would look like this:

```
SELECT
    RTRIM (G.GRANDPARENTNAME) [@GRANDPARENTNAME],
    (
        SELECT
            RTRIM (S.SONNAME) [@SONNAME]
        FROM
            DBO.SON   S
        WHERE
            S.GRANDPARENTID = G.GRANDPARENTID
        FOR XML PATH('SON'), TYPE
    )
FROM
    DBO.GRANDPARENT G
WHERE EXISTS
(
    SELECT
        *
```

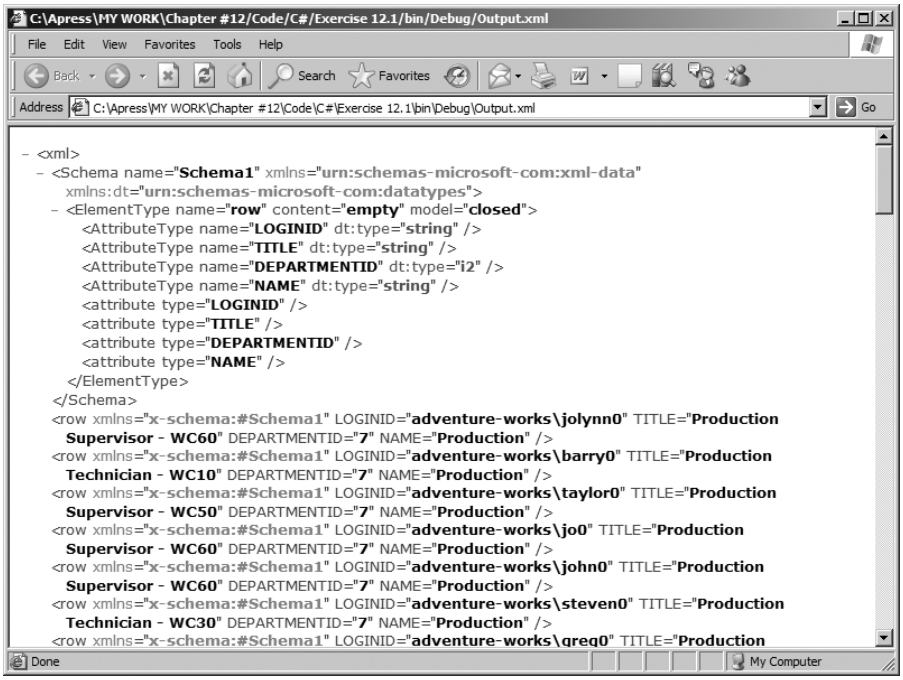


Figure 12-3.FOR XML queryÕs XML output using ADO.NET and XmlReader

As you might expect, you can simply replace the command text for the SqlCommand in ADO.NET, and in similar fashion run either RAW, AUTO, PATH, or EXPLICIT queries. Given the size that such queries can reach, though, itÕs worth bearing in mind that using stored procedures gives you better manageability than storing SQL queries in your code, making the former technique the preferred way of dealing with FOR XML queries in general.

OPENXML

The OPENXML function of SQL ServerÕs T-SQL allows an XML document to be viewed as a row set without the need for any involvement of .NET code. Once opened, this row set can immediately be manipulated using SQL statements such as SELECT, INSERT, UPDATE, and DELETE.

This tying of SQL Server to an XML document results in certain complexities. For example, what happens if the XML document inserted into a SQL Server table contains extra elements or attributes that werenÕt taken into account by the OPENXML command? This is referred to as *overflow*, and it results in the elements and tags in question being *unconsumed*. As you will see, the OPENXML mechanism has the ability to handle unconsumed XML by placing it in a designated column.

The OPENXML function of T-SQL is defined as follows, where parameters surrounded by square brackets ([flags byte[in]]) and clauses surrounded by square brackets ([WITH (SchemaDeclaration | TableName)]) are optional:

```
OPENXML(idoc int [in], rowpattern nvarchar [in], [flags byte [in]])
```

The parameters to OPENXML are defined as follows:

- idoc (input parameter of type int): A document handle referring to the parsed XML document. This document handle is created by calling the sp_xml_preparedocument stored procedure.
- rowpattern (input parameter of type nvarchar): An XPathpattern specifying the node of the XML document to be processed as a row set. For example, the following pattern indicates that the Region node is the level of the XML document to be interpreted: N'/Top/Region'.
- flags (input parameter of type byte): A flag that indicates how the XML node is to be interpreted. 1 indicates that attributes in the document become columns in the row set, while 2 indicates that elements in the document become columns. This flag can also be used to specify that data not consumed by the row set should be placed in an overflow column.

Two forms of the WITH clause can be specified with OPENXML:

- WITH SchemaDeclaration allows an XML-data schema to be specified.
- WITH TableName indicates that the schema associated with a specified table should be used to interpret the XML document specified. This is the simpler of the two variants.

In this section, let's look at how to use OPENXML in an example that requires you to add a stored procedure called RegionInsert to the Regions table in the test database. The relevant scripts containing the RegionInsert stored procedure and others can be found in the associated code download under OpenXMLSP.sql or is reproduced in Listing 12-6 for your convenience.

Listing 12-6. Script to Create Various Stored Procedures Using OPENXML

```
CREATE TABLE Region
(
RegionID INT PRIMARY KEY,
RegionDescription VARCHAR(3000)
)
GO

-- XML Document is of the form
-- <Top>
--   <Region> region elements here </Region>
--   <Region> region elements here </Region>
--   ...
-- </Top>
CREATE PROCEDURE RegionInsert @xmlDoc NVARCHAR(4000) AS
DECLARE @docIndex INT
EXECUTE sp_xml_preparedocument @docIndex OUTPUT, @xmlDoc

-- 1 is ATTRIBUTE-centric mapping
INSERT Region
SELECT RegionID, RegionDescription
```

```
EXECUTE sp_xml_removedocument @docIndex

GO

CREATE PROCEDURE RegionUpdate @xmlDoc NVARCHAR(4000) AS

   DECLARE @docIndex INT

   EXECUTE sp_xml_preparedocument @docIndex OUTPUT, @xmlDoc

   UPDATE Region
      SET Region.RegionDescription = XMLRegion.RegionDescription
   FROM OPENXML(@docIndex, N'/Top/Region',1) WITH Region AS XMLRegion
   WHERE Region.RegionID = XMLRegion.RegionID

   EXECUTE sp_xml_removedocument @docIndex

GO

CREATE PROCEDURE RegionDelete @xmlDoc NVARCHAR(4000) AS
   DECLARE @docIndex INT

   EXECUTE sp_xml_preparedocument @docIndex OUTPUT, @xmlDoc

   DELETE Region
   FROM OPENXML(@docIndex, N'/Top/Region', 1) WITH Region AS XMLRegion
   WHERE Region.RegionID=XMLRegion.RegionID

   EXECUTE sp_xml_removedocument @docIndex
```

This stored procedure contains an `INSERT` statement that uses `OPENXML`. The steps involved in the creation of the `RegionInsert` stored procedure are as follows:

1. Call the system-provided stored procedure, `sp_xml_preparedocument`, passing the XML document to be processed (`@xmlDoc`). This stored procedure parses the XML document and returns a handle (an integer, `@docIndex`) that's used by `OPENXML` to process the parsed document:

   ```
   CREATE PROCEDURE RegionInsert @xmlDoc NVARCHAR(4000) AS
   DECLARE @docIndex INT
   EXECUTE sp_xml_preparedocument @docIndex OUTPUT, @xmlDoc
   ```

2. Call `OPENXML` to create a row set from the XML document. This row set can then be processed by any applicable SQL command. The following `INSERT` statement demonstrates `OPENXML` creating a row set using the schema associated with the `Region` table (`WITH Region`) and then inserting the data into that table:

   ```
   -- 1 is ATTRIBUTE-centric mapping
   INSERT Region
   SELECT RegionID, RegionDescription
   ```

3. Call the system-provided stored procedure, sp_xml_removedocument, in order to clean up the handle to the XML document:

   EXECUTE sp_xml_removedocument @docIndex

You can add this stored procedure to the test database by executing the OpenXMLSP.sql SQL script from the code download. Once that is in place, an example of SQL code (including the XML document with data to insert) that executes the RegionInsert stored procedure is as follows (OpenXMLDemo.sql in the associated code download):

```
DECLARE @newRegions NVARCHAR(2048)

SET @newRegions = N'
<Top>
  <Region RegionID="11" RegionDescription="Uptown" />
  <Region RegionID="22" RegionDescription="Downtown" />
</Top>'

EXEC RegionInsert @newRegions
```

This calls RegionInsert to add two rows to the Region table (one with RegionID 11, and one with RegionID 22). Remember that XML is case sensitive, but SQL Server's SQL is not. When OPENXML was specified (OPENXML(@docIndex, N'/Top/Region', 1) in the RegionInsert stored procedure, the row pattern was /Top/Region. The XML document's elements must match these exactly (<Top> and <Region>). If <TOP> or <top> had been specified as the root element name, then the insertion would have failed, as there would have been a case mismatch.

OPENXML Stored Procedures: Deletes and Updates

Another of the SQL scripts in OpenXMLSP.sql demonstrates OPENXML being used in conjunction with a SQL DELETE operation (in a stored procedure called RegionDelete):

```
CREATE PROCEDURE RegionDelete @xmlDoc NVARCHAR(4000) AS
DECLARE @docIndex INT
EXECUTE sp_xml_preparedocument @docIndex OUTPUT, @xmlDoc

DELETE Region
FROM OPENXML(@docIndex, N'/Top/Region', 1) WITH Region AS XMLRegion
WHERE Region.RegionID = XMLRegion.RegionID

EXECUTE sp_xml_removedocument @docIndex
```

Here, the FROM clause of the DELETE statement uses the OPENXML function to generate a row set named XMLRegion:

```
OPENXML(@docIndex, N'/Top/Region', 1) WITH Region AS XMLRegion
```

OpenXMLSP.sql also includes a stored procedure called RegionUpdate, which uses an XML document to provide the data used to update the Region table:

Next, the application needs to create a parameter for this command's Parameters collection, setting its value to the XML document (strXMLDoc) that will be inserted using the RegionInsert stored procedure:

C#

```
SqlParameter xmlDocParm =
    openXMLCommand.Parameters.Add("@xmlDoc", SqlDbType.NVarChar, 4000);
xmlDocParm.Value = strXMLDoc;
```

VB.NET

```
Dim xmlDocParm As SqlParameter = _
    openXMLCommand.Parameters.Add("@xmlDoc", SqlDbType.NVarChar, 4000)
xmlDocParm.Value = strXMLDoc
```

The ExecuteNonQuery method of the openXMLCommand SqlCommand instance can now be called to insert the data. (ExecuteNonQuery is a good choice here because RegionInsert only inserts data and doesn't return the results of a query.)

C#

```
sqlConnection.Open();
openXMLCommand.ExecuteNonQuery();
```

VB.NET

```
sqlConnection.Open()
openXMLCommand.ExecuteNonQuery()
```

The next stored procedure to demonstrate is RegionUpdate. To facilitate this, the data associated with the parameter (the XML document) is tweaked by changing each instance of the word "town" to "state" (so "Uptown" becomes "Upstate", and "Downtown" becomes "Downstate"), courtesy of the String class's Replace method. Once the data is tweaked, the command's text is set to RegionUpdate and the command is executed using ExecuteNonQuery:

C#

```
xmlDocParm.Value = strXMLDoc.Replace("Town", "state");
openXMLCommand.CommandText = "RegionUpdate";
openXMLCommand.ExecuteNonQuery();
```

VB.NET

```
xmlDocParm.Value = strXMLDoc.Replace("Town", "state")
openXMLCommand.CommandText = "RegionUpdate"
```

The remainder of Exercise 12.2 sets the command's text to the stored procedure that handles deletion, RegionDelete. Once this is set, ExecuteNonQuery can work its magic again:

C#

```
openXMLCommand.CommandText = "RegionDelete";
openXMLCommand.ExecuteNonQuery();

sqlConn.Close();
```

VB.NET

```
openXMLCommand.CommandText = "RegionDelete"
openXMLCommand.ExecuteNonQuery()

sqlConn.Close()
```

The elegance of this technique is that .NET is blissfully unaware of how the XML document it passes to the stored procedure is eventually written to SQL Server. The stored procedure calls ultimately use OPENXML, but ADO.NET neither knows nor cares what's going on under the covers.

The XML Data Type: SQL Server 2005 Only

Of all the techniques you have seen so far that help you work with XML data, the common theme has been an attempt to bridge the relational table structures with a hierarchical XML structure. But in all these cases, the XML was either returned or accepted as a string that simply holds XML. That is probably the reason why, in spite of all the rich XML features in Microsoft SQL Server 2000 and 2005, a lot of developers choose to store their data directly as XML stored in varchar data type columns.

That approach has numerous downsides. For one, the data is not validated in any manner. Potentially someone could store data that does not validate any specified schema, because the database does not enforce those rules. Another problem is the inability to query against specific fields in that data. Probably the best you can do is string-like searching using the % characters. Yet another disadvantage that arises from the same fact that everything is stored as a string is that finding the right node to update might be difficult. The situation becomes more complex considering that you might encounter rows that do not follow a set schema so your program logic has to consider that.

Keeping all that in mind, SQL Server 2005 includes a first-class data type, like int or varchar, that allows in-place querying, processing, and storage of XML documents. This is the new xml data type.

Creating such a table can be achieved using the following script.

```
Create Table MyXmlTable
(
    MyXmlTableID INT IDENTITY PRIMARY KEY,
    MyXmlData XML
```

SQLXML and ADO.NET

Having said all that, what reasons could you have for using SQLXML from .NET code—especially when ADO.NET has so many XML features? Most of the time, ADO.NET is the way to go, but you'll want to consider using SQLXML managed classes in the following conditions:

- You're migrating to the .NET Framework from a previous version that used SQLXML in unmanaged code. In this case, it will be easier simply to migrate the SQLXML code to .NET code, rather than rewrite everything in ADO.NET.
- You want to do client-side XML formatting, which is currently not available in ADO.NET.
- You want to represent multidimensional table relations, as this cannot be done in ADO.NET.

Let's now look at the architecture of various SQLXML classes, and begin to see how you can use them.

The SQLXML Object Model

The SQLXML managed classes themselves are implemented in the Microsoft.Data.SqlXml.dll assembly (typically located in the C:\Program Files\SQLXML 4.0\bin folder). If you're using Visual Studio .NET, you'll need to add a reference to this assembly before you can use the three classes it contains:

- SqlXmlCommand
- SqlXmlParameter
- SqlXmlAdapter

The SqlXmlCommand Object

A SqlXmlCommand object is similar to an ADO.NET SqlCommand object, but it's used for working with SQLXML in SQL server. As you will see, however, the SqlXmlCommand class is slightly more useful in certain scenarios.

The first is that it can be used to make the SQL client do more work, rather than pushing the load onto the SQL Server. When you perform a FOR XML query using the SqlCommand class, the work of transforming the selected row set into XML format is done in the SQL Server process on the database. When you use a SqlXmlCommand object and FOR XML, you have the option of saying that you want this process to occur in the data layer, and hence reduce the load on your database.

Furthermore, the SqlXmlCommand class allows you to write the result to a new or existing stream instance (a class derived from the System.IO.Stream class), an option that's not available in ADO.NET. Table 12-4 looks over a few commonly used SqlXmlCommand properties while Table 12-5 goes over the various methods available.

```
    cmd.ClientSideXml = true;
    cmd.RootTag = "Person";

    string strResult;
    try
    {
        Stream strm = cmd.ExecuteStream();
        strm.Position = 0;
        using (StreamReader sr = new StreamReader(strm))
        {
            Console.WriteLine(sr.ReadToEnd());
        }
    }
    catch (SqlXmlException e)
    {
        //in case of an error, this prints error returned.
        e.ErrorStream.Position = 0;
        strResult = new StreamReader(e.ErrorStream).ReadToEnd();
        System.Console.WriteLine(strResult);
    }
}
```

Listing 12-8. Using Templatized Queries in Visual Basic .NET

```
Sub Main()
    Dim xmlQuery As FileStream = New FileStream("command.xml", FileMode.Open)
    Dim cmd As SqlXmlCommand = New SqlXmlCommand(connectionString)
    cmd.CommandStream = xmlQuery
    cmd.CommandType = SqlXmlCommandType.Template

    Dim parm As SqlXmlParameter
    parm = cmd.CreateParameter()
    parm.Name = "@LastName"
    parm.Value = "Achong"

    cmd.ClientSideXml = True
    cmd.RootTag = "Person"

    Dim strResult As String
    Try
        Dim strm As Stream = cmd.ExecuteStream()
        strm.Position = 0
        Using sr As StreamReader = New StreamReader(strm)
            Console.WriteLine(sr.ReadToEnd())
        End Using
    Catch e As SqlXmlException
        'in case of an error, this prints error returned.
```

```
        strResult = New StreamReader(e.ErrorStream).ReadToEnd()
        System.Console.WriteLine(strResult)
    End Try
End Sub
```

As you can see from this code, the method of declaring the command has changed. First, you have to stream the command in from a stream (this could be any stream). Second, you have to specify using the `CommandType` property that this is a templatized query.

Next, you have to arrange to pass the parameter to the query, but unless you give it a name, it won't work. In this case, the parameter was called `LastName` in the template XML file, so that is what is used here. Again, you need to set the `Value` property, execute the query just as before, and write the result to console output. The result when this is run is shown here:

```
<ROOT xmlns:sql="urn:schemas-microsoft-com:xml-sql">
  <Person.Contact FirstName="Gustavo" LastName="Achong"/>
</ROOT>
```

This is almost the same result as the last query, but this time it's wrapped in a `<ROOT>` element that's added automatically when you're working with templates. If you wanted to get back the XML without this element, you'd do better to load the result into an `XmlReader` instance. This would require you to modify the reading portion of the previous code and replace it with the following:

C#

```
XmlTextReader rdr ;
rdr = cmd.ExecuteXmlReader() ;
rdr.MoveToContent() ;
Console.WriteLine(rdr.ReadInnerXml()) ;
rdr.Close() ;
```

VB.NET

```
Dim rdr As XmlTextReader
rdr = cmd.ExecuteXmlReader()
rdr.MoveToContent()
Console.WriteLine(rdr.ReadInnerXml())
rdr.Close()
```

If you make these changes and run the code, the following will once again be output:

```
  <Person.Contact FirstName="Gustavo" LastName="Achong"/>
```

Updating with an UpdateGram

At some stage, you're going to want to update data that has been modified on the client. In SQLXML, this is typically performed using an `UpdateGram` when you create the update XML document yourself, but the way the `SqlXmlAdapter` object has been implemented means it will use the DiffGram format, as featured in ADO.NET.

CHAPTER 13

The CLR in SQL Server

T-SQL is the default language supported by SQL Server. It has well-defined syntax for database features such as queries, triggers, and stored procedures; those are various tools in your arsenal that allow you to interact, modify, or manage the data inside SQL Server. While T-SQL is a rather powerful language, its nature is quite different from a typical modern-day programming language such as C# or VB.NET. For instance, when using C# or VB.NET, it is common practice to architect your solution in an object-oriented way using the rich inheritance features of such languages to express your architecture. T-SQL, on the other hand, does not offer such facilities.

However, everything has a purpose and a reason. T-SQL is the right tool for interacting with relational data. For instance, writing a join query in C# for every single row in a three table query is something that T-SQL can do much better than C# or VB.NET can. This is natural to expect since database engines are able to optimize and cache a query, its plan, and its results in many stages whereas C# or VB.NET does not have any of that.

But when working in the database, you may frequently run into situations where the unwieldiness of T-SQL in doing tasks, such as mathematical calculations or recursive operations, makes it less than a pleasure to deal with.

