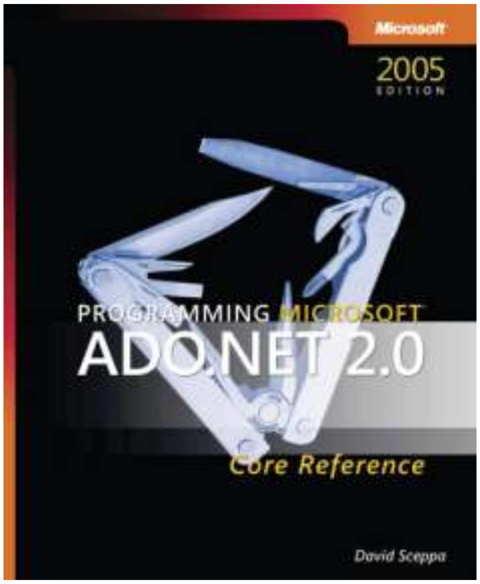

Programming Microsoft® ADO.NET 2.0 Core Reference

David Sceppa

To learn more about this book, visit Microsoft Learning at
http://www.microsoft.com/MSPress/books/8744.aspx

9780735622067
Publication Date: August 2006

Table of Contents

Chapter 1

Overview of ADO.NET

In this chapter:

ADO.NET is a set of libraries included with the Microsoft .NET Framework that help you communicate with various data stores from .NET applications. The ADO.NET libraries include classes for connecting to a data source, submitting queries, and processing results. You can also use ADO.NET as a robust, hierarchical, disconnected data cache to work with data offline. The central disconnected object, the *DataSet*, allows you to sort, search, filter, store pending changes, and navigate through hierarchical data. The *DataSet* also includes a number of features that bridge the gap between traditional data access and XML development. Developers can now work with XML data through traditional data access interfaces and vice versa.

In short, if you're building a .NET application that accesses data, you should use ADO.NET.

Microsoft Visual Studio includes a suite of design-time data access features that can help you build data access applications more efficiently. Many of these features can save you time during the development process by generating large amounts of tedious code for you. Other features improve the performance of the applications you build by storing metadata and updating logic in your code rather than fetching this information at run time. Believe it or not, many of Visual Studio's data access features accomplish both tasks.

As we examine ADO.NET throughout this book, we'll also look at features in Visual Studio that you can use to save time and effort.

No New Object Model?!?

Visual Studio 2005 breaks new ground for Microsoft in that it is the first major release of Visual Studio that does not introduce a new data access object model. (Visual Studio 2003 doesn't count because it contained only minor enhancements to Visual Studio 2002.) Developers who have honed their ADO.NET skills using versions 1.0 and 1.1 of the .NET Framework can continue to enhance those skills in version 2.0 of the .NET Framework.

Many developers who are new to the .NET Framework may have had experience with Microsoft's previous data access technology—Active Data Object (ADO). ADO served many developers well, but it lacks key features that developers need to build more powerful applications. For example, more and more developers want to work with XML data. Although later versions of ADO added XML features, ADO was not *built* to work with XML data. For example, ADO does not allow you to separate the schema information from the actual data. Microsoft might add more XML features to future releases of ADO, but ADO will never handle XML data as efficiently as ADO.NET does because ADO.NET was designed with XML in mind and ADO was not. The ADO cursor engine makes it possible to pass disconnected ADO *Recordset* objects between different tiers in your application, but you cannot combine the contents of multiple *Recordset* objects. ADO allows you to submit cached changes to databases, but it does not give you control over the logic used to submit updates. Also, the ADO cursor engine does not, for example, provide a way to submit pending changes to your database via stored procedures. Because many database administrators allow users to modify the contents of the database only through stored procedures, many developers cannot submit updates through the ADO *Recordset* object.

Microsoft built ADO.NET to address these key scenarios, along with others that I'll discuss throughout this book.

ADO.NET is designed to combine the best features of its predecessors while adding features requested most frequently by developers—greater XML support, easier disconnected data access, more control over updates, and greater update flexibility.

The ADO.NET Object Model

Now that you understand the purpose of ADO.NET and where it fits into the overall Visual Studio architecture, it's time to take a closer look at the technology. In this chapter, we'll look briefly at the ADO.NET object model and see how it differs from previous Microsoft data access technologies.

ADO.NET is designed to help developers build efficient multi-tiered database applications across intranets and the Internet, and the ADO.NET object model provides the means. Figure 1-1 shows the classes that comprise the ADO.NET object model. A dotted line separates the object model into two halves. The objects to the left of the line are connected objects. These objects communicate directly with your database to manage the connection and transactions as well as to retrieve data from and submit changes to your database. The objects to the right of the line aredisconnected objects that allow a user to work with data offline.

The objects that comprise the disconnected half of the ADO.NET object model do not communicate directly with the connected objects. This is a major change from previous Microsoft data access object models. In ADO, the *Recordset* object stores the results of your queries. You can call its *Open* method to fetch the results of a query and call its *Update* (or *UpdateBatch*) method to submit changes stored within the *Recordset* to your database.

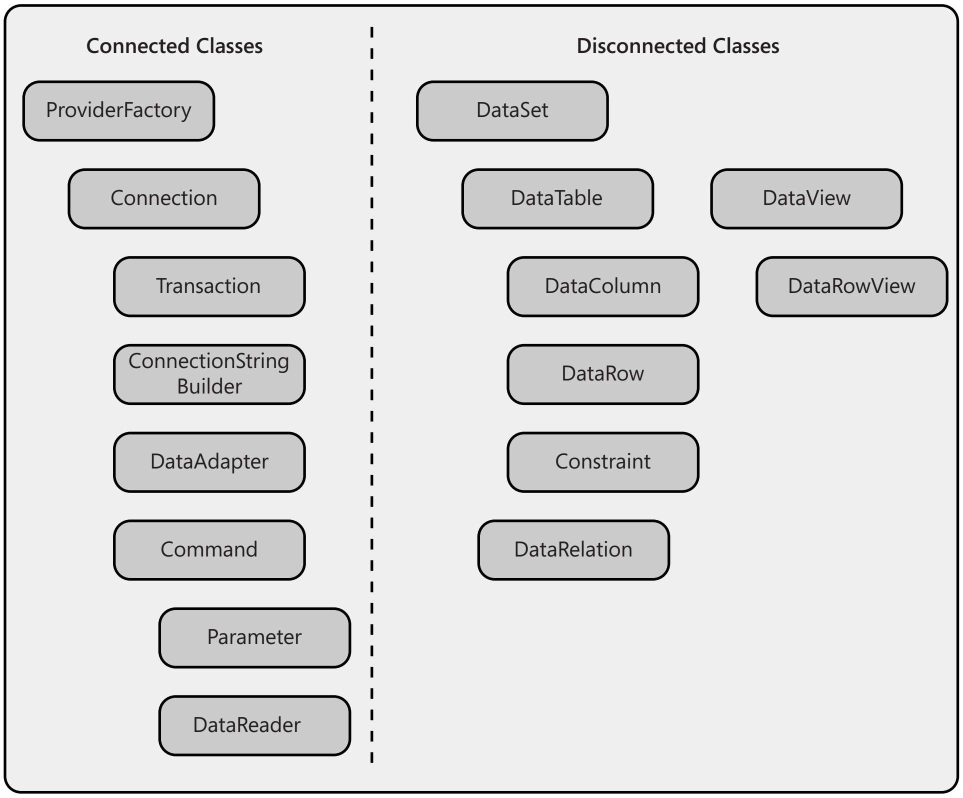

Figure 1-1 The ADO.NET object model

The ADO.NET *DataSet*, which we'll discuss shortly, is comparable in functionality to the ADO *Recordset*. However, the *DataSet* does not communicate with your database. To fetch data from your database into a *DataSet*, you pass the *DataSet* into the *Fill* method of a connected ADO.NET object—the *DataAdapter*. Similarly, to submit the pending changes stored in your *DataSet* to your database, you pass the *DataSet* to the *DataAdapter* object's *Update* method.

.NET Data Providers

A .NET data provider is a collection of classes designed to allow you to communicate with a particular type of data store. The .NET Framework includes four such providers, the SQL Client .NET Data Provider, the Oracle Client .NET Data Provider, the ODBC .NET Data Provider and the OLE DB .NET Data Provider. The SQL Client and Oracle Client .NET Data Providers are designed to talk to specific databases, SQL Server and Oracle, respectively. The ODBC and OLE DB .NET Data Providers are often called "bridge" components because they serve as a bridge to legacy technologies—ODBC and OLE DB. These providers let you communicate with various data stores through ODBC drivers and OLE DB providers, respectively.