Note There is a new feature in SQL Server 2005 called Common Table Expressions (CTE) that does allow you to write recursive T-SQL queries on tabular data. It has a configurable recursion depth with a default value of 100 levels deep. The problem arises when you wish to write a T-SQL stored procedure that calls itself to perform an operation such as calculate factorials. When the T-SQL stored procedure calls itself, the call depth of one stored procedure calling another cannot exceed 32. In other words, it would not be possible for a T-SQL stored procedure that calculates factorials using recursion to calculate the factorial of 33. CLR, on the other hand, would have no problem doing this.

Usually, the common approach in such cases would be to extract the relevant, unprocessed data from SQL Server as result sets and process upon the data in the data layer or business layer using a higher level programming language. This, however, pays the penalty of a network roundtrip and possibly extracting more data than required.

Starting with SQL Server 2005, you have the ability to author .NET code that lives inside SQL Server. In other words, right inside the database you have the same ability to use the rich features of languages such as C# or VB.NET for specific tasks that are better suited to them.

As described in the Chapter 5 sidebar titled "The CLR Is Inside Out," another major difference between the CLR on a Windows machine and the CLR inside SQL Server is the bootstrap mechanism used to load the CLR. An application that wishes to use the CLR uses ICorRuntimeHost, or CorBindToRuntimeEx, which then in turn calls MSCOREE.DLL that ends up loading the runtime.

Instead, SQL Server 2005 does not load the CLR unless it needs to. This is because SQL Server follows the principle of conserving memory and lazy loading any resource that it might need. By not loading the CLR unless it needs to, it saves a few MB of memory that the CLR would have otherwise occupied. Thus, if you have one minor piece of your code that uses the CLR, you are loading the CLR and, in turn, affecting any other operation on that particular computer. Even when it does load the CLR, it does so using ICLRRuntimeHost, which is a new interface introduced with .NET 2.0. Next it calls, ICLRRuntimeHost::GetHostManager, which essentially means that the CLR now delegates operations like resource locking/thread management, etc. to the SQL Server runtime instead.

Having delegated such responsibilities to the host (SQL Server) instead brings up interesting challenges. Essentially, what this means is that individual operations running inside SQL Server can now decide to be rogue and potentially be a security threat, or bring down the server. Therefore, it becomes critical that when taking the freedom of hosting the CLR and its various operations, the SQL Server application also takes on the responsibility of doing it correctly in such a manner that individual applications should not inadvertently or surreptitiously harm the server in any manner.

SQL Server enforces this by giving you a granular level of control on the set of operations your .NET code can perform inside SQL Server. This mechanism is built on top of Code Access Security (CAS) that is a part of the CLR. The developers at Microsoft took a long and hard look at every single class in the .NET Framework, and classified them into three categories that you must specify your code to fall within. In other words, you need to tell SQL Server that your code falls into one of three categories based upon the specific operations it intends to do. If your code tries doing something other than what you had initially specified, SQL Server will block it from doing so (you will see this as an example later in this chapter).

The three levels of control are as follows:

- SAFE: This is the default level and most restrictive. This means that your code does not need any external resources and the operation is wholly controlled inside SQL Server. Safe code can access data from the local SQL Server databases or perform computations and business logic that does not involve accessing resources outside the local databases. A good example of this could be factorial calculation. Factorial calculation only needs an input of type integer, and then it returns another integer. To calculate a factorial, you do not need to open a file on the disk.
- EXTERNAL_ACCESS: This level signifies that certain external resources such as files, networks, web services, environmental variables, and the registry are accessible. Thus, if your code intends to write out some results to a file on the disk, you would need to register that code inside SQL Server under the EXTERNAL_ACCESS security category.
- UNSAFE: This level, which you should try very hard to avoid, specifies that your code is allowed to do anything. In other words, you are requesting to be free of any granular-level control, and thus giving it the same permissions as an extended stored procedure. Even though you get the same rights and permissions as an extended stored procedure, CLR still gives you certain benefits in comparison, but there could be a hole in your logic that a hacker could abuse and gain access to crucial parts of your system code inside SQL Server.

Software Requirements to Run the Examples in This Chapter

As usual, you need two things: a data source and a programming environment, usually Visual Studio 2005.

Your data source for the examples in this chapter will be a database called Test running on the local instance of SQL Server 2005. This database can be easily set up by running the following command:

```
Create database test
```

Since the content in this chapter is specific to SQL Server 2005, you cannot run the examples presented here in an Oracle database[2] or SQL Server 2000 database, etc.

Code written for SQLCLR can be authored in Visual Studio 2005 easily using a SQL Server project template. Unfortunately, this project template is available only in Visual Studio 2005 Professional and Team System Editions. However, it is almost just as easy to author the code presented in this chapter in any other edition of Visual Studio 2005, such as Standard. Let's begin by looking at an example of a SQLCLR UDF that is written without leveraging the Visual Studio 2005 SQL Server project.

Handwritten UDF

In order for you to run any CLR code inside SQL Server, you will have to compile it as an assembly and register that assembly inside SQL Server using the CREATE ASSEMBLY statement. Thus, if you are able to create an assembly similar to what a SQL Server project would produce, SQL Server would be able to register it within itself.

Therefore, in order to create a SQLCLR UDF without leveraging a SQL Server project, all you really need to do is create a class library and follow certain basic rules in the class that holds the logic of the UDF.

As it turns out, these "rules" aren't that numerous or too complex to follow. You can download the code for this example from the associated code download for this chapter under HandWrittenUDF (see the Downloads section of the Apress website at http://www.apress.com), or you could follow the steps mentioned here to create such a project:

1. So the aim of the game is to be able to create a UDF that returns an integer random number. Begin by creating a class library project, call it HandWrittenUDF.

2. Rename the Class1.cs/Class1.vb file to UserDefinedFunctions.cs/UserDefinedFunctions.vb. Verify that Visual Studio changed the class name contained inside the file to UserDefinedFunctions. Also, verify that the scope of this class is public.

3. Wrap the UserDefinedFunctions class inside the HandWrittenUDF namespace and add a using/Imports statement bringing the relevant namespaces, including Microsoft.SqlServer.Server, in scope as shown in the following code:

---

2. It is quite possible that other major database vendors, such as Oracle and IBM, could eventually introduce their own CLR integration; chances are their implementations would be different from SQL

SQL Server Project UDF

In the last section you saw how to create a UDF without using the built-in SQL Server project in Visual Studio 2005. Since the project was just a simple class library, the rest of the SQLCLR examples in this chapter will use a SQL Server project instead.

Because the SQL Server project is available only within Visual Studio's Professional and Team System Editions, I will provide you with the relevant SQL scripts if, in case, you decide to implement the code as class libraries in Visual Studio's Standard or Express Editions instead. These SQL scripts are also useful in an enterprise development environment where "right-click deploy" for the many procedures you will write will probably not be a viable option.

Even then SQL Server projects offer significant advantages, especially when it comes to debugging SQLCLR code. In the next section, I will demonstrate various ways to debug SQLCLR code, and you can reuse those concepts throughout all the examples presented in this chapter.

But before I can show you the various methods of debugging handwritten SQLCLR code, or code written with the help of a SQL Server project, let's quickly walk through creating a UDF that generates random numbers, using a SQL Server project instead. You can download this code from the SqlServerUDF project, or you can create it yourself by following these steps:

1. Begin by creating a Visual Studio SQL Server project in the language of your choice. This can be seen for VB.NET in Figure 13-2. Name the project SqlServerUDF.

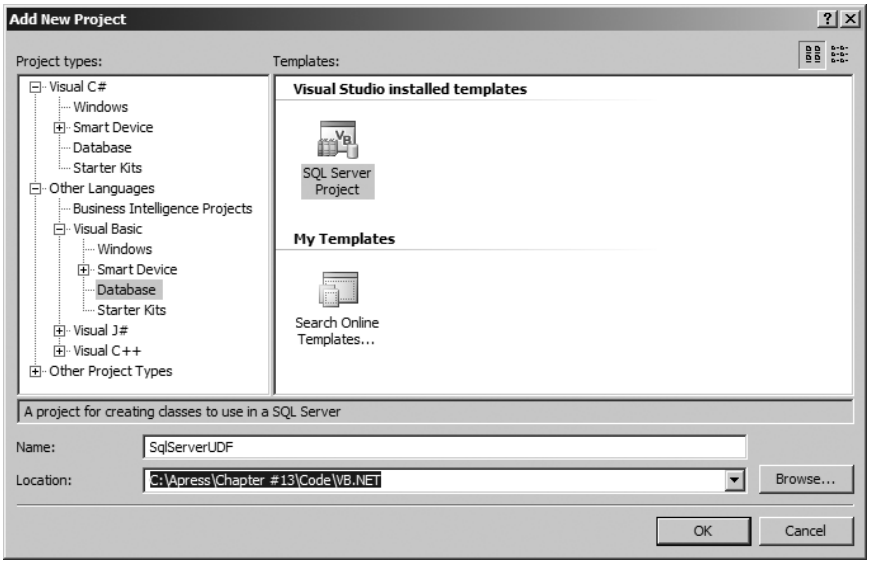


Figure 13-2. Creating a new SQL Server project

2. If you already have a database reference set up within Visual Studio, Visual Studio will prompt you to either pick the existing reference or create a new one. This dialog box can be seen in Figure 13-3.

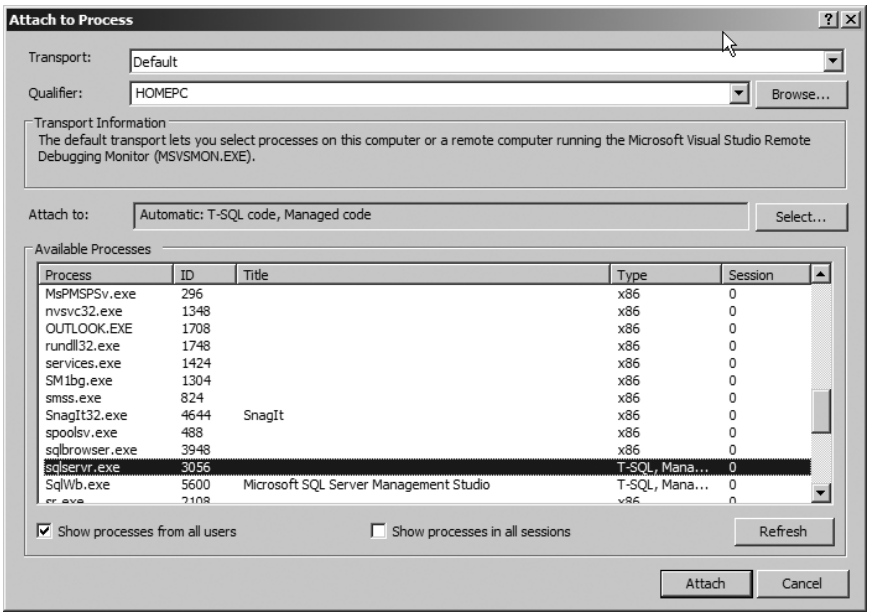
If you continue debugging through the UDF, you'll notice that the Debug Output window will show you the following output:

```
'Managed': Loaded 'SqlServerUDF', No symbols loaded.
Column1
-----------
1275838787
No rows affected.
(1 row(s) returned)
Finished running sp_executesql.
The thread 'homepc [56]' (0x1614) has exited with code 0 (0x0).
The program '[3056] sqlservr.exe: HOMEPC;
.Net SqlClient Data Provider;4876' has exited with code 259 (0x103).
The program '[3056] [SQL] homepc: homepc' has exited with code 0 (0x0).
```

Thus, the output of your UDF is shown as a part of the debug output. You can use a similar mechanism to debug other kinds of SQLCLR objects such as stored procedures, triggers, etc.

While the SQL Server project makes it easier for you to debug your code in a very simple manner, debugging your HandWrittenUDFs isn't very difficult either.

The basic idea remains the same: you need to attach with the sqlservr.exe process and then somehow execute the UDF. Since the project will not do this for you automatically, before running the SQL script you need to do this manually. This can be done by selecting Debug Attach to Process. In the dialog box that pops up, choose the sqlservr.exe process (you may need to check the "Show processes from all users" checkbox to see that process). With the sqlservr.exe process selected, click the Attach button to enter debug mode. This can be seen in Figure 13-11.



SQL Server project

Now set a breakpoint in the code, and execute the following command from any application including SQL Server Management Studio:

```
Select dbo.GetRandomNumber()
```

You'll notice that the breakpoint gets hit and then you can debug much like the SQL Server project.

So, truly, whatever you can do using a SQL Server project is achievable through a simple class library project as well. Therefore, in the rest of the examples that follow in this chapter, I'll be taking advantage of the SQL Server project to demonstrate the code. You can, however, reuse the same principles presented here and the relevant SQL scripts to work with a simple class library if you wish or need to.

Writing a TVF: Table-Valued Function

A Table-Valued Function (TVF) is a UDF that returns a table instead of a scalar value. The last example you saw returned a singular scalar value as an integer. Depending on your needs, you may be presented with a case where your UDFs would need to return a table instead. In other words, your result contains tabular data with rows and columns. An example could be a TVF that returns days in a week as seven rows, and two columns, namely DayNumber and DayName.

Thus, if you wish to return the results as a table, or in the form of rows and columns, your UDF needs to have two facilities:

- It needs to be able to return a number of rows. Returning rows, as it turns out, is easy—all you have to do is return an instance of a class that implements the IEnumerable interface, instead of the scalar value, and the infrastructure will understand that you are trying to send back rows.
- It needs to have the ability to split each row into a fixed number of columns of a predefined format. Sending columns requires you to add another method, which takes the responsibility of splitting the contents of each enumerated item (row) in the IEnumerable object into columns. But how will the infrastructure know "which method is responsible for splitting a row into columns?"

In addition to implementing these two methods, you also need a way to tell the framework that the rows returned by the IEnumerable object will be split into columns by a method of a specified name. You need to specify this method name using the FillRowMethodName property of the Microsoft.SqlServer.Server.SqlFunctionAttribute attribute, which is defined on the method that works as the entry point for the TVF.

This way, SQL Server knows that it needs to call the static/shared method marked with the SqlFunctionAttribute, which acts as the stored procedure, and then it needs to split each returned row into columns using a method specified as the FillRowMethodName property on the SqlFunctionAttribute instance.

Enough of theory; let's look at this using an example. Now that you have already seen how to create either a SQL Server or a class library project, you can follow the next steps using the same concepts. The code for this example can be found in the associated code download in the SqlServerTVF project.

C#

```
[Microsoft.SqlServer.Server.SqlFunction(FillRowMethodName = "FillRow",
    TableDefinition="charpart nchar(1), intpart int")]
public static IEnumerable NameToAscii(string InputName)
{
    return new NameSplitter(InputName.ToCharArray());
}
```

VB.NET

```
<Microsoft.SqlServer.Server.SqlFunction(FillRowMethodName:="FillRow", _
    TableDefinition:="charpart nchar(1), intpart int")> _
Public Shared Function NameToAscii(ByVal InputName As String) As IEnumerable
    Return New NameSplitter(InputName.ToCharArray())
End Function
```

Specifically in C#, you could use the yield keyword to simplify your code. The yield keyword gives you a convenient alternative to implementing an entire class that implements IEnumerable. Thus, by using the yield keyword, you no longer have to implement the NameSplitter class. This is, however, specific to C#. The usage is shown here:

C#

```
[Microsoft.SqlServer.Server.SqlFunction(FillRowMethodName = "FillRow",
    TableDefinition="charpart nchar(1), intpart int")]
public static IEnumerable NameToAscii(string InputName)
{
    foreach (char c in InputName)
    {
        yield return new NameRow(c, (int)c);
    }
}
```

Now, because VB.NET does not have the yield keyword, you'll need to implement a class to achieve the same functionality. The NameSplitter class is the class that implements IEnumerator and IEnumerable. This class can be seen in Listings 13-7 and 13-8.

Listing 13-7. Implementing the NameSplitter Class in C#

```
public class NameSplitter : IEnumerable, IEnumerator
{
    private Int32 idx = - 1;
    private Char[] _inputName;

    public NameSplitter(Char[] InputName)
    {
        _inputName = InputName;
```

```
Namespace SqlServerTVF
    Partial Public Class UserDefinedFunctions
        <Microsoft.SqlServer.Server.SqlFunction(FillRowMethodName:="FillRow", _
            TableDefinition:="charpart nchar(1), intpart int")> _
        Public Shared Function NameToAscii(ByVal InputName As String) _
         As IEnumerable
            Return New NameSplitter(InputName.ToCharArray())
        End Function

        Public Shared Sub FillRow(ByVal row As Object, _
            <Out()> ByRef charpart As Char, <Out()> ByRef intpart As Integer)
            ' break the row into its columnar parts.
            charpart = CType(row, NameRow).CharPart
            intpart = CType(row, NameRow).IntPart
        End Sub
    End Class


    Public Class NameSplitter
        Implements IEnumerable, IEnumerator

        Private idx As Int32 = -1
        Private _inputName As Char()
        Private _current As Char

        Public Sub New(ByVal InputName As Char())
            _inputName = InputName
        End Sub

        Public Function GetEnumerator() As System.Collections.IEnumerator _
            Implements System.Collections.IEnumerable.GetEnumerator
            Return New NameSplitter(_inputName)
        End Function

        Public ReadOnly Property Current() As Object _
            Implements System.Collections.IEnumerator.Current
            Get
                If (idx > -1) Then
                    Dim c As Char = _inputName(idx)
                    Return New NameRow(c, Asc(c))
                Else
                    Return -1
                End If
            End Get
        End Property

        Public Function MoveNext() As Boolean _
```

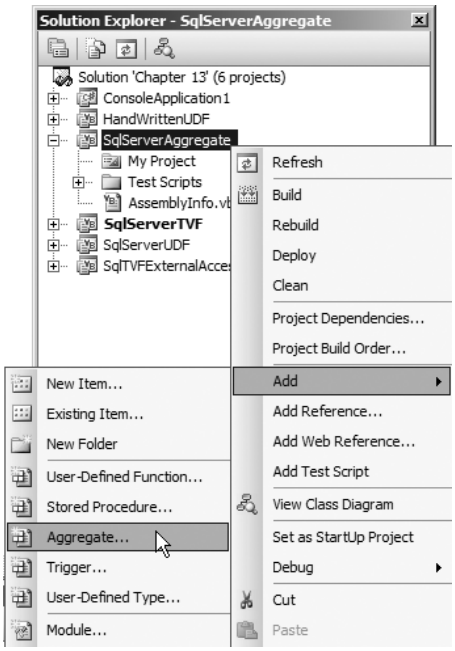


Figure 13-15. Adding a new SQLCLR aggregate to your project

3. As you will see in the skeleton of the code already created for you, there are four methods that exist:

   Init : This method is where you should perform the initialization.

   Accumulate This method is where you will perform the aggregate functionality, in this case the actual concatenation.

   Merge This is where you would write code if, in case, the aggregate is used with another aggregate.

   Terminate: This is where you would calculate the final value and send it back.

You can now go ahead and fill out the code for these four methods as per Listings 13-17 and 13-18.

Listing 13-17. The Skeleton Code for the SQLCLR Aggregate in C#

```
private StringBuilder sb;
public void Init()
{
    sb = new StringBuilder();
}
```

4. Now there is still one thing left to do before this becomes a valid, workable, aggregate method. If you note, the default value of the Format property, of the SqlUserDefinedAggregateAttribute attribute, is Format.Native . This won't work. You can only use Format.Native for data types that SQL Server has the ability to read and write to a byte stream directly. In other words, for Format.Native , you do not have to implement the serialization and deserialization process. The data types that can be used with Format.Native are System.Byte, System.SByte, System.Int16 , System.UInt16, System.Int32 , System.UInt32, System.Int64 , System.UInt64, System.IntPtr , and System.UIntPtr .