Each .NET data provider implements the same basic classes—ProviderFactory, *Connection*, ConnectionStringBuilder, *Command*, *DataReader*, *Parameter*, and *Transaction*—although their actual names depend on the provider. For example, the SQL Client .NET Data Provider has a *SqlConnection* class, and the ODBC .NET Data Provider includes an *OdbcConnection* class. Regardless of which .NET data provider you use, the provider's *Connection* class implements the same basic features through the same basic interfaces. To open a connection to your data store, you create an instance of the provider's connection class, set the object's *ConnectionString* property, and then call its *Open* method.

Each .NET data provider has its own namespace. The four providers included in the .NET Framework are subsets of the *System.Data* namespace, where the disconnected objects reside. The SQL Client .NET Data Provider resides in the System.Data.SqlClient namespace, the ODBC .NET Data Provider resides in System.Data.Odbc, the OLE DB .NET Data Provider resides in the *System.Data.OleDb*, and the Oracle Client .NET Data Provider resides in *System.Data.OracleClient*.

Namespaces

A namespace is a logical grouping of objects. The .NET Framework is large, so to make developing applications with the .NET Framework a little easier, Microsoft has divided the objects into different namespaces. Figure 1-2 shows a portion of the hierarchy of namespaces in the .NET Framework.

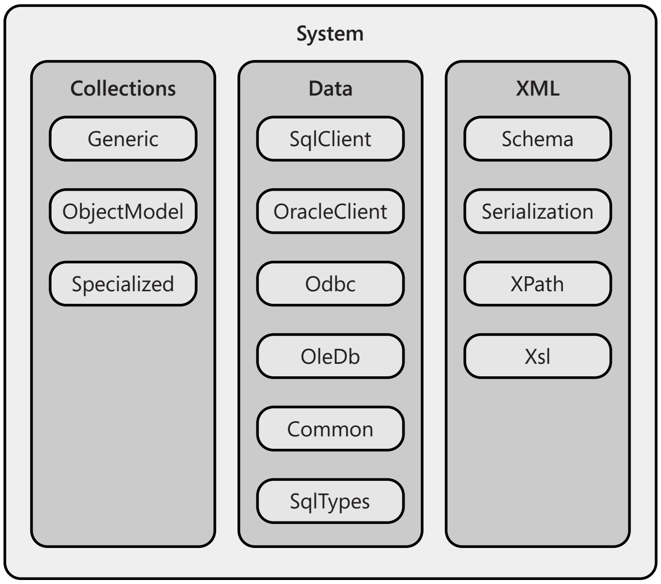

Figure 1-2 Namespaces in the .NET Framework

The most important reason for using namespaces is to prevent name collisions in assemblies. With different namespaces, programmers working on different components combined into a single solution can use the same names for different items. Because these names are separated, they don't interfere with each other at compile time. A more practical reason for namespaces is that grouping objects can make them easier to locate. Sometimes I forget the exact name of the class I'm looking for. If the classes in the .NET Framework were not broken out into smaller namespaces, I would have to find the desired class in an alphabetical list of all the classes in the framework. Thankfully, I can usually remember the namespace of the desired class. Finding the class within its namespace is simpler because there are fewer classes to examine.

For more information on using namespaces in the Microsoft .NET Framework or Visual Studio, see the .NET Framework SDK.

Because each .NET data provider implements the same basic features, the code you write will look fairly similar regardless of the provider you use. As you can see in the following code snippets, all you need to do to switch from using the ODBC .NET Data Provider to the SQL Client .NET Data Provider is to change the class you instantiate and the contents of the connection string to conform to the provider's standards.