   Our class, however, is returning a string, SqlString to be precise. Now, for you to be able to use Format.Native , the type has to be a blittable type, that is, no reference-type members or other non-value-based content. Thus, you will have to change the format from Native to UserDefined. In addition to changing the format from Native to UserDefined, you'll also have to implement an interface called IBinarySerialize where you'll be required to create two methods: Read and Write , to read and write from a BinaryReader and BinaryWriter , respectively.

   After having made these changes, your code should look like as shown in Listings 13-19 and 13-20.

   Listing 13-19. The Full Code for the Concatenator Aggregate in C#

```
[Serializable]
[Microsoft.SqlServer.Server.SqlUserDefinedAggregate(Format.UserDefined,
    MaxByteSize=8000)]
public struct Concatenator : IBinarySerialize
{
    private StringBuilder sb;
    public void Init()
    {
        sb = new StringBuilder();
    }

    public void Accumulate(SqlString Value)
    {
        sb.Append(Value);
        sb.Append(",");
    }

    public void Merge(Concatenator Group)
    {
        Accumulate(Group.ToString());
    }

    public override string ToString()
    {
        return sb.ToString();
```

Well, you could certainly create a loop-back connection,[3] but if you are connecting from within the same database then you are already in a connection. Wouldn't it be nice to be able to get a hold of the same connection that the stored procedure was called upon, and simply use that connection to execute the commands?

It's for this reason that Microsoft created the concept of a context connection. The context connection can be created using the following connection string:

```
"context connection = true"
```

Thus, the following instance of SqlConnection holds the context connection—in other words, the same connection the stored procedure was called upon:

C#

```
SqlConnection contextConnection = new SqlConnection("context connection = true");
```

VB.NET

```
Dim contextConnection as SqlConnection = _
    New SqlConnection("context connection = true")
```

Using a context connection versus a loop-back connection has several advantages:

- By doing so, you neatly skip over the TDS protocol layer and the TCP/IP or Named Pipes protocol layer, and directly get a hold of the connection inside the database.
- You do not need to reauthenticate.
- You have the ability to latch on to a currently running transaction easily. Any external non–context connections created by specifying a full ADO.NET connection string would automatically try to enlist themselves in the active running transaction and thus be promoted to MSDTC (the exception to this rule being loop-back connections into the same SQL Server 2005 database). Such connections cannot enlist themselves in the same transaction and must use enlist=false in their connection strings to work.

However, there are certain restrictions placed on a context connection as well. These are listed here:

- Within a running instance of a SQLCLR method (procedure, TVF, UDF, aggregate etc.), you can have at most only one SqlConnection object with its State = ConnectionState.Open that was opened within the same execution scope. The execution scope begins when the code is first called, typically the first line in the method, and ends when the method completely exits, typically a return statement or the last executable line of the method. In a case where the SQLCLR method calls T-SQL from within, which in turn calls another SQLCLR method, that causes a nested execution scope in the called SQLCLR method that is different from the original execution scope in the calling SQLCLR method. In that scenario,

3. Such a loop-back connection cannot be enlisted in the active transaction in SQL Server 2005. More on

the nested execution scope and calling execution scope can both have one concurrently open context connection each.

- MARS does not work with context connections. (MARS is covered in depth in Chapter 11.)
- SqlBulkCopy cannot work with a context connection.
- Update batching does not work with a context connection.
- SqlNotificationRequest cannot be used with commands that execute on a context connection.
- Canceling commands is not supported. SqlCommand.Cancel will be ignored.
- No other connection string keywords can be specified with context connection = true.

In order to facilitate your working with the context inside SQL Server, ADO.NET provides an object called SqlContext, which can be found in the Microsoft.SqlServer.Server namespace.

The SqlContext object provides you with a Pipe property of SqlPipe data type, which is your primary means of communication with the caller application. For instance, you can do the equivalent of the T-SQL PRINT statement using the SqlContext.Pipe.Send method. If, instead, this were called from an ADO.NET application, you could intercept that message using the SqlConnection.InfoMessage event in the client application.

Similarly, in order to execute a set-oriented T-SQL command and send the results back to a data reader, you could use either the SqlPipe's ExecuteAndSend method as a one-shot solution, or you could use SendResultsStart to begin sending results, SendResultsRow to send an individual row, followed by SendResultsEnd to complete the result set.

Let's examine what you have learned in a quick example that demonstrates both of these cases. You can create the example by following the steps here, or you can find it in the associated code download under the SqlServerStoredProc project:

1. In this example, let's write up two SQLCLR stored procedures. Both of the stored procedures get a hold of the context connection. The first stored procedure, GetConcatenatedNames, executes the SqlServerAggregate aggregate function you have written previously using the following T-SQL command:

   ```
   Select
       dbo.Concatenator(PersonName) from Person
   Where PersonRole = @Role
   Group By PersonRole
   ```

   The second stored procedure will return a result set using the following T-SQL command:

   ```
   Select PersonName from Person where PersonRole = @Role
   ```

2. So go ahead and create a new SQL Server project, call it SqlServerStoredProc, and add a new stored procedure to it—call that file GetNames.cs/GetNames.vb

3. Create the first stored procedure, GetConcatenatedNames, as shown in Listings 13-21 and 13-22.

```
            New SqlDataRecord(New SqlMetaData("PersonName", _
            SqlDbType.NVarChar, 200))

        ' start a  new result set
        SqlContext.Pipe.SendResultsStart(rec)

        ' send rows
        Dim rdr As SqlDataReader = contextCommand.ExecuteReader()
        While rdr.Read()
            rec.SetString(0, rdr.GetString(0))
            SqlContext.Pipe.SendResultsRow(rec)
        End While

        ' complete the result set
        SqlContext.Pipe.SendResultsEnd()
    End Using
End Sub
```

5. Compile, build, and deploy the stored procedures. If you wish to deploy the stored procedure using a script, you can use the script shown here:

```
Create Assembly SqlServerStoredProc
from
'C:\SqlServerStoredProc\SqlServerStoredProc.dll'
GO

Create Procedure GetConcatenatedNames
(
    @Role NVARCHAR(4000)
)
As
External Name
SqlServerStoredProc.[SqlServerStoredProc.StoredProcedures].
GetConcatenatedNames
Go

Create Procedure GetNames
(
    @Role NVARCHAR(4000)
)
As
External Name
SqlServerStoredProc.[SqlServerStoredProc.StoredProcedures].GetNames
Go
```

6. Now if you run the T-SQL here,

```
exec dbo.GetConcatenatedNames 'Reviewer'
```

Thus, you are trying to insert one row through T-SQL, and one through SQLCLR code. As expected, the one transaction that wraps both these inserts is rolled back completely and the table is entirely untouched.

Now go ahead and comment out the SQLCLR statement that is forcing a rollback of the transaction. Rebuild and redeploy your stored procedure. This time, run your stored procedure using the following T-SQL script. Note that this time a ROLLBACK is being issued from T-SQL instead:

```
BEGIN TRANSACTION

Insert into Person (PersonName, PersonRole)
Values ('Linda', 'Cover Editor')

EXEC   [dbo].[InsertName]
                              @personName = N'Emily',
                              @personRole = N'Project Manager'
ROLLBACK

Select * from Person
```

The results of this script's execution are shown here:

```
(1 row(s) affected)
PersonID    PersonName  PersonRole
-----------------------------------------------
1           Sahil Malik  Author
2           Erick Sgarbi Reviewer
3           Frans Bouma  Reviewer
4           Jon Hassell  Lead Editor

(4 row(s) affected)
```

Thus, as you can see, the first insert succeeded, but the ensuing rollback rolled back both the SQLCLR and T-SQL changes. You are still left with only four rows inside the table.

Finally, change the ROLLBACK to COMMIT, and run the following T-SQL script:

```
BEGIN TRANSACTION

Insert into Person (PersonName, PersonRole)
Values ('Linda', 'Cover Editor')

EXEC   [dbo].[InsertName]
                              @personName = N'Emily',
                              @personRole = N'Project Manager'
COMMIT

Select * from Person
```

Using Non-Context Connections Inside SQLCLR

The code shown previously in Listings 13-25 and 13-26 uses the following connection string:

```
context connection = true
```

But what would happen if you specified a connection string connecting to an external database server (win2k3-smalik) as shown here?

```
Server=win2k3-smalik;Database=Test;Integrated Security=SSPI
```

Well, the best way to find out is by modifying the code to use this connection string. Go ahead and modify the connection string for the GetConcatenatedNames stored procedure as shown here, so the code for GetConcatenatedNames looks like this:

C#

```
using (SqlConnection contextConnection =
    new SqlConnection("Server=win2k3-smalik;Database=Test;Integrated Security=SSPI"))
{
    SqlCommand contextCommand =
        new SqlCommand(
        "Select dbo.Concatenator(PersonName) from Person " +
        "where PersonRole = @Role Group By PersonRole", contextConnection);

    contextCommand.Parameters.AddWithValue("@Role", role);
    contextConnection.Open();

    SqlContext.Pipe.ExecuteAndSend(contextCommand);
}
```

VB.NET

```
Using contextConnection As SqlConnection = _
    New SqlConnection("Server=(local);Database=Test;Integrated Security=SSPI")

    Dim contextCommand As SqlCommand = _
        New SqlCommand( _
        "Select dbo.Concatenator(PersonName) from Person " & _
        "where PersonRole = @Role Group By PersonRole", contextConnection)

    contextCommand.Parameters.AddWithValue("@Role", role)
    contextConnection.Open()

    SqlContext.Pipe.ExecuteAndSend(contextCommand)
End Using
```

Now deploy this procedure and run the T-SQL script, shown here, to execute the procedure:

non-context connection. One such instance would be when connecting to a different server, say a different SQL Server or even Oracle database, from within SQLCLR. The second instance could be where you do not wish to enlist within the same transaction as the SQLCLR is executing upon (more on this shortly).

The point of transactions brings up another important question: "If an update is issued on the newly spawned regular connection object, does it roll back or commit with the transaction that the calling SQLCLR code was already running under?"

The good news is that the framework, with the help of System.Transactions , will be able to detect this new connection instance as an RM that is enlisting within the same transaction. The only exceptions are loop-back connections, which, in SQL Server 2005, are unable to enlist within the active running transaction. Thus, even if you were to open a connection to an Oracle database from within SQLCLR, due to the seamless integration of System.Transactions , the commands executing on the new OracleConnection will be enlisted within the same transaction as the commands on the existing SqlConnection. But since there are two connection object instances within the same transaction, the transaction will now be a distributed transaction.

There exists, however, a way to prevent this auto-enlist behavior. All you have to do is add the following to the connection string to prevent auto-enlistment:

```
enlist = false
```

Note There is one nuance to this that you must be careful of: the same does not apply to creating a loop-back connection to the same database that you were working with in the case of SQL Server 2005. The new loop-back connection will not be able to enlist itself in the same transaction. In fact, when creating a new loop-back connection, the connection will fail if there is an active transaction, unless you add `enlist = false` to prevent it from trying to enlist in the same transaction.

Here is a cool thing you could not do (easily at least) before SQLCLR. In SQLCLR code, you can open a non–context connection by specifying a full connection string instead. When doing so, add `enlist = false` to the full connection string. This will create a brand new connection that does not enlist in the calling transaction. This way, you will be able to start a brand new and not-nested transaction from within one transaction and be able to log failures, etc.

Regular non–context connections inside SQLCLR work under a few restrictions.

First and foremost is the fact that unless you specify explicitly that you do not wish your new connection to be auto-enlisted in the transaction, the new connection will automatically enlist itself under a distributed transaction. There is no way that you can work under a non-distributed transaction even if you were looping back to the same database.

In addition to that, features such as asynchronous command execution and the `SqlDependency` object and the related infrastructure do not work on regular connections inside SQLCLR.

In addition to being able to leverage automatic enlistment of your new connection instances inside an already running transaction, you can use TransactionScope just as you would use it in the middle tier. The usage of TransactionScope has been explained in depth in Chapter 11. The one thing to be careful of in SQL Server 2005 is that even if you wrap only the context connection inside a transaction scope, the transaction scope will cause the transaction to promote. This will have negative performance implications; thus, if you are using context con-

CHAPTER 14

ADO.NET Best Practices

Welcome to the last chapter of this book. Up till now, you have looked at various important components of ADO.NET and data access in general. No matter what your role in an organization may be, no matter where you are in your level of career, there is a piece of application architect in you and, as an application architect, you need to make various decisions.

In the previous 13 chapters, you looked at various ways to achieve the same end by using a range of choices and tools available to you. With all of these possibilities and tools, and the freedom they give you, come responsibilities. As an application architect, you have to weigh the possibilities and intelligently decide which tool is right for which situation.

Anything that surrounds application architecture, especially data access, is subject to varying shades of gray. In other words, you canÕt just listen and apply any presented concept verbatim; instead, you must listen, learn, and then apply an inventive combination of solutions that were presented in this book to the specific situation you are faced with. There are various shades of gray because, depending on your needs and the underlying table structure, the decisions you need to make could change. Table structures are driven by requested requirements; hence, a table structure that works well for a reporting application does not work well for an online transactionÐprocessing application.

Thus, if your decisions are driven by requirements, then it makes sense to start with the requirements first.

Know Your System Requirements

I can safely assume that you, my reader, are in some manner related to the software industry. So you will relate very well to what I am about to tell you.

The software industry is weird. We walk into a clientÕs building knowing very little about the nature of their business and what they do. In addition, the client knows very little about software (but in many instances, he thinks he knows a lot). Yet, we are expected to solve their problemsÑand most of the time we succeed. Of course, at times, we need to solve problems that were created by other software developed in the past, primarily because of a phenomenon commonly referred to as Òscope-creep.Ó Then as the scope of the project creeps even further, someone else solves the problems we created.

Not many nonÐcomputer professionals are able to appreciate that our work involves imagining a skyscraper with all its intricate details, down to the very last brick and steel harness, and then building every part of it as only imagined, without any physical entity to touch,

This is in contrast with the data reader where, even if your result set had two rows in it, while you were working on the first row and hadn't explicitly closed the data reader and the connection, your connection is still open and cannot be used by anyone else. Thus, in such a situation even if the data reader is faster in a single-user scenario, in a highly concurrent scenario it will actually be slower and, hence, not a good choice.

You might raise an eyebrow here and bring up the question that a data adapter internally uses a data reader, too. But, again, even if the data adapter uses a data reader, it frees up the data reader as soon as it can. It fills up a DataSet or DataTable and returns the object. The time that it takes you to actually process upon the data is when a connection will stay closed and, hence, effectively pooled.

DataSet or Strongly Typed DataSets

DataSets out of the box suffer from one big problem: even though they try to mimic an in- memory relational data structure, they fall short on representing the actual data types contained within. Everything is an object. While that is the most portable mechanism, you would typically then need to implement checks and conditions at each point to ensure that the data contained within is not corrupted.

The other way out is to create a strongly typed DataSet. Strongly typed DataSets are as the name suggests—strongly typed, but they, too, suffer from their own set of disadvantages. The biggest problem with strongly typed DataSets is that their structure needs to be continually updated to reflect the underlying table structure. This involves code generation and recompilation.

Also, as the data structure underneath changes, the table adapters associated with the strongly typed DataSet continually get out of date as well. This is probably not an issue in a small application, but maintaining and deploying such a library of strongly typed DataSets can turn into a major headache.

Most of all, DataSets and strongly typed DataSets provide you with a ready-to-use, in-memory disconnected data cache, but at the end of the day, they contain no true knowledge of what that data truly is. It's just a dumb bucket of data. Strongly typed DataSets try to come close to the true representation of your data, but they fall short by far.

It is in these instances where you must resort to the use of a custom business object.

Strongly Typed or Not? DataSet vs. Business Object

Given the fact that for good performance for highly concurrent systems (and in order to make a distributed, disconnected architecture work) you need some kind of disconnected data cache, you can use either the disconnected data cache that comes as a part of the .NET Framework (the DataSet), or you can build your own business objects.

A business object is an object representation of a logical entity in the specific business domain that the program is being written for. For instance, if you are writing a program for an insurance company, an example business object could be a policy, premium, payment plan, etc. Business objects present several advantages as compared with DataSets. Some of the advantages are as follows:

1. Your customer in an insurance company understands a "policy," not a DataSet or a DataTable. This allows you to abstract and communicate what you, as a programmer, see on your screen and in your program more effectively with the end customer. This is helpful because the customer doesn't care about the technical implementation and will

transaction ends up promoting to MSDTC if a second RM comes into the picture. This, of course, can be controlled programmatically by not using a `TransactionScope` block, but instead manually enlisting within such a transaction (this has been covered in Chapter 11).

5. There may be a situation where you might wish to enlist in a transaction from either within SQLCLR on a separate database, or from any ADO.NET code that tries to enlist an RM that deals with a durable resource, which has nothing to do with databases in general. In this scenario, your transaction will be managed by MSDTC by default and, hence, will be rather expensive to manage.

6. The final flavor of transactions is where you need transactional sanctity over a distributed or loosely coupled system, such as a web service or message queuing architecture. In this scenario, you cannot lock the resources until the outcome of the transaction is definite. Here you need to build an architecture that loosely wraps functionality similar to a two-phase commitÑexcept you cannot use MSDTC, `TransactionScope`, `SqlTransaction`, or anything in that family to directly wrap your entire loosely coupled transaction. Instead, what you need to do is create rollback/fail mechanisms yourself. For instance, you could implement such a transaction by storing a snapshot of previous data, which acts as your Òrecovery contractÓ and a flag on the Òin doubtÓ rows. In the event of success, you can clear out the recovery contract and the in doubt flag and, in event of failure, you can roll back using the recovery contract and then clear out the Òin doubtÓ on the various entities involved. The in doubt flag serves the purpose of informing other parts of the system to desist from using that data since its state is Òin doubt.Ó

So which of these six options should you pick? The lowest one you can get away with In other words, you should prefer to stay away from transaction architectures designed for more than what you need. True, you can kill a mosquito with a sledgehammer, but do you really want to pay the extra price of picking up that heavy hammer and risking your friendÕs hand that the mosquito was sitting on?

There are, however, two choices that seem juxtaposed with each other. These are wrapping transactions in SQL or wrapping them in `DbTransaction` objects. The cardinal rule here is prefer to wrap them in API (i.e., `DbTransaction` objects). If a stored procedure or SQL in general needs a transaction, then itÕs okay to embed a transaction in there, though then you need to be careful of nested scopes. An example of where that might be a better choice would be where you need to ferry large amounts of data back to the client in order to process the next command. In that circumstance, it makes sense to keep your logic close to the data itselfÑin the database. The situation you want to avoid is where SQL commits or rolls back transactions started in the API or vice versa. In other words, donÕt mix and match.

Picking the Right Tools: The Clearly Darker Shades of Gray

Depending on the exact situation, itÕs not just an array of choices you need to pick from. There are situations where it isnÕt a matter of choice, but discipline. In spite of the many shades of gray, there are a few portions that are clearly black and should be watched out for. Thus, doing certain things is clearly good and not doing them is clearly bad. LetÕs look at some of these car-

Implementing a Data Layer

In most scenarios, implementing a data layer is a good idea. A data layer is simply a set of classes, a web service, an application server, or just any piece of code that all your data access is done through. Implementing a data layer is a good idea because of the following reasons:

1. You isolate yourself from any changes in the framework. If something changes in the framework, you only need to change your data access layer to ensure that your entire architecture doesn't change.
2. You ensure that best practices are followed by everyone. This is because in a team it might get difficult to enforce rules on everyone. If all data access is done out of one data layer, you ensure that the best practices are enforced in the data layer, and everyone automatically garners the benefit.
3. You ensure that everyone uses the same connection string defined in the configuration file, or otherwise. This helps in connection pooling all the connection requests.
4. A data layer ensures that you have one convenient place to put performance-measuring metrics and if there is a problem with your data access architecture, then you have one place to look, and one place to fix.

Closing Connections

This is one of those insidious and serious issues that tends to creep into your application architecture with the least of warning. It only raises its ugly head when the application is in production, working (or failing) under a heavy load. The reason this is a difficult problem to track is because not only will the effect not be apparent in development, where you are testing logic, not scalability, but also because generally for situations such as transactions, your data layer will either need to expose an open connection or accept an array of `DbCommand`s with parameters in place. While the first approach relies on the end user of your data layer to make sure that any open connections are eventually closed, the second approach results in an unwieldy class structure.

Before .NET was around, programmers had to worry about memory leaks. The garbage collector makes it a lot easier by routinely performing the clean-up job for them. While you still have to be somewhat careful, the rules are much easier to live by now. If you forget to call the `Dispose` method on a particular object, while that isn't a very ideal situation, the object will eventually be picked up by the garbage collector, which will call the `Finalize` method instead. For objects that occupy a large amount of memory, the garbage collector will sense the memory pressure and be invoked more frequently as need be. However, `DbConnection` objects in themselves do not occupy large amounts of memory. What they do occupy is a valuable network connection resource, which the garbage collector will blissfully ignore. This results in the garbage collector firing less frequently and independent of your open connections, thus leaving the state of `DbConnection`s open for longer than it should be.

Let's consider the situation where the garbage collector did fire between every 4 to 8 minutes and did the clean-up job that you should have done in the first place. Where you would have required the connection to be open for only a few milliseconds (typical query execution time), now your connection state remains open for a much longer time. First, when the garbage collector bumps up your unused, out of scope, but open connection objects to the second

minutes [1] before it finally cleans up objects that were bumped to the second generation. Thus, where you could have closed a connection in 100 milliseconds, your connection may now remain open for 8 to 16 minutes. Four to eight minutes for each iteration of garbage collector, I might add, is a rather conservative guesstimate on 16GB RAM application servers.

Now, the garbage collector will eventually call `Finalize`, which eventually changes the state of the connection to Òclosed.Ó But the `DbConnection` connection state is not the actual state of the underlying physical database connection. Even after the `DbConnection` is closed, the underlying network connection will still remain open for a little bit longer to serve any other immediate requests using the same connection string.

While instead of keeping a connection open for 100 milliseconds, you ended up keeping it open for 8 minutes (these are more or less the best-case scenario numbers), ADO.NET was piled up with requests to create more and more open connections, simply because the connection you should have released was still unavailable.

So what happens, essentially, is that because there is a fewer number of open available connections, more requests stand in queue; and where you needed to serve a time period of 100 milliseconds, you are now serving 8 minutesÕ worth of connection open requests, not to mention the backlog that might create. So the whole thing comes tumbling down like a deck of cards.

So, as a conservative guess, if your connection state remains open for 8 minutes instead of 100 milliseconds, your pool would now need to handle 4,000 times more connections than it should have. And if you had ten servers connecting to the same database, each with the same problem, you can safely assume that your database is now handling 40,000 more connections than it needs to. It is amazing that the whole application still works! It probably works because the default setting for the maximum connection pool size is 100 connections, which means starting at the 101st connection request all the way to the 4,000th connection request (possibly much more than 4,000) will have to wait in line for an available connection.

Now if you think this sounds bad, sit back in your seat, hold on to something firm and consider thisÑthis awful scenario is based on the conservative assumption of a single leaky connection and the garbage collector firing every 4 minutes. You should consider yourself lucky if you were to get away with so little.

So thank the maximum connection pool setting that the database server didnÕt catch fire, commit suicide, or file for divorce, because not closing connections is truly a terrible thing to do. The short-term solution, of course, is to increase the connection pool size, but the true solution is to find the source of that leaky open connection and plug it.

And of course, let me reiterate the golden rule one last time in this bookÑ open a connection as late as possible, and close it as early as you can

Network Latency

One of the most important things to remember about large applications is that latency is your enemy. Latency, in our context, is the delay and lock of resources incurred while obtaining and opening a connection, executing commands, and getting the results back over a network. You want to perform this activity as infrequently as possible throughout an application. If you keep this in mind, performance tuning an application may be easier than expected.

In small applications or desktop applications, when accessing local data stores such as an Access database or an Excel spreadsheet, the overhead of opening and closing connections

---

, but this is a rough guesstimate.

Index

Numbers and Symbols

! (exclamation mark), indicating an error in a DataRow, 333Ð334
& (ampersand) character, entity encoding representation for, 426
<, as the entity encoded form of the less-than character, 426
? (question mark) character, representing a configurable parameter with, 332
< (less-than) character, used to start each element within an XML document, 426
<sync> element, for designating synchronization should take place, 455
> (greater-than) character, entity encoding representation for, 426
Õ (apostrophe) character, entity encoding representation for, 426
Ó (quotation mark) character, entity encoding representation for, 426

A

AcceptChangesDuringUpdate property, setting on DataAdapter object, 391Ð392
ACID properties, guidelines followed by transactions, 356Ð357
Action property, for DataRowChangeEventArgs object, 121
ActiveX Data Objects (ADO), 3Ð4
  vs. ADO.NET, 7
Add Connection dialog box, choosing the right database in, 182
Add Database Reference dialog box, choosing a connection for your project in, 469
Add(DataColumn(), DataColumn()) method, function of, 126
Add(String, DataColumn(), DataColumn()) method, function of, 126
Add(String, DataColumn(), DataColumn(), Boolean) method, function of, 126
Add(ÒColumnNameÓ) method, for adding a DataColumn with a specified name to a table, 113
Add() method, for adding a new DataColumn to the DataColumnCollection, 113
Add(myDataColumn) method, for adding a specified preexisting DataColumn object to the DataColumnCollection, 114
Add() method, creating a new DataColumn by invoking, 113
Add(String, DataColumn, DataColumn) method, as DataRelation: function of, 126
Add(DataRelation) method, function of, 126
Add(String, DataColumn, DataColumn, Boolean) method, function of, 126
Add(DataColumn, DataColumn) method, function of, 126
Add(DataColumn[], DataColumn[]) method, function of, 126
Add(String, DataColumn[], DataColumn[]) method, function of, 126
Add(String, DataColumn[], DataColumn[], Boolean) method, function of, 126
Add() method, list of seven overloaded, 126
Add(default option) setting, for MissingSchemaAction property of the data adapter, 211
Add(ÒSubTotalÓ, Type.GetType(ÒSystem.SingleÓ), ÒSum(Price)Ó) method, for adding a DataColumn with a specified name, data type, and Expression property, 114
AddWithKey setting, for MissingSchemaAction property of the data adapter, 211
ADO.NET
  10,000-ft. birdÕs-eye view of, 16
  ability to connect to a data source provided by, 53Ð66
  vs. ADO, 7
  best practices, 513Ð527
  connected objects, disconnected objects, and the DataAdapter, 6
  creating parameterized commands: DbParameter, 20Ð21
  database transaction support in, 359Ð362
  a drag-and-drop operation in for creating a data-driven application, 16
  establishing a connection: DbConnection, 17Ð19

ADO.NET (continued)
exception handling in, 26Ð30
executing a command: DbCommand and DbTransaction, 19Ð21
extracting DataSet changes as an XML diffgram, 299Ð300
holding disconnected data: DataSet, 22Ð24
implementing batching at the TDS protocol level in, 274
implementing savepoints in, 386Ð389
important objects in, 4Ð7
an introduction to, 1Ð14
main objects in the connected part of, 4Ð5
mapping mechanism, 206Ð211
objects and facilities provided by for sorting, searching, and filtering, 213Ð245
picking the right tool for the right job, 514Ð522
possible isolation levels in, 371Ð372
reading XML columns in, 444Ð446
some cardinal rules of working with, 522Ð526
and SQLXML, 447
structure of SqlDataAdapter and OracleDataAdapter in, 178
using FOR XML queries with, 434Ð437
using OPENXML with, 441Ð443
using TableMappings in, 207
ways of writing provider agnostic code in 1.1 and 2.0, 9
what it is, 2Ð3
what sets it apart from previous data access technologies, 3
and XML, 413Ð459
ADO.NET 1.1
common behavior enforced by the IDbConnection interface, 65
dealing with corrupt connection pools in, 74
ADO.NET 2.0
advanced single database techniques, 385Ð392
asynchronous execution introduced in, 92
dealing with corrupt connection pools in, 74
implementing distributed transactions in, 397Ð401
new ConflictOption property in, 263
new WriteXML method in, 117
ADO.NET 2.0 Framework, common data providers that come with, 8Ð9
ADO.NET exceptions, four major categories, 28Ð30
ADO.NET Hello World!, 31Ð52
code to fill and write from the data source, 45Ð46
hybrid approach for creating code, 45Ð48
steps for writing a fully hand-written
ADO.NET mapping method vs. SQL aliasing method, 206
ADO.NET object model, reference chapter, 15Ð30
ADO.NET transactions, pros and cons of using, 409
Advanced .NET Remoting,Second Edition (Apress, 2005), by Ingo Rammer and Mario Szpuszta, 135
Advanced Options dialog box
accessing from the TableAdapter Configuration dialog box, 251
options in, 252
aggregate calculations. See also calculations
filtering to include only a subset of rows based on a condition, 225Ð226
performing, 224Ð226
aggregate functions
creating, 487Ð494
defined, 487
<all> element, in <complexType> XSD element group, 148
ampersand (&) character, entity encoding representation for, 426
AnimalID property
in the Animals table, 129
in the Pets table, 129
using instead of animal name in optimistic concurrency, 326Ð327
AnimalName property, in the Animal table, 129
Animals table
properties in, 129
script for creating in Test database, 248
setting up the insertAnimalCommand in, 339
stored procedure for inserting an animal into, 340
Animals.AnimalID, set up as primary key in Exercise 10.1, 335
animalsBindingNavigator control, checking that Save button enabled property is set to true, 249
AnimalsCopy table, inserting data read from data reader into, 277
animalsTable
contents of after Erick calls DataAdapter.Update, 283
ErickÕs view of, 283
AnimalsTableAdapter, various methods in, 254Ð255
AnimalsTableAdapter TableAdapter, configuring, 250Ð254
AnimalsTableAdapter.Fill method, 255
AnimalsTableAdapter.InitCommandCollection method, 256

<annotation> element
for annotating XML schemas, 150
using <appinfo> element within, 151
apostrophe (Ô) character, entity encoding representation for, 426
App.Config file, code for the connection string in the Windows Forms application, 43
<appinfo> element, using with XML schema file, 151
Applications, picking the right tool for writing, 514Ð522
appSetting element, specifying connection strings as in .NET Framework 1.x, 62
ArrayList object, binding with myDataGrid, 90
AS keyword, using in Exercise 7.5, 204Ð206
ASP.NET 2.0, creating a new website in, 33Ð39
Atomic transactions, defined, 356
Attach to Process dialog box, debugging SQLCLR code without the aid of the SQL Server project, 474Ð475
<attribute> element, important attributes of in XSD, 144
AttributeOrElementName, function of in FOR XML EXPLICIT query, 424
AUTO mode, function of in FOR XML clause, 415

B

BasePath property, for SqlXmlCommand object, 448
batched SQL command, in Microsoft SQL Server, 97
BCP (Bulk Copy), moving large amount of data between databases with, 276
Begin method
code illustrating usage of, 389
use of in transactions, 358
BeginEdit/EndEdit methods, of DataRow object for modifying existing rows in a DataTable, 280
BeginExecuteDataReader, code for callback implementation for, 93
BeginExecuteReader
code for calling, 92Ð93
BeginTransaction method
for beginning a transaction on a database, 66
specification of the IsolationLevel by, 66
best practices in ADO.NET, 513Ð527
BINARY BASE64 argument, for retrieving binary data for various FOR XML queries, 419
BinaryFormatter
DataSet serialized as true binary using, DataSet serialized as XML using, 139
using to Serialize and Deserialize a DataSet, 137Ð138
BindingSource class, function of under the System.Windows.Forms namespace, 16
BookDataSet class, definition of the Books property, 165
BookDataSet.xsd file
definition of relationship between Books table and BookReviews table, 165
new for Example 6.6, 155Ð156
opening in Notepad and viewing BookReviews table defining element, 164
opening in Notepad and viewing Books table element representation, 164
BookReviews table
rows defined by a <BookReviews> element, 158Ð159
updated Books.xml file containing information for, 158
BooksRow class, looking at the Title property of and VB.NET, 166Ð167
Books.xml document, creating for your sample application to read, 153
Books.xml file, updated that contains information for BookReviews table, 158
btnClose event handler, adding code to ensure the form is closed when Exit button is clicked, 297
btnGetChanges OnClick event handler, adding code to for performing a GetChanges and rebinding DataSet to UI, 297Ð298
btnSumPrices button, for computing aggregate sum function on the Price column, 224Ð225
business layer vs. SQLCLR, 462
business objects
vs. DataSets, 134
vs. DataSets, strongly typed or not, 517Ð519
defined, 22
disadvantages of, 519

C

C#
adding code in the Form1_Load event handler in, 227Ð228
adding the UpdateBatchSize property in, 274
AnimalsTableAdapter.Fill method in, 255
AnimalsTableAdapter.InitCommandCollection method in, 256
binding the DataGridView with the right table in Exercise 7.3, 195

C# (continued)
BookDataSet class definition of the Books property in, 165
changing the CommndText and dataAdapter.Fill statements in, 195
checking row state of new row in, 271
checking the final row states of the updated DataTable in Exercise 9.3, 274
command text for querying for a result set, 85
constructing a DataRelation object using an array of DataColumns in, 127Ð128
constructing the DataRelation object and adding to the Relation collection, 127
constructor of DataViews allowing you to specify information in one code line, 233
constructor that directly ties the DataView to a DataTable, 232
correct way to order exception-handling blocks in, 27Ð28
creating a calculated column in, 222
creating a data adapter to retrieve records from a database in, 205
creating a DataView without specifying any information, 232
creating an event handler for DataBind button click event, 187
creating a new XmlDataDocument instance called xdd in, 242
creating a ready-to-use SqlConnection object, 55
creating connection objects in, 54
data loaded, sorted, and databound to a DataGridView control in, 235
defining a connection object in Exercise 7.5 in, 205
DeleteCommand in, 257Ð258
displaying the contents of a DataSet in, 200
displaying the contents of one row in, 218
enumerating through available providers in, 10Ð11
filling a DataSet in for Exercise 7.4, 198
filling a DataSet programmatically in, 190
filling a DataSet schema in Exercise 7.4, 199
filling a DataSet with the UserTable in, 194
filling a DataTable and displaying the row states in, 269
filling the productsTable in, 118
finding child rows for a given selected row in, 228Ð229
finding parent rows for various rows in, 230
function of Add(DataColumn[], DataColumn[], Boolean) method in, 126
function of Add(String, DataColumn[], DataColumn[]) method in, 126
going through rows one by one and printing values to Console, 206
incorrect way of ordering exception-handling blocks in, 27
indexing through BooksRow objects in strongly typed Books.Rows collection, 163
InitAdapter in, 258Ð260
invocation of GetChildRows() through the GetBookReviewsRows() method in, 166
looking at the Title property of BooksRow class in, 166
making changes to the DataTable in, 270
modifying code to include only a subset of rows based on a condition in, 226
proving DataSet relationship has same name as <keyref> element in XSD file in, 157
putting the ProviderBase model to work in, 11Ð12
refreshing the contents of XmlViewer, 243
retrieving UDT data in a connected fashion from a database, 105
selecting a number of rows in a DataTable in, 219
serialization and deserialization of a DataSet in, 137
setting up the DataSet in, 215
setting up the productsTable in, 117
setting various commands with SqlCommandBuilder in, 272
sorting data on FirstName column in ascending order, 235
specifying DataType property on a basic DataColumn in, 223
specifying parameters of OracleType.Cursor type to SelectCommand on the OracleDataAdapter, 192
specifying parameters using the ConnectionString property, 54
vs. T-SQL, 461Ð462
update command in CommandBuilder object, 264
using DoDataMappings method to add table mapping to the data adapter, 209
using DoDataMappings method to create the usersMapping object, 208
using Find method to identify one DataRow using a basic DataTable in, 217Ð218

using GetString method to retrieve textual representation of the UDT, 106
using Select method to find multiple rows matching a criterion in, 220
using Select method to find rows and specify a sort in, 220Ð221
using SqlDataReader object to query for a result set in, 85
using TableMappings in ADO.NET using, 207
using the Compute method in, 225
using the DoDataMappings method for declaring DataColumnMapping objects, 208
using the Fill method to fill the DataTable, 189
various methods in AnimalsTableAdapter in, 254
working with a connection object in, 55Ð56
working with a strongly typed DataSet in, 162Ð163
writing a batch file to build a strongly typed DataSet in, 167
caching data, using the HttpContext.Cache object for improved performance, 526
calculated column
creating in C#, 222
creating in VB.NET, 223
using FillStrongDataSet method to create, 223
calculations. See alsoaggregate calculations
performing aggregate, 224Ð226
callback execution, importance of closing the connection after it has occurred, 95Ð96
CancelEdit method, for canceling an edit to a DataTable, 280
Cascade value, System.Data.Rule enumerator, 116
cascading action, 115
ChangeDatabase function, for changing databases within the same server, 66
Chaos value, possible in ADO.NET IsolationLevel enumeration, 371
check-in/check-out logic, as alternative to concurrency mechanism, 331Ð334
child rows, example showing filtered out rows identified as child rows of Tarzan, 231
<choice> element, in <complexType> XSD element group, 148Ð149
ClearErrors method, for clearing all errors on a DataRow, DataTable, or DataSet, 334
ClearParameters() method, for
ClientSideXml property, for SqlXmlCommand object, 448
Close method, for closing an open connection to the database, 67
CloseConnection command behavior, for ExecuteReader method, 86
CLR (Common Language Runtime)
how it works on Windows machine vs. inside SQL Server, 462Ð464
inside out, 100
inside SQL Server 2005, 94
in SQL Server, 461Ð511
Code Access Security (CAS), as part of the CLR, 463
code example
for accessing a customer first name in a DataSet, 140Ð141
for accessing a given column in the IDataRecord, 84
accessing data in a more programmer- and reader-friendly fashion, 141
for accessing multiple parallel data readers in Oracle, 378
for adding a new DataRow into a DataTable using LoadDataRow method, 279
adding a record to DB.MDB database, 311Ð314
adding a totalPrice column, setting an expression on it, and adding to DataTable, 224
adding code to the btnClose handler to ensure form is closed, 297
for adding new DataRow objects to DataTable.Rows collection, 114Ð115
for adding new rows to a DataTable, 278Ð279
for adding the UpdateBatchSize property, 274
of ADO.NET commands for accessing a database, 78
AnimalsTableAdapter.Delete(int Original_AnimalID, string Original_AnimalName) method, 256Ð257
AnimalsTableAdapter.Fill method, 255
AnimalsTableAdapter.InitCommandCollection method, 256
AnimalsTableAdapter.Update(testDataSet dataSet) method, 262
attempting to run two data readers on an open connection concurrently, 379Ð381
for autogenerated code in the Windows Forms application, 44
binding the DataGridView with the right table, 195