Visual Basic

```
'Open and close an OdbcConnection
Dim cnOdbc As New OdbcConnection
cnOdbc.ConnectionString = "Driver={SQL Server};" & _
                          "Server=.\SQLExpress;" & _
                          "Database=Northwind;..."
cnOdbc.Open()
...
cnOdbc.Close()

'Open and close a SqlConnection
Dim cnSql As New SqlConnection
cnSql.ConnectionString = "Data Source=.\SQLExpress;" & _
                         "Initial Catalog=Northwind;..."
cnSql.Open()
...
cnSql.Close()
```

Visual C#

```
//Open and close an OdbcConnection
OdbcConnection cnOdbc = new OdbcConnection();
cnOdbc.ConnectionString =  "Driver={SQL Server};" +
                          @"Server=.\SQLExpress" +
                          "Database=Northwind;...";
cnOleDb.Open();
...
cnOleDb.Close();

//Open and close a SqlConnection
SqlConnection cnSql = new SqlConnection();
cnSql.ConnectionString = @"Data Source=.\SQLExpress;" +
                          "Initial Catalog=Northwind;...";
cnSql.Open();
...
cnSql.Close();
```

Why Use Separate Classes and Libraries?

No previous Microsoft data access technology has used separate libraries and classes for different data stores. Many developers have asked why Microsoft has made such a major change. There are three main reasons: performance, extensibility, and proliferation.

Better Performance

How does moving to .NET data providers improve performance? When you write ADO code, you're essentially using the ADO interfaces as middlemen when communicating with your data store. You tell ADO which provider you want to use, and ADO forwards your calls to the appropriate provider. The provider performs the requested action and returns the result to you through the ADO library.

.NET data providers don't involve a middle layer. You communicate directly with the data provider, which communicates with your data store using the data store's low-level programming interfaces. Communicating with SQL Server by using the SQL Client .NET Data Provider in ADO.NET is faster than using ADO and the SQL Server OLE DB provider because one less layer is involved.

Greater Extensibility

When SQL Server 2000 introduced XML features, the ADO development team faced an interesting challenge. To add features to ADO that would let developers retrieve XML data from SQL Server 2000, the ADO development team had to add new interfaces to the OLE DB API and to the SQL Server OLE DB provider.

.NET data providers are more easily extensible. They need to support only the same basic interfaces and can provide additional provider-specific features when appropriate. The SQL Client .NET Data Provider's *Command* object (*SqlCommand*) exposes all the same methods and properties that its OLE DB .NET Data Provider counterpart does, but it also adds a method to fetch the results of a query as XML.

SQL Server 2005 includes a suite of new features, including the ability to have applications use Notification Services to receive notifications when the results of a query change on the server. Rather than change the inner workings of ADO.NET's common classes, Microsoft simply introduced two new classes to the SQL Client .NET Data Provider to leverage these new SQL Server features.

Proliferation

Microsoft first shipped OLE DB providers for SQL Server, Microsoft Access, and Oracle with the release of the Microsoft Data Access Components (MDAC) version 2.0 in July 1998. Microsoft and other development teams have created native OLE DB providers to communicate with other data stores, but not a whole lot of OLE DB providers are available. If you're using

ADO but aren't using a Microsoft-built OLE DB provider, there's a high probability that you're using an ODBC (OLE DB's predecessor) driver instead. Many more ODBC drivers are available, primarily because they were easier to develop. Many developers simply found it too difficult to build their own OLE DB providers.

By comparison, a .NET data provider is simple to write. There are far fewer interfaces to implement. Microsoft simplified the process of building providers for ADO.NET so that developers can build .NET data providers more easily. The more .NET data providers there are, the more different data sources you can access via ADO.NET.

I wrote the previous two paragraphs for the first edition of this book. Since that time, .NET Data Providers have been developed by database vendors, independent software vendors and various open source projects. There are fewer and fewer teams producing new versions of OLE DB providers. At the time of writing the second edition of this book, most teams that are creating components to be used from Microsoft development technologies are focusing on producing .NET Data Providers, ODBC drivers, or both.

Coverage of .NET Data Providers in This Book

Because each .NET data provider implements the same base interfaces, there's no need for me to explain the use of these interfaces for every .NET data provider. Instead, I'll mostly focus on one provider: the SQL Client .NET Data Provider. In the first edition of the book, I focused on the OLE DB .NET Data Provider, but OLE DB providers have fallen out of vogue, as has the OLE DB .NET Data Provider. Plus, the SQL Client .NET Data Provider offers a ton of new features—asynchronous query execution, batch updating, query notifications, and bulk copy, to name just a few.