code example (continued)
C# command text for querying for a result set, 85
for calling a Transaction clone, 404
calling the BeginTransaction method of the connection object, 367
for checkbox that sets the nested property on the relation, 131Ð132
for checking row state of new row, 271
code to split the enumerated values into columns, 480
committing or rolling back database application changes, 368
for configuration file with encrypted section in .NET Framework 2.0, 63
the connection string defined for you in the Web.Config file, 38
for connection string for connecting to an Access database, 61
the connection string in the App.Config file, 43
for constructing a DataRelation object using an array of DataColumns, 127Ð128
for constructing the DataRelation object and adding to the Relation collection, 127
contents of batch file to build a strongly typed DataSet, 167
contents of the DataSet with Nested property set to false, 132Ð133
contents of the DataSet with Nested property set to true, 133Ð134
correct way to order exception-handling blocks, 27Ð28
creating a Books.xml document for sample application to read, 153
for creating a command object, 78
for creating a data adapter for Exercise 7.2, 189
creating a data adapter to retrieve records from a database, 205
for creating a disconnected cache of DbDataRecords, 89Ð90
for creating and setting up SqlConnection object in Main procedure, 79Ð80
for creating an event handler for DataBind button click event, 187
for creating a new DataTable object, 114
for creating a new instance of the SqlXmlCommand object, 449
for creating a parameter for the Parameters collection, 442
for creating a relationship in a DataSet using overloaded methods, 126
for creating a restriction within a
creating a SqlCommand for the MS SQL Server 2005 database, 188Ð189
for creating a table that uses the previous UDT, 104
for creating a UDF from a register assembly in SQL Server 2005, 467
for creating a UDF using a SQL Server project, 471
for creating ConsoleApp2 application, 48Ð51
for creating Demo table in Test database, 31Ð32
for creating GetConcatenatedNames SQLCLR stored procedure, 496Ð497
for creating GetNames SQLCLR stored procedure, 497Ð499
for creating InsertName SQLCLR stored procedure, 500Ð501
creating NameRow class for SqlServerTVF project, 476
for creating Test database on local instance of SQL Server 2005, 179
for creating two tables of identical structure, 276Ð278
for data binding the DataTable to the DataGridView to display results, 224
declaring a complex type with a <complexType> XSD element, 147
declaring DataColumnMapping objects, 208
for defining relationships in the customerProducts DataSet, 228
DeleteCommand, 257Ð258
for deleting a row in a DataTable, 125
demonstrating connection pooling, 70
demonstrating use of RegionUpdate stored procedure, 442
for deploying SqlServerStoredProc, 499
for deploying the MonkeyTrigger after compiling and building, 507
displaying the contents of a DataSet, 200
for displaying the row states, 268
for enabling isolation level, 377
ensuring that table rows will insert and commit or not insert and be rolled back, 339Ð340
for enumerating through available providers, 10Ð11
for executing a simple command, 82
for executing the modified GetConcatenatedNames stored procedure, 508
for explicitly disabling connection pooling, 71
filling a DataSet for Exercise 7.4, 198
filling a DataSet for Exercise 7.4, 198
filling a DataSet programmatically, 190
for filling a DataSet schema, 199

for filling a DataSet with the UserTable, 194
for filling a DataTable and displaying the row states, 269
filling the DataSet and data binding it to the right DataGridViews, 336Ð337
for filling the DataSet and VB.NET in Exercise 7.5, 205Ð206
for filling userTable programmatically, 186Ð187
of final code for the TVF, 480Ð483
for finding a particular row in a database table, 217
for finding child rows for a given selected row, 229
for finding parent rows for various rows, 230
of form that per-column encoding takes, 424
for GiveMeFakeData(), 135Ð137
the GridView control defined on the Default.aspx page, 38
grouping attributes for reuse throughout a schema, 150
hierarchical FOR XML EXPLICIT query with various directives, 429Ð431
for implementations of event handlers, 122Ð123
for implementing distributed transactions in .NET 1.1, 393Ð397
for implementing distributed transactions in .NET 2.0, 398Ð399
for implementing savepoints in ADO.NET, 386Ð389
for implementing the NameSplitter class in C#, 477Ð478
for implementing the NameSplitter class in VB.NET, 478Ð479
of incorrect way of ordering exception-handling blocks, 27
InitAdapter in C#, 258Ð260
InitAdapter in VB.NET, 260Ð261
Insert, Update, and Delete code for a hierarchical DataSet, 348Ð350
for inserting data into the UDT column, 105
for inserting data read from data reader into AnimalsCopy table, 277Ð278
for iterating through DataSet rows and printing Title column value, 157Ð158
of key/keyref syntax for defining a parent-child relationship, 171
loading and data binding the CustProd DataSet in the constructor, 296Ð297
loading DataSet data and schema from loading schema and data for a relational DataSet, 156Ð157
for loading the DataTable in the event handler for btnLoadÕs click event, 287Ð288
for looking at the history of DataRows while changes are being made, 282
for making changes to the DataTable, 270
for manually enlisting a SqlConnection instance into a running transaction, 404Ð405
for Merge Case 1: same table structures, no primary key, 302Ð303
for Merge Case 2: same table structures, with primary key, 303Ð305
for Merge Case 3: common column, no primary key, 305Ð306
for Merge Case 4: common column, with primary key, 306Ð308
for Merge Case 5: absolutely different table structures, 308Ð310
for Merge method and VB.NET, 302
of modified script for running the InsertName stored procedure, 502Ð504
for modifying column Name from first row in DataTable, 124
for modifying existing rows in a DataTable, 279Ð281
for modifying the connection string in GetConcatenatedNames stored procedure, 508
MS SQL Server stored procedure returning multiple results, 191
of myfile.udl file content, 61Ð62
of new BookDataSet.xsd file for Example 6.6, 155Ð156
of the OPENXML function used to generate a row set, 440
Oracle package and stored procedure returning multiple results, 191Ð192
of output produced from deleting a row in a DataTable, 125
for passing a cloned transaction to a threadÕs entry point, 404
for passing property values to FillRowMethodName, 479Ð480
persisting inserts back into the database, 344Ð345
primary key specified in the DataSet schema in Exercise 7.4, 203
for producing a working DataSet(BookDataSet.xsd), 152Ð153
for proving DataSet relationship has same name as <keyref> element in XSD file, 157
for putting the ProviderBase model to work, 11Ð12

code example (continued)
for querying calculated columns in database tables, 217
for querying two tables from the TestDemo database, 97
for reading multiple result sets out of a data reader, 98
for reading out the contents of the Xml column as a string, 445
reading up an XmlReader, 436
for registering the assembly in the SQL Server, 104
for registering the HandWrittenUDF assembly inside SQL Server 2005, 466
for registering the UDT as a type from the previous registered assembly, 104
for retrieving a connection string, 65
for retrieving number of rows in the TestDemo table, 81
for retrieving UDT data in a connected fashion from a database, 105
for returning multiple result sets in a stored procedure in Oracle, 97Ð98
for running the InsertName stored procedure T-SQL script, 501
running various queries using SqlCommand object, 367Ð368
for saving encrypted data to a configuration file, 63
schema used to update the Person.Contact table, 457
of script for creating Person table, 487
of script to create various stored procedures using OPENXML, 438Ð439
for selecting a number of rows by specifying a query, 217
of sequence of commands for optimistic concurrency, 326Ð327
of sequence of queries from Erick and Frans, 328
for serialization and deserialization of a DataSet, 137Ð138
for setting a DataTableÕs primary key to an array of DataColumns, 116
for setting a particular column to null in non-strongly typed DataSets, 332
for setting a particular column to null in strongly typed DataSets, 332
for setting the CommandText property to the query string, 449
for setting the ConflictOption property value in, 264Ð265
setting the DataAdapter as a CommandBuilder object property, for setting the openXMLCommandÕs text to RegionDelete, 443
setting up a FOR XML query command in ADO.NET, 435
for setting up a Schema for petsData, 130
for setting up a SqlCommand that works with an Xml Column, 445
for setting up the DataSet, 215
setting up the empty tableÕs schema, 286Ð287
for setting up the Exercise 7.2 connection, 188
setting up the insertAnimalCommand in Animal table, 339
setting up the insertPetCommand and insertPetBelongingCommand, 340Ð343
for setting up the productsTable, 117
setting up two transactions with ReadCommitted and ReadUncommitted isolation levels, 373Ð374
for setting up various event handlers on a DataTable, 122
setting various commands with SqlCommandBuilder, 272
shortened DoDataMappings method using TableMappings in ADO.NET, 210Ð211
showing <MyElement> element with MyEnum and MyString attributes, 145
showing inefficient SQL query code for UpdateCommand.CommandText, 263
showing output from modifying column Name in DataTable, 124
showing use of FOR XML AUTO queries, 416Ð417
showing use of FOR XML EXPLICIT queries, 417Ð418
showing use of FOR XML PATH queries, 418
showing use of FOR XML RAW queries, 415Ð416
showing use of FOR XML RAW query with BINARY BASE64 argument, 420
of simple SQL command for querying a UDT column, 105
for sorting data on FirstName column in ascending order, 235
for sorting the results of a query, 217
of specifying a hidden column, 428
for specifying DataType property on a basic DataColumn, 224
specifying parameters of OracleType.Cursor type to SelectCommand, 192

for specifying the Timestamp column during table creation, 330
for specifying the .udl file as your connection string, 61
of SqlConnection holding the context connection, 495
for a SqlConnection instance to enlist itself in a transaction, 403Ð404
the SqlDataSource control defined on the Default.aspx page, 37Ð38
stored procedure for inserting an Animal into the database, 340
stored procedure for inserting a Pet and a PetBelonging into the database, 343Ð344
of syntax for declaring an attribute in XSD, 144
of template for defining a query that selects products, 452
three possible ways of instantiating the command object, 80
transactional command execution and VB.NET, 365Ð367
for turning off connection pooling for SqlClient, 69
of TVF used to display the files contained in a directory, 485Ð486
of a typical DbProviderFactories section in the Machine.Config file, 10
for UP_ANIMALINSERT stored procedure, 291
update command in CommandBuilder object, 264
updating a record using ColumnMappings, 315Ð317
for updating only the DogÕs weight, 328
using a key constraint, 154
using <all> element in <complexType> XSD element group, 148
using BeginEdit/EndEdit methods, 280
using <choice> element in <complexType> XSD element group, 148Ð149
using CommandStream property to set a stream instance, 449
using Commit and Rollback methods, 361Ð362
using <complexType> XSD element mixed attribute, 147
using <documentation> element, 151
using DoDataMappings method to add table mapping to the data adapter, 209
using ExecuteToStream() method to run query and fetch results, 450Ð451
using ExecuteXmlReader method to read
using FindByCustomerID method by using a strongly typed DataTable, 219
using Find method to identify one DataRow using a basic DataTable, 217Ð218
using FOR XML optional arguments, 419
using FOR XML queries with ADO.NET, 435
using GetXml to show DataSet contents as XML, 131
using <group> element in <complexType> XSD element group, 149Ð150
using Merge method to integrate changes with an existing DataSet, 300
using OPENXML in conjunction with a SQL DELETE operation, 440
using Select method to find multiple rows matching a criterion, 220
using Select method to find rows and specify a sort, 220Ð221
using <sequence> element in <complexType> XSD element group, 149
for using SqlConnectionStringBuilder class, 58
using SqlParameter, 451
using SqlXmlParameter object, 451
using templatized queries in Exercise 12.5, 452Ð454
using the CancelEdit method, 280
using the element directive in a SQL query, 427
using the Fill method to fill the DataTable, 189
using the new FOR XML PATH syntax, 433Ð434
using the ORDER BY clause, 428
using the RegionUpdate stored procedure UPDATE statement, 441
using the xml directive in a SQL query, 427
using the xmltext directive in a SQL query, 428
using UpdateGrams to update date in Exercise 12.6, 455Ð456
VB.NET command text for querying for a result set, 85
working with a connection object, 55Ð56
working with a strongly typed DataSet, 162Ð163
working with two commands in two transactions on different isolation levels, 374Ð376
for wrapping UserDefinedFunctions class inside the HandWrittenUDF namespace, 465Ð466

code example (continued)
of FOR XML AUTO query that accesses ProductProduct Photo and ProductPhoto, 421
of FOR XML AUTO query that specifies the ELEMENTS option, 421Ð422
XML document with two-level hierarchy, 147
of an XML file of an UpdateGram used to update a product, 455
of an XSD file, 143Ð144
XSD for visually designed BookDataSet strongly typed DataSet, 161Ð162
for the XYCoOrdinate UDT, 101Ð104
codegen namespace, attributes that affect code generation of a DataSet, 169Ð170
column ordinal, using to access a column value, 84
Column property, for DataColumnChangeEventArgs object, 120
ColumnChanged event, function of, 120
ColumnChanging event
behavior of vs. RowChanging event, 125
function of, 120
ColumnMappings, using to update a record, 315Ð317
Columns, calculating on the fly, 222Ð224
Columns collection, in DataTable class, 113
Command Builder object, using, 262Ð266
Command object
function of in ADO.NET, 5
minimum requirements for it to execute successfully, 78Ð79
specifying the CommandType property on, 81
using to query your database, 78Ð83
CommandBuilder object
experimenting with in Exercise 9.2, 263Ð266
leveraging to generate queries for you, 265Ð266
setting the DataAdapter as a property, 264
commands
executing to retrieve results, 82Ð83
three major categories of, 22
CommandStream property
for SqlXmlCommand object, 448
using to set a stream instance where query to be executed is stored, 449
CommandText property
setting to the query string for the SQLXML query, 449
for SqlXmlCommand object, 448
CommandType property
specifying on a command object, 81
Commit method, use of in transactions, 358
Common Language Runtime (CLR). See CLR (Common Language Runtime)
Common Table Expressions (CTE), new feature in SQL Server 2005, 461
complex types, function of in XML schemas, 147Ð150
<complexType> XSD element, 144
attribute groups, 150
for declaring a complex type, 147
element groups, 148
Compute method
example of aggregate values using, 226
for performing aggregate calculations on an entire DataTable, 224
using and VB.NET, 225
Concatenator aggregate
building the project and deploying it, 493
execution results of query, 494
full code for in C#, 491Ð492
full code for in VB.NET, 492Ð493
running a SQL query on, 493
concurrency, implementing: practical concerns, 331Ð334
concurrency management scheme
cardinal rules when deciding on, 330Ð331
deciding which to use, 247, 330Ð331
concurrency model. See concurrency management scheme
configuration file
adding a protectedData element to, 63Ð66
code with encrypted section and encrypted data, 64Ð65
conflict detection and concurrency resolution, 322Ð331
ConflictOption property
new in ADO.NET 2.0, 263
setting the value for, 264Ð265
conflicts
handling, 323Ð331
preventing, 322Ð323
connected mode, typical sequence of operations in a transaction, 359Ð360
Connection Lifetime parameter, function of in SqlClient connection pooling, 69
Connection object
defining in the Sub Main or static void main in Exercise 7.5, 205
function of in ADO.NET, 4
viewing in the Visual Studio .NET component tray, 18Ð19
connection objects
creating, 54Ð57
differences between calling Close and Dispose methods or none at all, 75

connection pooling, 22
  closing connection: good application design, 74Ð75
  code for explicitly disabling, 71
  dealing with corrupt connection pools, 73Ð74
  deciding on the right pool size for your application, 72Ð73
  effects of in data readers vs. DataSets, 516Ð517
  how it works, 71Ð72
  parameters that can be set using the connection string, 69Ð70
  what it is and how it is used, 68Ð74
connection pools, dealing with corrupt, 73Ð74
Connection Reset parameter, function of in SqlClient connection pooling, 69
connection strings
  adding code to allow use of asynchronous commands, 94
  code for connecting to an Access database, 61
  code for retrieving, 65
  code for specifying the .udl file as yours, 61
  the easy way to any, 60Ð62
  generating provider-specific, 57Ð60
  reasons not to hard code, 62
  securing, 62Ð66
  for specifying settings to connect to a data source, 54
connections
  importance of closing, 523Ð524
  importance of closing after use, 50
  opening on a database called ÒTestÓ, 55Ð57
ConnectionString property
  function of, 67
  using to specify parameters, 54Ð55
ConnectionTimeout property, function of, 67
Consistent transactions, defined, 356
ConsoleApp application, downloading from Apress Source Code section, 45
ConsoleApp2 application
  creating, 48Ð51
  downloading from Apress Source Code section, 48
ConsoleApplication, creating and naming Exercise 7.5, 205
constraining facets, list of that can be applied to a simple type, 146
Constraint object, function of in ADO.NET, 6
ConstraintName attribute, of msdata namespace, 170
ConstraintOnly attribute, of msdata
constraints
  enforcement of data integrity in relational databases with, 115Ð116
  use of on contents of a DataSet, 154Ð159
Constraints collection, in DataTable class, 113
ContainedElementName, function of in FOR XML EXPLICIT query, 424
context connection, 494Ð504
  advantages of using vs. a loop-back connection, 495
  restrictions on using, 495Ð496
ContinueUpdateOnError property, setting for a DataAdapter, 333Ð334
control levels, designating for your SQL Server code, 463Ð464
cowboy-style application, defined, 72
CREATE XML SCHEMA statement, for setting up a MySchemaCollection database, 444
CreateCommand method, for creating a Command object, 67
CreateDataSet class library project, downloading from Apress website, 214
CreateParameter() method, for SqlXmlCommand object, 448
Current constant, for DataRowVersion enumeration, 282
CustomerID, using to find out what products were ordered by customer, 227Ð232
customerProducts DataSet
  code for defining relationships in, 228
  code for loading in C# and VB.NET, 227Ð228
CustomerProducts table, many-to-many mapping between customers and products in, 214
Customers table, populating sample rows for, 364
Customers/Products scenario, data structure for, 214
CustomersView DataView
  data loaded in and databound to a DataGridView control, 235
  data loaded, sorted, and databound to a DataGridView control in, 235
  filtering out rows where last name ends in OfJungle, 235Ð236
  finding rows based on a sort key in, 237Ð238
CustProd strongly typed DataSet
  loading and data binding in the constructor, 296Ð297
  structure of, 295

D

data
  advanced scenarios for updating, 321Ð353
  fetching with DataReader and DataAdapter objects, 24Ð25

data (continued)
questions to ask before creating the updating part of your application, 247
refreshing in a real-world application, 203Ð204
retrieving in a connected fashion, 77Ð107
updating back into the data source, 247Ð319
using mapping when retrieving, 207Ð211
data access layer
advantages of implementing, 74
for interacting with a database, 74Ð75
data adapter, adding the DataTableMapping object to TableMappings property of, 208Ð209
data connection, choosing in the Data Source Configuration Wizard, 40Ð41
Data Definition Language (DDL),used to define structure of the database, 22
data layer
deciding what kind of data sanctity or integrity you require, 247
deciding which concurrency management scheme to use, 247
determining what you are designing it for, 247
reasons for implementing, 523
Data Link Properties dialog box, creating connection strings to connect to an Access database in, 60Ð62
Data Manipulation Language (DML), used to run queries, 22
data providers, advantages of using data-source specific over generic, 9
Data Query Language (DQL)
interleaving rules regarding, 382Ð383
used to query data out of the database, 22
data readers
attempting to run two on the same open connection concurrently, 379Ð381
vs. DataSet when writing applications, 515Ð517
effects of connection pooling in, 516Ð517
memory consumption of vs. that of DataSets, 515
support for multiple result sets, 516
Data Source Configuration Wizard
choosing the appropriate data source in, 181
choosing the data connection in, 41
choosing the Demo table to be part of your data source, 42
choosing to save the connection string in, 41
choosing your database objects in,
data sources
connecting to, 53Ð75
creating for the Hello World application, 31Ð32
creating table structure for, 179Ð180
operations involved in communicating with, 77Ð78
setting up, 34Ð35, 179Ð180
a typical program and its, 2
Data Sources window
choosing display mode for UserTable in, 183Ð184
enabling, 180
in Visual Studio 2005, 39Ð40
Data Source type, choosing in the Data Source Configuration Wizard, 40
data transfer objects, DataSets as, 134Ð140
data types, valid in XSD schema, 143
DataAdapter
setting as a CommandBuilder object property, 264
using transactions with, 389Ð392
DataAdapter class, function of, 13
DataAdapter object
downside of using, 275
fetching data, 24Ð25, 177Ð212
function of in ADO.NET, 5
structure of, 178Ð179
DataAdapter.RowUpdated event
adding and updating a new row in the database with, 292Ð293
using vs. UP_ANIMALINSERT stored procedure for adding a new row, 292Ð293
DataAdapters, putting to use, 179Ð204
database
code for persisting inserts back into, 344Ð345
drawbacks of locking other users out of, 323Ð325
querying for multiple result sets, 96Ð99
retrieving UDT data in a connected fashion from, 105Ð106
storing objects in, 100Ð105
what can happen between querying and saving, 322
database commands, for manipulating database data, 78
database connections, opening on a database called ÒTestÓ, 55Ð57
Database Explorer/Server Explorer window
adding a connection to set up the data source, 34Ð35
opening, 34
Database Manipulation Language (DML), commands for manipulating database data, 78
atabase property, function of, 67

database schema, querying, 196Ð204
database table, 217
database transactions
  in business applications, 357Ð358
  commands frequently used in, 358Ð359
DataBind button, creating an event handler for its click event, 187
DataColumn, function of in a DataTable, 113Ð114
DataColumn object, function of in ADO.NET, 6
DataColumnChangeEventArgs object
  properties, 120
  received by handlers for column-related events, 120
DataColumnCollection data type
  Columns property of, 113
  overloaded versions of Columns property available on, 113Ð114
data-driven application
  creating using the drag-and-drop approach, 32Ð45
  the Òwrite code yourselfÓ approach, 48Ð51
DataGridView
  adding to Exercise 7.2 form, 185
  a DbDataRecords ArrayList databound with, 90
DataGridView control (dgView)
  adding to the Exercise 8.8 form, 234
  adding to the XmlDataDocument, 241
DataGridView controls, adding to CustProd strongly typed DataSet, 295Ð296
DataReader object
  as bridge between connected and disconnected worlds, 22
  fetching data with, 24Ð25
  function of in ADO.NET, 5
DataRelation object
  constructing explicitly and adding to the Relation collection, 127
  constructing using an array of DataColumns, 127Ð128
  function of in ADO.NET, 6
  as simile of a foreign-key constraint, 23
  working with, 226Ð232
DataRelationCollection class, DataSet.Relations property as instance of, 125Ð128
DataRelations, finding parent rows for all identified rows to get product details, 231
DataRow, state management and its use in updating data, 266Ð274
DataRow objects
  adding to the DataTable.Rows collection, 114Ð115
DataRowChangeEventArgs properties, 121
DataRowCollection object, finding a row with, 217Ð219
DataRow.HasErrors property, for verifying if a row has errors in it or not, 333Ð334
DataRows
  adding new to a DataTable, 278Ð279
  deleting existing, 281Ð283
  filling the DataTable object with, 117Ð118
  looking at the history of while changes are being made, 282
  modifying by loading an object array into the ItemArray property of, 280
  modifying existing, 279Ð281
DataRowState enumeration, table of various values in, 266Ð267
DataRowVersion enumeration, constants table, 282
DataRowViews, ways to search through those contained inside a DataView object, 233
DataSet(BookDataSet.xsd), code for producing a working file, 152Ð153
DataSet class, overloaded Merge() methods, 310
DataSet object, InferXmlSchema method on, 240
DataSet object model, main constituents of, 111Ð112
DataSet objects
  as closest parallel to an RDBMS database, 23
  contents of, 23
  creating a relationship between two tables in, 126
  function of in ADO.NET, 5
  holding disconnected data, 22Ð24
  how XSDs apply to, 151Ð159
  using for working with a disconnected model, 111
DataSet object structure, overview of major data providerÐindependent disconnect objects, 23
DataSet schema
  code for filling in C# and VB.NET, 199
  and data example without the schema loaded in Exercise 7.4, 202
  and data example with the schema loaded in Exercise 7.4, 201
DataSet type, adding to Exercise 7.3, 193
DataSet.Relations property, of the DataRelationCollection class, 125Ð128
DataSet.Relations.Add() method, invoking to create a relationship between two tables in a DataSet, 126

DataSets, 109Ð175
accessing in a more programmer- and reader-friendly fashion, 141
advantages of business objects as compared to, 517Ð519
annotated typed example, 171Ð174
building strongly typed, 159Ð168
building strongly typed manually, 167Ð168
vs. business objects, 134
vs. business objects, strongly typed or not, 517Ð519
code for loading data and schema from XML and XSD, 153Ð154
contents of with Nested property set to false, 132Ð133
contents of with Nested property set to true, 133Ð134
vs. data readers when writing applications, 515Ð517
as data transfer objects, 134Ð140
displaying contents of as XML in the xmlBrowser control, 200
effects of complicated hierarchical on performance, 525Ð526
effects of connection pooling in, 516Ð517
filling more than one table, 190Ð196
filling programmatically, 190Ð191
how relations and constraints work within, 128Ð134
an introduction to strongly typed, 140Ð174
keeping in sync with the XmlDocument, 244
memory consumption vs. that of data readers, 515
merging, 310
passing over Remoting or web-service boundaries, 135
performance of strongly typed, 168
problems due to not being strongly typed, 140
pros and cons of specifying schemas on, 240
setting up in C# and VB.NET, 215
vs. strongly typed DataSets, 517
support for multiple result sets, 516
taking changes and persisting them to the database, 262
use of constraints on contents of, 154Ð159
using BinaryFormatter to Serialize and Deserialize, 137Ð138
using Merge method to integrate changes with an existing, 300
using transactions with, 389Ð392
DataTable
checking the final row states of the updated in Exercise 9.3, 273Ð274
creating a new row and not adding to the animalsTable DataTable, 271
extracting a DataTable with fewer columns than the original from, 239Ð240
finding distinct result sets in, 239
loading in the event handler for btnLoadÕs click event, 287Ð288
row states for a freshly filled, 269
row states for a modified, 271
selecting a number of rows from, 219Ð222
using Merge method to integrate changes with an existing, 300
working in, 216Ð226
DataTable class
function of, 112Ð113
various collections in, 113
DataTable events
main categories of, 121
practical usage of, 121Ð125
table of System.Data.DataTable events, 120
DataTable object
constraints, 23
contents of, 23
dynamically constructing, 116Ð120
filling with DataRows, 117Ð118
function of in ADO.NET, 6
setting up various event handlers on, 122
as simile to a table, 23
using Math.IEEERemainder() method to populate, 117Ð118
DataTableClearTableEventHandler properties, 121
DataTableMapping Object, creating to use ADO.NET mappings, 207
DataTable.Rows collection, adding new DataRow objects to, 114Ð115
DataTable.Rows object, finding a row with, 217Ð219
DataTables
other ways of merging, 309Ð310
relationships set up between in Exercise 10.1, 335
DataType property, specifying on a basic DataColumn before itÕs filled with data, 223
DataView
converting to a DataTable, 239Ð240
creating, 232Ð240
DataView object
creating a new DataTable based on rows available in, 239
function of in ADO.NET, 6

properties to convey necessary information, 233
working with, 232Ð240
DbCommand class, function of, 13
DbCommand object, for executing a command, 19Ð21
DbCommandBuilder class, function of, 13
DbCommand.ExecuteReader method, for getting a collection of IDataRecord objects, 85
DbConnection class
establishing a connection, 17Ð19
function of, 13
hierarchy showing SqlConnection and OracleConnection classes, 17
introduced in ADO.NET 2.0, 68
DbConnectionOptions class, function of, 13
DbConnectionStringBuilder class
function of, 13
for generating provider-specific connection strings, 57
DbDataAdapter class
the bridge between connected and disconnected worlds, 25Ð26
function of, 13
SqlDataAdapter and OracleDataAdapter inherit from, 26
DbDataAdapter.Update(DataTable) constant, function of, 346
DbDataAdapter.Update(DataSet) constant, function of, 346
DbDataAdapter.Update(DataRow[]) constant, function of, 346
DbDataAdapter.Update(DataSet, String) constant, function of, 346
DbDataAdapter.Update(DataRow[].DataTableMapping) constant, function of, 346
DbDataAdapter.Update overloads, table of, 346
DbDataReader class
the connected way to fetch data, 24Ð25
function of, 13
DbDataRecord class, function of, 13
DbDataRecords, code for creating a disconnected cache of, 89
DbDataRecords Arraylist, databound with a DataGridView, 90
DbException class, function of, 13
DB.MDB database (Microsoft Access), adding a record to, 311Ð314
DbParameter class
creating parameterized commands, 20Ð21
function of, 14
DbProviderConfigurationHandler class, function of, 14
DbProviderFactories section, typical code in the Machine.Config file, 10
dbRecordsHolder, code for adding to DbDataRecords file, 90
DbTransaction class
function of, 14
that implements IDbTransaction, 19
DbTransaction object, for grouping together various commands, 392
DDL. SeeData Definition Language (DDL)
deadlock detection mechanisms, resolving database contention issues with, 352
deadlocks, within one MARS transaction, 385
Debug Output window, making visible and setting the proper filter, 473
debugging
making the Output window visible and setting the proper filter, 473
SQLCLR code, 472Ð475
Default constant, for DataRowVersion enumeration, 282
default result sets, 379
Default.aspx page, opening, 33
DELETE command, using to manipulate database data, 78
DELETE method, calling to delete existing rows from a DataTable, 281Ð282
DeleteCommand, 257Ð258
DeleteRule attribute, of msdata namespace, 170
deleting
hierarchical data, 347
a record identified by AnimalID and AnimalName, 260Ð261
Demo table
configuring the data sourceÕs, 42
creating, 31Ð32
DemoID column, creating in Demo table, 32
DemoOpenXML method, using to created a SqlCommand instance for executing RegionInsert, 441
DemoValue column, creating in Demo table, 32
Design mode, Exercise 7.2 form in, 185
Design view
application form created using drag and drop in, 249
many-to-many relationship mapping example form in, 227
dgView (DataGridView)
databinding to a DataView called CustomersView, 234Ð235
dialog boxes
Example 5.4 final databound UI with a large number of results, 95
Example 5.4 notification of the callbackÕs execution, 95
Diary column, <Chapter> elements retrieved from, 428Ð429

Directive, function of in FOR XML EXPLICIT query, 424
Directives, table of for the FOR XML EXPLICIT clause, 427
dirty read
  in reference to isolation levels, 370
  as a result of the first part of a distributed transaction, 400
disconnect objects, overview of major data providerÐindependent involved in DataSet object structure, 23
disconnected data
  adding new rows or existing rows to a DataTable, 278Ð279
  editing, 278Ð283
  implementation of, 22
  leveraging XML to work with, 240Ð244
  a real-world example of editing, 283Ð294
  setting up the data source, 214Ð216
disconnected data caches, various exceptions thrown by, 28
disconnected mode, typical sequence of operations in a transaction, 360Ð361
disconnected model
  the case for, 109Ð111
  desirable characteristics for, 110
disconnected objects, in ADO.NET, 5Ð7
Dispose method, on a connection object defined by IDisposable interface, 66
Distributed Transaction Coordinator (DTC), transactional coordination for distributed applications provided by, 392Ð393
distributed transaction model, function of, 357
distributed transactions, 392Ð408
  example for dealing with distributed transactions in .NET 1.1, 397
  example of a dirty read as a result of the first part of, 400
  implementing the .NET 1.1 way, 393Ð397
  implementing the .NET 2.0 way, 397Ð401
  important players in, 392Ð393
  ODP.NET component with ability to enlist itself in using enterprise services, 394Ð396
  typical flow of, 393
DML. SeeData Manipulation Language (DML)
<documentation> element, using with XML schema file, 151
DoDataMappings method
  creating the usersMapping object, 208Ð209
  declaring DataColumnMapping objects, 208
  shortened version using TableMappings in
DQL. SeeData Query Language (DQL)
drag-and-drop approach
  in ASP.NET 2.0, 33Ð39
  creating a data-driven application with, 32Ð45
  creating a data-driven Windows Forms application with, 39Ð45
DragDropWebsite
  downloading from Apress Source Code section, 33
ds.GetChanges(), using to update the database, 458Ð459
DTC. SeeDistributed Transaction Coordinator (DTC)
DTS (Data Transformation Services), moving large amount of data between databases with, 276
dummy IDs
  generating for the AnimalID column when disconnected from the database, 286Ð287
  setting up the empty tableÕs schema in C#, 286
  setting up the empty tableÕs schema in VB.NET, 286Ð287
durable enlistment, function of, 402
Durable transactions, defined, 357

E

EditData application, creating the loaded XML file with, 215Ð216
Editing disconnected data, 278Ð283
element directive, using in a SQL query, 427
element groups, <complexType> XSD element, 148
<element> element, in XSD file, 144
ELEMENTS, use of in FOR XML AUTO queries, 419Ð420
encrypted data, code for saving to a configuration file, 63
Enlist parameter, function of in SqlClient connection pooling, 70
entity encoding
  for including XML special characters in data, 426
  table of various representations for use in XML documents, 426
enumeration constraining facet, function of, 146
enumerations, function of in XML documents, 144Ð145
error messages
  results of attempting to insert Monkey into the Person table, 507
  from running the InsertName stored procedure T-SQL script, 502

Error setting
for MissingMappingAction property of the data adapter, 211
for MissingSchemaAction property of the data adapter, 211
error-handling blocks, incorrect way of ordering, 27
event handlers
code for implementations of, 122Ð123
code for setting up various on a DataTable, 122
Example 5.1 project, importing disconnected and connected namespaces for, 79Ð81
Example 9.5, for examining the behavior of GetChanges and Merge methods, 295Ð311
exception handling, in ADO.NET, 26Ð30
exceptions
disconnected stack inheriting from System.Data.DataException, 28Ð29
other ADO.NET inheriting directly from System.Exception, 30
provider stack-specific inheriting from System.Data.Common.DbException, 29
SqlTypes-specific inheriting from System.Data.SqlTypes.SqlTypeException, 29Ð30
various others thrown by classes in ADO.NET, 29
various thrown by disconnected data caches, 28
exclusive lock, in ADO.NET IsolationLevel enumeration, 371
ExecuteNonQuery method
calling to insert data into an XML document, 442
problems with using with triggers, 333
ExecuteNonQuery() method, for SqlXmlCommand object, 448
ExecuteNonQuery method, using on DbCommand object for catching conflicts, 327
ExecuteReader method
function of in retrieving a result set, 83
results of using to iterate through a result set, 87
using to execute the SELECT * FROM TESTDEMO query, 85
various command behaviors available for, 86
ExecuteScalar method, calling, 82
ExecuteStream() method, for SqlXmlCommand object, 448
ExecuteToStream(Stream) method, for SqlXmlCommand object, 448
ExecuteToStream() method, using to run
ExecuteXmlReader() method, for SqlXmlCommand object, 448
ExecuteXmlReader method
using to execute queries containing a FOR XML clause, 414
using to read Xml data type, 445
Exercise 7.1
final form in Design mode and final running application, 184
objects added automatically in the component tray, 184
Exercise 7.2
adding and instantiating DataTable in the constructor and VB.NET, 186
adding buttons to, 185
code for setting up the connection, 188
creating a data adapter in, 189
creating a SqlCommand for the MS SQL Server 2005 database, 188
example of final running application after compiling, 188
using the Fill method to fill the DataTable, 189
writing code for querying one table, 185Ð190
Exercise 7.3
adding controls and buttons to form, 192Ð193
changing the CommndText and dataAdapter.Fill statements in, 195
DataSet UI fill example with tables filled, 196
for filling a DataSet using two different DbCommands, 192Ð196
filling a DataSet with the UserTable, 194
Exercise 7.4
the ÒCannot load schema with incorrect dataÓ exception, 202
DataSet schema and data example without the schema loaded, 202
DataSet schema and data example with the schema loaded, 201
DataSet schema and data fill form in Design mode, 197
filling a DataSet, 198
filling a DataSet schema, 199
querying the database schema, 196Ð204
Exercise 7.5
creating a data adapter to retrieve records from a database, 205
filling the DataSet, 205Ð206
going through rows one by one and printing values to Console, 206
using the SQL AS keyword, 204Ð206

Exercise 7.6, using the DoDataMappings method for declaring DataColumnMapping objects, 208
Exercise 8.2, selecting a number of rows in a DataTable in, 219Ð220
Exercise 8.5, creating an application for calculating columns on the fly, 222Ð224
Exercise 8.7, many-to-many relationship mapping example form in Design view, 227
Exercise 8.8, creating a DataView object, 233Ð240
Exercise 8.9
  creating a Window Forms application in, 241
  examining how XmlDataDocument works, 241Ð244
  keeping the XmlDocument and the DataSet in sync, 244
  XmlDataDocument and DataTable being displayed concurrently, 243
Exercise 9.1
  adding a new data source for Test database, 248
  current data in the database for testing row states, 267
  a few rows added and saved to the database, 249Ð250
  main application form created using drag and drop in Design view, 249
  objects added by Visual Studio, 249
  resolving concurrency problems in, 252Ð254
Exercise 9.2, console application using ConflictOption property, 263
Exercise 9.3
  trace results from compiling and running application, 273
  writing application for testing row states, 268Ð274
Exercise 9.4
  adding a new row in the DataTable, 288
  loading the DataTable in the event handler for btnLoadÕs click event, 287Ð288
  main form of the application in Design view, 285
  new row added and latest keys fetched from the database, 292
  original data in the DataTable, 285
  saving the added DataRows to the data source, 288Ð290
  setting up the InsertCommand and saving, 289Ð290
  SQL statements for generating next key and selecting last generated value, 294
Exercise 9.5, form in Design view, 296
Exercise 9.6
  containing a strongly typed DataSet that defines four tables, 300Ð311
  main form in Design view, 301
  various DataTables for, 301
Exercise 10.1
  application allowing the user to edit data at runtime, 337
  downloading code from the Apress website, 334
  example of added data after clicking the Save button, 346
  filling the DataSet and data binding it to the right DataGridViews in, 336Ð337
  Insert, Update, and Delete code for a hierarchical DataSet in, 348Ð350
  primary keys set up in, 335
  resolving issue of code not working for hierarchical updates, 351Ð352
  the user interface in Design view, 336
Exercise 11.1, downloading from Apress Source Code section, 365
Exercise 11.2, application that changes default isolation level of a SQL Server database, 373Ð378
Exercise 11.4, application involving two databases, 398Ð401
Exercise 11.5, for manually enlisting and multithreaded environments, 403Ð408
Exercise 12.1, demonstrating a method to use the various styles of FOR XML query, 414
Exercise 12.2
  demonstrates using OPENXML, 414
  using ADO.NET to demonstrate execution of stored procedures, 441
Exercise 12.3, reading XML columns in ADO.NET, 444Ð446
Exercise 12.4
  downloading from Apress Source Code section, 448
  using SqlXmlCommand object methods and properties, 448Ð451
Exercise 12.5, code using templatized queries, 452Ð454
Exercise 12.6, for updating data using UpdateGrams, 455Ð456
Exercise 12.7, for updating SqlXmlAdapter class with XPath and a schema, 456Ð459
Expert C# Business Objects (Apress, 2004), by Rockford Lhotka, 22
Expert One-on-One Visual Basic .NET Business Objects (Apress, 2003), by Rockford Lhotka, 22

EXPLICIT mode, function of in FOR XML clause, 415
Expression property, example of calculated columns using, 223
Expressions, calculating columns on the fly, 222Ð224
extended stored procedures, in comparison with SQLCLR, 464
extended stored procedures vs. T-SQL vs. SQLCLR, important points to consider when making this decision, 520Ð521
ExtendedProperties property, in DataSet object model, 112
EXTERNAL_ACCESS control level, designating for your SQL Server code, 463Ð464
EXTERNAL_ACCESS permission level, specifying appropriate for the TVF, 486Ð487