Appendix A discusses the features of the other three .NET data providers, as well as the common provider model. Chapter 12 addresses the SQL XML .NET Data Provider to demonstrate the use of some of ADO.NET's XML features. Because the SQL XML .NET Data Provider offers no new features and omits many classes included in other .NET data providers, it's generally not considered a full-fledged .NET data provider.

If I'm discussing a class that's common to all .NET data providers, I'll often refer to it by its provider-independent name—for example, *DataAdapter* rather than *SqlDataAdapter* or *OdbcDataAdapter*.

Connected Objects

The ADO.NET object model includes classes designed to help you communicate directly with your data source. I'll refer to such classes, which appear to the left of the dotted line in Figure 1-1 (shown earlier), as ADO.NET's "connected" classes. Most of these classes represent basic data access concepts such as the physical connection to the database, a query, and the query's results.

ProviderFactory Class

A *ProviderFactory* class is new to ADO.NET 2.0 and acts as an object factory, allowing you to create instances of other classes for your .NET data provider. Each *ProviderFactory* class offers a *Create* method that creates Connections, ConnectionStringBuilders, Commands, Parameters, DataAdapters and CommandBuilders.

Connection Class

A *Connection* object represents a connection to your data source. You can specify the type of data source, its location, and other attributes through the various properties of the *Connection* class. A *Connection* object is roughly equivalent to an ADO *Connection* object or a DAO *Database* object; you use it to connect to and disconnect from your database. A *Connection* object acts as a conduit through which other objects, such as *DataAdapter* and *Command* objects, communicate with your database to submit queries and retrieve results.

ConnectionStringBuilder Class

The *ConnectionStringBuilder* class is new to ADO.NET 2.0 and simplifies the process of building connection strings for a .NET data provider. Each *ConnectionStringBuilder* class exposes properties that correspond to options available in that .NET data provider's connection strings. For example, the *OdbcConnectionStringBuilder* class exposes a *Driver* property and the *OleDbConnectionStringBuilder* class exposes a *Provider* property. Once you've used your *ConnectionStringBuilder* to build your connection string, you can access the connection string by using the *ConnectionStringBuilder* object's *ConnectionString* property.

Command Class

Command objects are similar in structure to ADO *Command* or DAO *QueryDef* objects. They can represent a query against your database, a call to a stored procedure, or a direct request to return the contents of a specific table.

Databases support many types of queries. Some queries retrieve rows of data by referencing one or more tables or views or by calling a stored procedure. Other queries modify rows of data, and still others manipulate the structure of the database by creating or modifying objects such as tables, views, or stored procedures. You can use a *Command* object to execute any of these types of queries against your database.

Using a *Command* object to query your database is rather straightforward. You set the *Connection* property to a *Connection* object that connects to your database and then specify the text for your query in the *CommandText* property. You can supply a standard SQL query such as this one:

```
SELECT CustomerID, CompanyName, ContactName, Phone FROM Customers
```

You can also supply just the name of a table, view, or stored procedure and use the *Command* object's *CommandType* property for the type of query you want to execute. The *Command* class offers different ways to execute your query. If the query does not return rows, simply call the *ExecuteNonQuery* method. The *Command* class also has an *ExecuteReader* method, which returns a *DataReader* object that you can use to examine the rows returned by your query. If you only want to retrieve the first column of the first row returned by the query, you can save yourself a few lines of code by calling the *Command* object's *ExecuteScalar* method instead. The *SqlCommand* includes a fourth execution method, *ExecuteXmlReader*, that is similar to *ExecuteReader* but is designed to handle queries that return results in XML format.

DataReader Class

The *DataReader* class is designed to help you retrieve and examine the rows returned by your query as quickly as possible. You can use the *DataReader* class to examine the results of a query one row at a time. When you move forward to the next row, the contents of the previous row are discarded. The *DataReader* doesn't support updating. The data returned by the *DataReader* is read-only. Because the *DataReader* class supports such a minimal set of features, it's extremely fast and lightweight.

Developers with experience using cursors in previous data access technologies may recognize the DataReader as a forward-only read-only cursor, or a firehose cursor.

Transaction Class

At times, you might want to group a number of changes to your database and treat them as a single unit of work. In database programming, that unit of work is called a *transaction*. Let's say that your database contains banking information and has tables for checking and savings accounts and a user wants to transfer money from a savings account to a checking account. In your code, you'll want to make sure that the withdrawal from savings and the deposit to checking complete successfully as a single unit or that neither change occurs. You use a transaction to accomplish this.