F

Facets, for defining a legal set of values for a simple type, 146
FamilyDB database
  hierarchy description for our query, 424
  schema for and relationships in, 423
fetching data, using DataReader and DataAdapter objects, 24Ð25
Fill Data button, creating an event handler for its click event, 186
Fill(DataSet) method, for filling a DataSet with XML results retrieved from a query, 456
Fill method, using to fill the DataTable in Exercise 7.2, 189Ð190
FillSchema method, preserving data sanctity by filling a schema right from the database, 203
FillStrongDataSet method, using to create a calculated column, 223
Find method
  for DataView object, 233
  for finding a row in a database table, 217Ð219
  using to identify one DataRow using a basic DataTable, 217Ð218
FindByCustomerID method, using by using a strongly typed DataTable, 219
FindRows method
  for DataView object, 233
  exception thrown when attempting to execute without a sort specified, 238
firehose cursors, 379
flags parameter, to OPENXML, 438
FOR XML AUTO queries, example showing
FOR XML clause
  code defining, 415
  function of in SELECT queries, 414
  permissible modes for generating SQL, 415
  in SQL Server 2005, 414
FOR XML EXPLICIT directives, 426Ð429
  table of, 427
FOR XML EXPLICIT hierarchy, creating using UNION ALL to combine results of multiple queries, 425Ð426
FOR XML EXPLICIT queries
  example showing use of, 417Ð418
  flexibility provided by for generation of XML documents, 422Ð432
  a three-level example, 429Ð432
  two-level example, 425Ð429
  using with FamilyDB database to generate an XML document, 423
FOR XML PATH and SQL Server 2005, 433Ð434
FOR XML PATH queries, example showing use of, 418
FOR XML PATH syntax, introduced in SQL Server 2005, 432
FOR XML queries
  optional arguments that can be used in conjunction with, 419Ð420
  a quick review, 415Ð418
  using with ADO.NET, 434Ð437
  XML output using ADO.NET and XmlReader, 437
FOR XML RAW queries
  example showing use of, 415Ð416
  example showing use of with BINARY BASE64 argument, 420
foreach construct, using GetEnumerator method in, 88Ð89
foreign keys (keyrefs), and relationships, 155Ð159
ForeignKeyConstraint, for enforcing referential integrity, 115Ð116
form attribute, in <attribute> element in XSD, 144
Format.Native, data types that can be used with, 491
fractionDigits constraining facet, function of, 146

G

Get[DataType] methods, implemented by the IDataRecord interface, 84
GetChanges button, DataSet resulting from a GetChanges operation, 298
GetChanges method
  DonÕt Dos for, 298
  filtering out a DataTable or DataSet with, 295
  optimizing your application with, 294Ð311

GetChildRows() method, invocation of through the GetBookReviewsRows() method, 166
GetConcatenatedNames stored procedure
creating, 496Ð497
modifying the connection string in, 508
GetName method, implemented by the IDataRecord interface, 84
GetNames stored procedure, creating, 497Ð499
GetNames.cs/GetNames.vb stored procedure, creating and naming, 496
GetOrdinal method, implemented by the IDataRecord interface, 84
GetProviderSpecificValue method, for returning a string rather than an XmlReader, 446
GetRandomNumber UDF
creating, 466
output of SQLCLR handwritten, 467
testing, 472
GetString method, using to retrieve textual representation of the UDT, 106
GetUpdateCommand method, using, 264
GetXml
using to show DataSet contents as XML, 131
GiveMeFakeData()
code for in C#, 135Ð136
code for in VB.NET, 136Ð137
greater-than (>) character, entity encoding representation for, 426
GridView control
adding to the .aspx page, formatting, and setting properties on, 36
defined on the Default.aspx page, 38Ð39
<group> element, in <complexType> XSD element group, 149Ð150
GUIDs, using as primary keys, 294

H

HandWrittenUDF class library project, creating, 465Ð467
HasErrors property, for DataSet and DataTable for identifying errors at a global level, 333Ð334
Hello World application. See alsoADO.NET Hello World!
compiling and running in ASP.NET, 36Ð37
setting up the data source, 31Ð32
hide directive, example using, 428
hierarchical data
deleting, 347
inserting into Exercise 10.1 application, 338Ð346
order for saving, 348
a sample database, 334
updating, 347
working with, 334Ð352
high-demand applications, writing ADO.NET code for, 68Ð69
history of changes, importance of for working with disconnected model, 110
hybrid approach
for creating code for ADO.NET Hello World application, 45Ð48
results of compiling and running the application, 46
steps involved in producing code, 46Ð48

I

IDataReader interface, implementation of IDataRecord and IDisposable interfaces by, 83
IDataRecord interface
a few important methods implemented by, 83Ð84
reasons to use SqlDataReader object instead of, 85Ð86
IDbConnection interface
common behavior enforced by, 66Ð67
methods it requires the .NET data provider writer to implement, 66Ð67
properties it requires you to implement, 67
IdbTransaction, implemented by the DbTransaction class, 19
IDisposable interface, importance of implementation of by connection object, 66
IDisposable.Dispose method, used for cleanup jobs for any object, 74Ð75
idoc parameter, to OPENXML, 438
Ignore setting
for MissingMappingAction property of the data adapter, 211
for MissingSchemaAction property of the data adapter, 211
implicit casting, a basic DataSet back from a strongly typed DataSet, 224
inbuilt scalar types, defined, 99
InferXmlSchema method, on DataSet object, 240
InitAdapter
code for in C#, 258Ð260
code for in VB.NET, 260Ð261
injection attacks, parameterized queries resistant to, 20
INSERT command, using to manipulate database data, 78
InsertCommand, setting up and saving in Exercise 9.4, 288Ð290
Inserting, hierarchical data, 338Ð346
InsertName stored procedure, creating, 500Ð501

insertPetBelongingCommand, setting up with insertPetCommand, 340Ð343
insertPetCommand, setting up with insertPetBelongingCommand, 340Ð343
Int32 indexer, using in a loop instead of the string indexer, 85
interleaved command execution, with MARS, 381
interleaving rules, in a MARS-enabled world, 382Ð383
interop, used to access COM objects, 3
IsDbNull method, implemented by the IDataRecord interface, 84
IsNull, Null methods, function of in SQL queries, 104
Isolated transactions, defined, 357
isolation levels
  changing, 373Ð378
  defined, 370
  examining the effect of, 370Ð378
  reading current value of, 372
  setting up two transactions with ReadCommitted and ReadUncommitted, 373Ð374
  understanding related terms used when discussing, 370Ð371
  working with two commands on the same table in two transactions on different, 374Ð376
IsolationLevel, specification of by the BeginTransaction method, 66
IsolationLevel enumeration, values accessible via, 371Ð372
ItemArray property, modifying a DataRow by loading an object array into, 280

K

key constraints
  creating, 154
  vs. unique constraints, 154Ð155
KeyInfo command behavior, for ExecuteReader method, 86
key/keyref syntax, for defining a parent-child relationship, 171
<keyref> elements, within an XML schema, 155Ð159

L

last-in wins, in optimistic concurrency scheme, 325Ð326
lblSumPrice label, that holds results of aggregate calculations, 224Ð225
length constraining facet, function of, 146
less-than (<) character, used to start each element within an XML document,
Lhotka, Rockford
  Expert C# Business Objects (Apress, 2004) by, 22
  Expert One-on-One Visual Basic .NET Business Objects (Apress, 2003) by, 22
Lightweight Transaction Manager (LTM), introduction of in .NET 2.0, 401
lingua franca, defined, 413
LoadDataRow method, for adding a new DataRow into a DataTable, 279
local transaction model, function of, 357
loop-back connection vs. advantages of using context connection, 495
LTM. See Lightweight Transaction Manager (LTM)

M

Machine.Config file, code showing a typical DbProviderFactories section in, 10
Main procedure, creating and setting up the SqlConnection object in, 79Ð80
mapped names, using to update records, 311Ð318
mapping
  column names with built-in SQL feature, 204Ð206
  provided by ADO.NET, 206Ð211
  using when retrieving data, 207Ð211
MARS (Multiple Active Resultsets), 73
  code for accessing multiple parallel data readers in Oracle, 378
  deadlocks within one transaction, 385
  interleaved command execution with, 381
  a little practical word of advice, 384Ð385
  new feature supported in SQL Server 2005 and above, 378Ð382
  and transaction savepoints, 383
  and transactions, 382Ð385
  when not using SqlClient, 381Ð382
Math.IEEERemainder() method, using to populate the DataTable object, 117Ð118
Max Pool Size parameter, function of in SqlClient connection pooling, 70
maxExclusive constraining facet, function of, 146
maxInclusive constraining facet, function of, 146
maxLength constraining facet, function of, 146
Merge Case 1, same table structures, no primary key, 302Ð303
Merge Case 2, same table structures, with primary key, 303Ð305
Merge Case 3, common column, no primary key, 305Ð306

Merge Case 4, common column, with primary key, 306Ð308
Merge Case 5, absolutely different table structures, 308Ð310
Merge methods
  code for, 302
  for merging a specified DataSet, DataTable, or array of DataRow objects, 295
  merging two DataSets/DataTables with different schemas, 310Ð311
  optimizing your application with, 294Ð311
  overloaded DataSet class, 310
  using to integrate changes with an existing DataSet, 300
Microsoft ADO.NET. See ADO.NET
Microsoft Data Access Application Block, website address for downloading, 74
Microsoft Distributed Transaction Coordinator (MSDTC). See also Distributed Transaction Coordinator (DTC)
  used by large applications where more than one database is involved, 357
Microsoft .NET Framework, ADO.NET as a part of, 2Ð3
Microsoft SQL Server 2005, strong support for transactions provided by, 357
Microsoft Transaction Server (MTS), and COM+ as popular middlewares, 357
Microsoft universal data link (.udl) file, as alternative method for creating complex connection strings, 60Ð62
Min Pool Size parameter, function of in SqlClient connection pooling, 70
minExclusive constraining facet, function of, 146
minInclusive constraining facet, function of, 146
minLength constraining facet, function of, 146
MissingMappingAction property, settings, 211
MissingSchemaAction argument, function of, 310
MissingSchemaAction enumeration, values table, 311
MissingSchemaAction property, value settings, 211
mixed attribute, using with <complexType> XSD element, 147Ð148
MonkeyTrigger, deploying after compiling and building, 507
msdata namespace, attributes to control the
MSDTC (Microsoft Distributed Transaction Coordinator). See also Distributed Transaction Coordinator (DTC)
  events that cause transaction escalation to, 402Ð403
  used by large applications where more than one database is involved, 357
MSDTC transactions, pros and cons of using, 409
Multiple Active Resultsets (MARS). See MARS (Multiple Active Resultsets)
multiple result sets, support for by data readers and DataSets, 516
myfile.udl file, code example of content, 61Ð62

N

NameRow class, code for creating for SqlServerTVF project, 476
Namespaces property, for SqlXmlCommand object, 448
NameSplitter class
  creating for SqlServerTVF project, 477
  implementing, 478Ð479
Nested property
  contents of DataSet with it set to false, 132Ð133
  contents of DataSet with it set to true, 133Ð134
nested transactions, initiating, 389
.NET code, enabling execution of inside SQL Server, 467
.NET data providers
  specific implementations for a specific database referred to as, 7
  third party, 12Ð14
  and where they fit into the bigger picture, 7
.NET Framework
  kinds of serialization in, 135
  SqlBulkCopy class introduced in, 275
  System.XML namespace built into it, 413
.NET Framework 1.x, specifying connection string as appSetting element in, 62
.NET Framework 2.0
  code for configuration file with encrypted section, 63
  connection string security enhancements in, 62Ð63
network latency, negative effect of on large applications, 524Ð525
new database reference
  creating in the New Database Reference dialog box, 469
  warning dialog box when adding a new, 470

New Database Reference dialog box, creating a new database reference in, 469
Newton, Sir Isaac, 15
NOCOUNT setting, using inside triggers, 333
non-context connections
  opening by specifying a full connection string, 510
  resolving problems with, 509Ð510
  using inside SQLCLR, 508Ð511
None value, System.Data.Rule enumerator, 116
non-repeatable read, in reference to isolation levels, 370
null values, working with nullable columns in the database, 331Ð332
nullValue attribute, of codegen namespace, 169Ð170

O

object-oriented representation vs. relational representation, 99Ð107
Open method, for opening a connection to the database, 67
OPENXML, using with ADO.NET, 441Ð443
OPENXML function
  defined showing optional parameters, 437
  parameters to, 438
  script to create various stored procedures using, 438Ð439
  in SQL Server 2005, 414
  of SQL ServerÕs T-SQL, 437Ð443
  using in conjunction with a SQL DELETE operation, 440
  utilization of in the consumption of XML, 414
OPENXML stored procedures, Deletes and Updates, 440Ð443
openXMLCommand, creating a parameter for the Parameters collection, 442
OpenXMLSP.sql script, downloaded from Apress Source Code section, 438
optimistic concurrency
  check all columns before an update, 326Ð327
  checking only modified columns and primary key before an update, 327Ð328
  last-in wins, 325Ð326
  various options in, 325Ð330
Oracle
  code for returning multiple result sets in a stored procedure, 97Ð98
  writing Exercise 9.4 in, 293Ð294
Oracle 9i, strong support for transactions provided by, 357
Oracle sequence
  creating, 293
OracleCommand object
  specifying parameters to, 21
  Transaction and Connection properties, 20
OracleConnection class, establishing a connection, 17Ð19
OracleConnectionStringBuilder class, for generating provider-specific connection strings, 57
OracleDataAdapter, structure of in ADO.NET, 178Ð179
OracleDataReader, inheriting from DbDataReader, 25
OracleType.Cursor type, specifying parameters of to SelectCommand on the OracleDataAdapter, 192
ORDER BY clause, example using, 428
Original constant, for DataRowVersion enumeration, 282
OutputEncoding property, for SqlXmlCommand object, 448
Overflow, defined, 437

P

Parameter object, function of in ADO.NET, 5
parameterized queries, advantages of, 20Ð21
parameters
  specifying to a command, 19Ð21
  specifying to the SqlCommand and OracleCommand objects, 21
  specifying using the ConnectionString property of SqlConnection object, 54Ð55
  syntax for adding to the SqlXmlCommand object query, 450
parent rows, finding for all identified rows to get product details, 231
Parse method, used to perform a translation between a scalar type and the UDT, 104
Passthrough setting, for MissingMappingAction property of the data adapter, 211
PATH mode, function of in FOR XML clause, 415
pattern constraining facet, function of, 146
per-column encoding, subcomponents that make up the explicit declaration of a column, 424
performance, effect of transactions on, 409
Performance Monitor, verifying number of connections being established to the database in, 72
Permission Level property, setting the appropriate for SqlTVFExternalAccess, 486

Person table, SQL query for getting names of all people with the ÒReviewerÓ role, 487
Person.Contact table, schema used to update in Exercise 12.7, 457
pessimistic concurrency, drawbacks of locking other users out of database, 323Ð325
pessimistic locking. See alsœexclusive lock
  in ADO.NET IsolationLevel enumeration, 371
PetBelonging.PetBelongingID, set up as primary key in Exercise 10.1, 335
PetID= property, in the Pets table, 129
PetName property, in the Pets table, 129
Pets table, properties in, 129
petsData
  code for data binding, 131
  code for setting up a Schema for, 130
Pets.PetID, set up as primary key in Exercise 10.1, 335
phantom read, in reference to isolation levels, 371
Pipe property, of SqlPipe data type provided by the SqlContext object, 496
Pooling parameter, for enabling or disabling pooling in SqlClient connection pooling, 70
primary key
  setting a DataTableÕs to an array of DataColumns, 116
  specified in the DataSet schema in Exercise 7.4, 203
primary-key conflicts, avoiding by using GUIDs, 322Ð323
PrimaryKey property, setting a DataTableÕs to an array of DataColumns, 116
primitive data types, providing enumerations for, 144Ð145
Products table, populating sample rows for, 364
productsTable
  code examples for filling, 117
  code for in XML, 119Ð120
promotable enlistment
  function of, 402
  a quick primer, 401Ð403
Proposed constant, for DataRowVersion enumeration, 282
ProposedValue property, for DataColumnChangeEventArgs object, 120
provider model, supported by ADO.NET, 7Ð8
ProviderBase model
  putting to work, 11Ð12
  using, 9Ð12
provider-specific connection strings,

Q

quotation mark (Ò) character, entity encoding representation for, 426

R

Rammer, Ingo and Mario Szpuszta, Advanced .NET Remoting, Second Edition (Apress, 2005) by, 135
RAW mode, function of in FOR XML clause, 415
RDBMS transactions, pros and cons of using, 409
ReadCommitted value, possible in ADO.NET IsolationLevel enumeration, 371
ReadUncommitted isolation level, dirty read demonstration using, 376
ReadUncommitted value, possible in ADO.NET IsolationLevel enumeration, 371
real-world example
  of editing disconnected data, 283Ð294
  using stored procedures, 285
REF CURSORS, code for creating a stored procedure that returns multiple output, 97Ð98
reference chapter, for the ADO.NET object model, 15Ð30
RegionDelete stored procedure
  setting the openXMLCommandÕs text to, 443
  using OPENXML in conjunction with, 440
RegionInsert stored procedure
  adding to the test database, 440
  steps involved in creation of, 439Ð440
RegionUpdate stored procedure
  demonstrating use of, 442
  included in OpenXMLSP.sql, 440
  using an XML document for data used to update Region table, 440Ð441
relational data, 125Ð128
relational representation vs. object-oriented representation, 99Ð107
Relations property, in DataSet object model, 112
Relationship element, function of in msdata namespace, 170Ð171
Remoting, book for good discussion on, 135
RemotingFormat property, backward compatibility with ADO.NET 1.1, 138
Remove/RemoveAT methods, deleting existing DataRows from a DataTable with, 281Ð282
RepeatableRead value, possible in ADO.NET IsolationLevel enumeration, 372

resource managers (RMs), as important players in distributed transactions, 392Ð393
<restriction> tag, creating with a base attribute within a <simpleType> element, 145
result sets
  fetching an entire, 85
  querying for storage, 88Ð91
  querying large ones asynchronously, 91Ð96
  querying the database for multiple, 96Ð99
  reading, 22
  results of iterating through, 99
  results of using ExecuteReader method to iterate through, 87
  retrieving, 83Ð88
RMs. Seeresource managers (RMs)
Rollback method
  use of in transactions, 359
  used with the savepoint name as a parameter, 386Ð389
ROOT keyword, used for wrapping entire output in one node named XML, 431Ð432
RootTag property, for SqlXmlCommand object, 448
Row property
  for DataColumnChangeEventArgs object, 120
  for DataRowChangeEventArgs object, 121
  for TableNewRow object, 121
row states
  changes after they are examined in Exercise 9.1, 267
  checking for new row added to a DataTable, 271
  final after a successful update in Exercise 9.3, 274
  for a modified DataTable, 271
  writing a Console application for testing, 268Ð274
RowChanged event, function of, 120
RowChanging event
  behavior of vs. ColumnChanging event, 125
  function of, 120
RowDeleted event, function of, 120
RowDeleting event, function of, 120
RowFilter property, function of on the DataView object, 233
rowpattern parameter, to OPENXML, 438
rows
  adding new to a DataTable, 278Ð279
  building a check-in/check-out functionality of in an application, deleting existing, 281Ð283
  modifying existing, 279Ð281
  removing all from the DataTableÕs Rows collection, 281
  working with multiple being updated, 333Ð334
Rows collection, in DataTable class, 113
RowStateFilter property, function of on the DataView object, 233