The *Connection* class has a *BeginTransaction* method that you can use to create *Transaction* objects. You use a *Transaction* object to either commit or cancel the changes you make to your database during the lifetime of the *Transaction* object. In our banking example, the changes to both the savings and checking accounts would be included in a single transaction and, therefore, would be either committed or cancelled as a single unit of work.

Parameter Class

Say that you want to query your Orders table for all the orders for a particular customer. Your query will look something like this:

```
SELECT CustomerID, CompanyName, CompanyName, Phone FROM Customers
   WHERE CustomerID = 'ALFKI'
```

The value you use for the CustomerID column in the query's *WHERE* clause depends on the customer whose orders you want to examine. But if you use this type of query, you must modify the text for the query each time you want to examine the orders for a different customer.

To simplify the process of executing such queries, you can replace the value for the CustomerID column with a parameter marker, as shown in the following query:

```
SELECT CustomerID, CompanyName, CompanyName, Phone FROM Customers
    WHERE CustomerID = @CustomerID
```

Then, prior to executing the query, you supply a value for the parameter. Many developers rely heavily on parameterized queries because they can help simplify your programming and make for more efficient code.

To use a parameterized *Command* object, you create *Parameter* objects for each of the parameters in your query and append them to the *Command* object's *Parameters* collection. The ADO.NET *Parameter* class exposes properties and methods that let you define the data type and value for your parameters. To work with a stored procedure that returns data through output parameters, you set the *Parameter* object's *Direction* property to the appropriate value from the *ParameterDirection* enumeration.

DataAdapter Class

The *DataAdapter* class represents a new concept for Microsoft data access models; it has no true equivalent in ADO or DAO, although you can consider the ADO *Command* and DAO *QueryDef* objects to be its second cousins, once removed.

DataAdapter objects act as a bridge between your database and the disconnected objects in the ADO.NET object model. The *Fill* method, which is part of the *DataAdapter* object class, provides an efficient mechanism to fetch the results of a query into a *DataSet* or a *DataTable* so you can work with your data offline. You can also use *DataAdapter* objects to submit the pending changes stored in your *DataSet* objects to your database.

The ADO.NET *DataAdapter* class exposes a number of properties that are actually *Command* objects. For instance, the *SelectCommand* property contains a *Command* object that represents the query you'll use to populate your *DataSet* object. The *DataAdapter* class also has *UpdateCommand*, *InsertCommand*, and *DeleteCommand* properties that correspond to *Command* objects you use when you submit modified, new, or deleted rows to your database, respectively.

These *Command* objects provide updating functionality that was automatic (or "automagic," depending on your perspective) in the ADO and DAO *Recordset* objects. For example, when you run a query in ADO to generate a *Recordset* object, the ADO cursor engine asks the databases for metadata about the query to determine where the results came from. ADO then uses that metadata to build the updating logic to translate changes in your *Recordset* object into changes in your database.

So why does the ADO.NET *DataAdapter* class have separate *UpdateCommand*, *InsertCommand*, and *DeleteCommand* properties? To allow you to define your own updating logic. The updating functionality in ADO and DAO is fairly limited in the sense that both object models translate changes in *Recordset* objects into action queries that directly reference tables in your database. To maintain the security and integrity of the data, many database administrators restrict access to the tables in their databases so that the only way to change the contents of a table is to call a stored procedure. ADO and DAO don't know how to submit changes using a stored procedure; neither provides mechanisms that let you specify your own updating logic. The ADO.NET *DataAdapter* does.

With a *DataAdapter* object, you can set the *UpdateCommand*, *InsertCommand*, and *Delete-Command* properties to call the stored procedures that will modify, add, or delete rows in the appropriate table in your database. Then you can simply call the *Update* method on the *DataAdapter* object and ADO.NET will use the *Command* objects you've created to submit the cached changes in your *DataSet* to your database.

As I stated earlier, the *DataAdapter* class populates tables in the *DataSet* object and also reads cached changes and submits them to your database. To keep track of what goes where, a *DataAdapter* has some supporting properties. The *TableMappings* collection is a property used to track which table in your database corresponds to which table in your *DataSet* object. Each table mapping has a similar property for mapping columns, appropriately called a *ColumnMappings* collection.