S

SAFE control level, designating for your SQL Server code, 463Ð464
Save method
  for implementing savepoints in ADO.NET, 386Ð389
  use of in transactions, 359
savepoints
  defined, 386
  implementing in ADO.NET, 386Ð389
  and MARS, 383
saving, hierarchical data, 347Ð350
scalar value, retrieving from your database, 78Ð83
SchemaOnly command behavior, for ExecuteReader method, 86
SchemaPath property, for SqlXmlCommand object, 448
Schemas, annotating, 150Ð151
SCOPE_IDENTITY(), working as a nonblocking sequence number generated in SQL Server, 292
Security, categories of access for managed code inside SQL Server, 100Ð101
SELECT * FROM TESTDEMO query, using ExecuteReader method to execute, 85
SELECT @@ROWCOUNT, executing to find exact number of rows modified, 333
SELECT command
  for querying your database, 78
  using to query SQL Server, 379
Select method
  selecting a number of rows from a DataTable with, 219Ð222
  using to find multiple rows matching a criterion, 220
  using to find rows and specify a sort, 220Ð221
SELECT queries, function of FOR XML clauses in, 414
<sequence> element, in <complexType> XSD element group, 149
SequentialAccess command behavior, for ExecuteReader method, 86
serializable object, defined, 110
Serializable value, possible in ADO.NET IsolationLevel enumeration, 372

Serialization, kinds of in .NET Framework, 135
SerializationFormat.XML, as only choice in ADO.NET 1.1, 140
Server Explorer dialog box, new database reference listed in, 470
server-side cursor, creating a scrollable and updatable, 324
ServiceConfig class, example for dealing with distributed transactions in .NET 1.1, 397
SetAbort, function of, 394
SetComplete, function of, 394
SetDefault value, System.Data.Rule enumerator, 116
SetNull value, System.Data.Rule enumerator, 116
shared locks, in ADO.NET IsolationLevel enumeration, 371
Show All Files button, in Visual Studio IDE, 250
<simpleType> element
  creating a <restriction> tag with a base attribute within, 145
  using in XSD to create an enumeration, 145
SingleResult command behavior, for ExecuteReader method, 86
SingleRow command behavior, for ExecuteReader method, 86
Snapshot value
  possible in ADO.NET IsolationLevel enumeration, 372
  running example with IsolationLevel set to, 377
Solution Explorer, adding a new UDF to your project in, 470Ð472
Sort property, function of on the DataView object, 233
sorting, searching, and filtering, various objects and facilities ADO.NET provides for, 213Ð245
sp_addtype stored procedures vs. User-Defined Types (UDTs), 100
SQL (Structured Query Language)
  traditional mapping methods provided by, 204Ð206
  use of for communication with a data source, 77Ð78
  using to query for UDT data, 105
SQL aliasing method vs. ADO.NET mapping method, 206
SQL DELETE operation, using OPENXML in conjunction with, 440
SQL query
  for getting names of all people with the ÒReviewerÓ role, 487
  results of getting names of all people with
SQL Server
  categories of access security for managed code inside, 100Ð101
  the CLR in, 461Ð511
  designating level of control for your code in, 463Ð464
  how CLR works on vs. on a Windows machine, 462Ð464
  native XML support, 413
  requirements for running CLR code inside, 465
  setting up a trace on to capture all queries being sent to the database, 273
  working with XML features SqlXml, 446Ð459
SQL Server 2005
  Common Table Expressions (CTE) feature in, 461
  key XML features in, 414
  registering the HandWrittenUDF assembly inside, 466
  and FOR XML PATH, 433Ð434
SQL Server CLRSee alsoSQLCLR
  different from your operating system in two major respects, 100
SQL Server database, changing isolation level from ReadCommitted to ReadUncommitted, 373Ð378
SQL Server Management Studio, possible error message from SQL query, 105
SQL Server project
  steps for creating in Visual Studio 2005, 468Ð472
  using to author code written for SQLCLR in Visual Studio 2005, 465
SQL Server Project wizard, for creating a UDT in Visual Studio 2005, 101Ð105
SqlBulkCopy application
  creating the datareader for, 276Ð277
  steps for creating, 276Ð278
SqlBulkCopy class
  introduced in .NET Framework, 275
  moving large amount of data between databases with, 276Ð278
SqlClient
  connection pooling key-value pairs, 69Ð70
  interleaved command execution with MARS, 381
  MARS when not using, 381Ð382
  turning off connection pooling, 69
SQLCLR
  appropriate use of, 462Ð464
  vs. business layer, 462
  in comparison with extended stored procedures, 464
  reliability of code vs. extended stored procedures, 464
  rules for using vs. T-SQL, 462

SqlTransaction in, 504
using non-context connection inside, 508Ð511
writing stored procedures in, 94
SQLCLR aggregate
adding a new to your project, 489
creating, 488Ð490
methods that exist for Concatenator, 489
skeleton code for in C#, 489Ð490
skeleton code for in VB.NET, 490
SQLCLR code
debugging, 472Ð475
debugging without the aid of the SQL Server project, 474Ð475
SQLCLR stored procedures
vs. extended stored procedures, 464
steps for writing, 496Ð504
using to call the aggregate written in the SqlServerAggregate project, 494
writing, 494Ð504
SQLCLR triggers
adding and modifying the code, 505Ð506
using transactions in, 504Ð507
SQLCLR UDF
code for creating GetRandomNumber, 466
creating without leveraging a SQL Server project, 465Ð467
SQLCLR vs. T-SQL vs. extended stored procedures, important points to consider when making this decision, 520Ð521
SqlCommand object
creating a new and specifying the connection information, 80
running various queries using, 367Ð368
specifying parameters to, 21
three possible ways of instantiating, 80
SqlCommand objects, Transaction and Connection properties, 20
SqlCommandBuilder object
command used to retrieve metadata from the database directly, 263
setting various commands with, 272
using, 262Ð266
SqlConnection class
establishing a connection, 17Ð19
using one of the constructors for a ready-to-use SqlConnection object, 55
SqlConnection object
creating and setting up in the Main procedure, 79Ð80
holding the context connection, 495
using to take advantage of LTM, 402
SqlConnectionStringBuilder class
for generating provider-specific connection strings, 57Ð60
SqlContext object, provided by ADO.NET for working with context inside SQL Server, 496
SqlContext.Pipe.ExecuteAndSend command, for sending results in one shot, 94
SqlContext.Pipe.Send method, using as equivalent of the T-SQL PRINT statement, 496
SqlDataAdapter
constructor overloads supported by, 189
structure of in ADO.NET, 178Ð179
SqlDataReader, inheriting from DbDataReader, 25
SqlDataReader class, using GetEnumerator method in, 88Ð89
SqlDataReader object
code for reading the results from, 86Ð87
code using to query for a result set, 85
reasons to use instead of a collection of IDataRecords, 85Ð86
SqlDataSource control, adding to the .aspx page, 36
SqlFunctionAttribute, preceding the method declaration with for marking it as a UDF, 479
SQLOLEDB
connection string needed in order to run, 382
MARS-like behavior of, 381Ð382
SqlServerAggregate project
code in for developing a user-defined aggregate called Concatenator, 488Ð490
using the SQLCLR stored procedure to call aggregate written in, 494
SqlServerStoredProc project
adding a third stored procedure to, 500Ð501
downloading from Apress Source Code section, 496
SqlServerTVF project
creating, 476
downloading code from Apress Source Code section, 475
SqlServerUDF project
choosing a connection for in the Add Database Reference dialog box, 469
downloading from Apress Source Code section, 468
SqlTransaction, in SQLCLR, 504
SqlTriggerAttribute attribute, for identifying the SQLCLR trigger, 506Ð507
SqlTVFExternalAccess, setting the appropriate permission level for, 486
SqlUserDefinedAggregateAttribute attribute, default value of the Format property of, 491
.NET, 447

T

Table property, for DataTableClearTableEventHandler, 121
TableAdapter Configuration dialog box
  Advanced Options, 251
  for configuring AnimalsTableAdapter TableAdapter, 251
  configuring the SELECT query being used to fill the testDataSet, 251
TableCleared event, function of, 120
TableClearing event, function of, 120
TableDefinition property, for identifying the structure of the output table, 480
TableMappings
  using in ADO.NET, 207
TableName property, for DataTableClearTableEventHandler, 121
TableNamespace property, for DataTableClearTableEventHandler, 121
TableNewRow event
  function of, 120
  good practical use for, 125
TableNewRow object, DataTableNewRowEventHandler received by, 121
tables
  creating a relationship between two in a DataSet, 126
  how drag and drop updating works, 254Ð262
  using drag-and-drop approach for updating, 248Ð262
Tables property, in DataSet object model, 112
Table-Valued Functions (TVFs). SeeTVFs (Table-Valued Functions)
tag values, function of in FOR XML EXPLICIT query, 424
TDS (Tabular Data Stream) protocol level
  implementing batching in ADO.NET at, 274
  why batching is implemented in and not in SQL, 275
Test Connection, for testing your new connection, 34Ð35
Test database
  creating table structure for, 179Ð180
  Demo table created in, 31Ð32
  executing against, 79Ð81
TestDataSet data source, adding to your Windows Forms application, 42
TestDemo table, for Chapter 5 examples, 79
ThreadEntryPoint method, function of, 404
Timestamp column
  specifying during table creation, 330
  using in Microsoft SQL Server database, timestamps, checking for in optimistic concurrency, 328Ð330
tools, picking the right ones for the job, 522Ð526
ToString() method
  used when you use the UDT in a SQL query, 104
  using to retrieve UDT data in a connected fashion, 105
totalDigits constraining facet, function of, 146
trace, setting up on the local SQL Server to capture queries sent to the database, 273
transaction classes
  Commit and Rollback methods of, 361Ð362
  implementation of, 361Ð362
Transaction object, function of in ADO.NET, 4
Transaction property, of DbCommand object, 19
transaction savepoints, Multiple Active Resultsets (MARS) and, 383
transactional database applications
  committing or rolling back changes, 368
  implementing transactions in, 364Ð365
  most common sequence of steps performed while developing, 363
  sample database for, 364
  transactional vs. non-transactional code updates, 369
  writing, 363Ð385
Transaction.Clone, calling, 404
Transaction.Current, using to access System.Transactions.Transaction, 403
Transaction.EnlistDurable method, use of by RMs to use durable enlistment, 402
Transaction.EnlistPromotable SinglePhase method, function of, 402
Transaction.EnlistVolatile method, use of by RMs to use volatile enlistment, 402
TransactionOption enumeration, values associated with, 394
transactions, 355Ð411
  in a connected mode, 359Ð360
  database support for, 357Ð358
  deciding which to pick, 521Ð522
  default behavior for, 410
  defined, 356Ð359
  in a disconnected mode, 360Ð361
  and effect of on performance, 409
  effect of using simultaneous ADO.NET and RDBMS, 410
  ideal characteristics of, 356Ð357
  implementing, 363Ð369
  initiating nested, 389
  judicious use of, 408Ð410

local and distributed categories of, 357Ð358
and MARS (Multiple Active Resultsets), 382Ð385
performance hit when using distributed involving MSDTC, 408Ð409
practices to follow to achieve acceptable results, 409
support for in ADO.NET, 359Ð362
and user confirmation, 410
using in SQLCLR triggers, 504Ð507
using with a DataSet and DataAdapter, 389Ð392
vocabulary, 358Ð359
TransactionScope instance
negative performance implications from wrapping context connections in, 510Ð511
use of in implementing distributed transactions in .NET 2.0, 398Ð399
traversal direction, considerations for a particular task, 515Ð516
triggers and number of rows affected, 332Ð333
try...catch...finally constructs, using to ensure your connections are disposed of properly, 75
T-SQL
vs. C# and VB.NET, 461Ð462
default language supported by SQL Server, 461Ð462
rules for using vs. SQLCLR, 462
T-SQL vs. SQLCLR vs. extended stored procedures, important points to consider when making this decision, 520Ð521
TVFs (Table-Valued Functions), 464
defined, 475
output of a simple, 484
output of a simple using an order by clause in the end, 484
script used to deploy, 483
used to display the files contained in a directory, 485Ð486
using to enumerate files in a directory, 485Ð487
writing, 475Ð487
two-phase commits, in distributed transactions, 393
type attribute, in <attribute> element in XSD, 144
typed DataSets. SeeDataSets; strongly typed DataSets
typedChild attribute, of codegen namespace, 169
typedName attribute, of codegen namespace, 169
typedParent attribute, of codegen namespace, 169
typedPlural attribute, of codegen namespace, 169

U

UDFs (User-Defined Functions), 464
creating from a registered assembly in SQL Server 2005, 467
creating new called GetRandomNumber, 466
handwritten, 465Ð467
UDTs. SeeUser-Defined Types (UDTs)
Unconsumed elements and tags, 437
UNION ALL
combining the results of multiple queries with, 425Ð426
using in hierarchical FOR XML EXPLICIT query with various directives, 429Ð431
unique constraints
creating, 154Ð155
vs. key constraints, 154Ð155
UniqueConstraint, for enforcing that the values in a column or columns should be unique, 116
UNSAFE control level, designating for your SQL Server code, 463Ð464
Unspecified type value, possible in ADO.NET IsolationLevel enumeration, 372
UP_ANIMALINSERT stored procedure
used to insert a new row and send back autogenerated values, 291
using vs. DataAdapter.RowUpdated event for adding a new row, 292Ð293
UPDATE command
examining text after changes, 265
using to manipulate database data, 78
Update(DataSet) method, for updating database with data specified in the DataSet, 456
UpdateBatchSize property, using on the SqlDataAdapter to update DataTable in Exercise 9.3, 274
UpdateComboBox method
for keeping the combo box and DataSet in sync, 195
UpdateCommand
calling for updating hierarchical data, 347
UpdateCommand.CommandText, code example showing inefficient SQL query code for, 263
UpdateGram, updating data with, 454Ð456
UpdateRule attribute, of msdata namespace, 170
Updating, hierarchical data, 347
efix, using, 455

use attribute, in <attribute> element in XSD, 144
user-defined aggregate, developing one called Concatenator, 488Ð490
User-Defined Functions (UDFs). See also UDFs (User-Defined Functions)
  adding new called NameToAscii to SQLServerTVF project, 476
User-Defined Types (UDTs)
  creating in Microsoft SQL Server 2005, 100
  pragmatic use of, 106Ð107
  querying for data using SQL, 105
  retrieving data in a connected fashion from a database, 105Ð106
  vs. sp_addtype stored procedures, 100
  in XML schema, 146
UserDefinedFunctions class, wrapping inside the HandWrittenUDF namespace, 465Ð467
userTable
  code for filling programmatically, 186Ð187
UserTable, getting contents of in a disconnected fashion, 180Ð185
using blocks, using to ensure your connections are disposed of properly, 75

V

VB.NET. See Visual Basic .NET (VB.NET)
Visual Basic .NET (VB.NET), 54
  adding code in the Form1_Load event handler in, 228
  adding the UpdateBatchSize property in, 274
  AnimalsTableAdapter.Fill method in, 255
  AnimalsTableAdapter.InitCommandCollection method in, 256
  binding the DataGridView with the right table in Exercise 7.3, 196
  BookDataSet class definition of the Books property in, 165
  changing the CommndText and dataAdapter.Fill statements in, 195
  checking row state of new row in, 271
  checking the final row states of the updated DataTable in Exercise 9.3, 274
  command text for querying for a result set, 85
  constructing the DataRelation object and adding to the Relation collection, 127
  constructor of DataViews allowing you to specify information in one code line, 233
  constructor that directly ties the DataView
  correct way to order exception-handling blocks in, 28
  creating a calculated column in, 223
  creating a data adapter to retrieve records from a database in, 205
  creating a DataView without specifying any information, 232
  creating an event handler for DataBind button click event, 187
  creating a new XmlDataDocument instance called xdd in, 242
  creating a ready-to-use SqlConnection object, 55
  creating connection objects in, 54
  data loaded, sorted, and databound to a DataGridView control in, 235
  defining a connection object in Exercise 7.5 in, 205
  DeleteCommand in, 258
  displaying the contents of a DataSet in, 200
  displaying the contents of one row in, 218
  enumerating through available providers in, 11
  filling a DataSet in for Exercise 7.4, 198
  filling a DataSet programmatically in, 191
  filling a DataSet schema in Exercise 7.4, 199
  filling a DataSet with the UserTable in, 194
  filling a DataTable and displaying the row states in, 269
  filling the productsTable in, 118
  finding child rows for a given selected row in, 229
  finding parent rows for various rows in, 230
  function of Add(DataColumn(), DataColumn()) method in, 126
  function of Add(String, DataColumn(), DataColumn()) method in, 126
  function of Add(String, DataColumn(), DataColumn(), Boolean) method in, 126
  going through rows one by one and printing values to Console, 206
  incorrect way of ordering exception-handling blocks in, 27
  indexing through BooksRow objects in strongly typed Books.Rows collection, 163
  InitAdapter in, 260Ð261
  invocation of GetChildRows() through the GetBookReviewsRows() method in, 166
  looking at the Title property of BooksRow class in, 166Ð167
  making changes to the DataTable in, 270
  modifying code to include only a subset of rows based on a condition in, 226

proving DataSet relationship has same name as <keyref> element in XSD file in, 157
putting ProviderBase model to work in, 12
refreshing the contents of XmlViewer, 243
selecting a number of rows in a DataTable in, 219
serialization and deserialization of a DataSet in, 137Ð138
setting up the DataSet in, 215
setting up the productsTable in, 117
setting various commands with SqlCommandBuilder in, 272
sorting data on FirstName column in ascending order, 235
specifying DataType property on a basic DataColumn in, 223
specifying parameters of OracleType.Cursor type to SelectCommand on the OracleDataAdapter, 192
specifying parameters using the ConnectionString property, 55
vs. T-SQL, 461Ð462
update command in CommandBuilder object, 264
using DoDataMappings method to add table mapping to the data adapter, 209
using DoDataMappings method to create the usersMapping object, 208Ð209
using Find method to identify one DataRow using a basic DataTable in, 218
using GetString method to retrieve textual representation of the UDT, 106
using Select method to find multiple rows matching a criterion in, 220
using Select method to find rows and specify a sort in, 221
using SqlDataReader object to query for a result set in, 85
using TableMappings in ADO.NET using, 207
using the Compute method in, 225
using the DoDataMappings method for declaring DataColumnMapping objects, 208
using the Fill method to fill the DataTable, 189
various methods in AnimalsTableAdapter in, 255
working with a connection object in, 56
working with a strongly typed DataSet in, 163
writing a batch file to build a strongly

Visual Studio 2005
creating a new Console Application project in, 79Ð81
creating a new website in, 33Ð39
creating a UDT in, 101Ð105
Visual Studio .NET, building strongly typed DataSets in, 160Ð167
volatile enlistment, use of by RMs, 402

W

WebBrowser control (xmlViewer), adding to the XmlDataDocument, 241
website address
for Apress Source Code section, 55
for downloading CreateDataSet class library project, 214
for downloading Microsoft Data Access Application Block, 74
for downloading SqlXml 3.0 for working in .NET 1.1, 446
for work-around for SerializationFormat.XML problem, 140
whiteSpace constraining facet, function of, 146
Windows Forms application
autogenerated code in, 44
code for the connection string in the App.Config file, 43
creating a data-driven with drag-and-drop approach, 39Ð45
creating and naming Exercise 7.2, 185
for creating Exercise 9.1, 248
example 6.3Õs user interface, 129
showing Example 6.3 in action, 132
various controls added in the component tray, 43
WITH SchemaDeclaration clause, specifying with OPENXML, 438
WITH TableName clause, specifying with OPENXML, 438
ÒWrite Code YourselfÓ approach, for ADO.NET Hello World application, 48Ð51
WriteXML method, new in .NET 2.0, 117

X

XML
and ADO.NET, 413Ð459
code for productsTable in, 119Ð120
importance of for working with disconnected model, 110
leveraging to work with disconnected data, 240Ð244
as the lingua franca of the computer world, 413
using GetXml to show DataSet contents as, 131
XML columns, reading in ADO.NET, 444Ð446