Disconnected Classes

You've seen that you can use the connected classes in a .NET data provider to connect to a data source, submit queries, and examine their results. However, these connected classes let you examine data only as a forward-only, read-only stream of data. What if you want to sort, search, filter, or modify the results of your queries?

The ADO.NET object model includes classes to provide such functionality. These classes act as an offline data cache. Once you've fetched the results of your query into a *DataTable* (which we'll discuss shortly), you can close the connection to your data source and continue to work with the data. As mentioned earlier, because these classes do not require a live connection to your data source, we call them "disconnected" classes.

Let's look at the disconnected classes in the ADO.NET object model.

DataTable Class

The ADO.NET *DataTable* class is similar to the ADO and DAO *Recordset* classes. A *DataTable* object allows you to examine data through collections of rows and columns. You can store the results of a query in a *DataTable* by calling a *DataAdapter* object's *Fill* method, as shown in the following code snippet:

Visual Basic
```
Dim strConn, strSQL As String
strConn = "Data Source=.\SQLExpress;" & _
```

```
          "Initial Catalog=Northwind;Integrated Security=True;"
strSQL = "SELECT CustomerID, CompanyName FROM Customers"
Dim da As New SqlDataAdapter(strSQL, strConn)
Dim tbl As New DataTable()
da.Fill(tbl)
```

Visual C#

```
string strConn, strSQL;
strConn = @"Data Source=.\SQLExpress;" +
           "Initial Catalog=Northwind;Integrated Security=True;";
strSQL = "SELECT CustomerID, CompanyName FROM Customers";
SqlDataAdapter da = new SqlDataAdapter(strSQL, strConn);
DataTable tbl = new DataTable();
da.Fill(tbl);
```

Once you've fetched the data from your database and stored it in a *DataTable* object, that data is disconnected from the server. You can then examine the contents of the *DataTable* object without creating any network traffic between ADO.NET and your database. By working with the data offline, you no longer require a live connection to your database, but remember that you also won't see any changes made by other users after you've run your query.

The *DataTable* class contains collections of other disconnected objects, which I'll discuss shortly. You access the contents of a *DataTable* through its *Rows* property, which returns a collection of *DataRow* objects. If you want to examine the structure of a *DataTable*, you use its *Columns* property to retrieve a collection of *DataColumn* objects. The *DataTable* class also lets you define constraints, such as a primary key, on the data stored within the class. You can access these constraints through the *DataTable* object's *Constraints* property.

DataColumn Class

Each *DataTable* has a *Columns* collection, which is a container for *DataColumn* objects. As its name implies, a *DataColumn* object corresponds to a column in your table. However, a *DataColumn* object doesn't actually contain the data stored in your *DataTable*. Instead, it stores information about the structure of the column. This type of information, data about data, is commonly called *metadata*. For example, *DataColumn* exposes a *DataType* property that describes the data type (such as string or integer) that the column stores. The *Data-Column* class has other properties such as *ReadOnly*, *AllowDBNull*, *Unique*, *Default*, and *AutoIncrement* that allow you to control whether the data in the column can be updated, restrict what can be stored in the column, or dictate how values should be generated for new rows of data.

The *DataColumn* class also exposes an *Expression* property, which you can use to define how the data in the column is calculated. A common practice is to base a column in a query on an expression rather than on the contents of a column in a table in your database. For example, in the sample Northwind database that accompanies most Microsoft database-related products, each row in the Order Details table contains UnitPrice and Quantity columns. Traditionally, if you wanted to examine the total cost for the order item in your data structure, you would add a calculated column to the query. The following SQL example defines a calculated column called *ItemTotal*:

```
SELECT OrderID, ProductID, Quantity, UnitPrice,
       Quantity * UnitPrice AS ItemTotal
    FROM [Order Details]
```

The drawback to this technique is that the database engine performs the calculation only at the time of the query. If you modify the contents of the UnitPrice or Quantity columns in your *DataTable* object, the ItemTotal column doesn't change.

The ADO.NET *DataColumn* class defines an *Expression* property to handle this scenario more elegantly. When you check the value of a *DataColumn* object based on an expression, ADO.NET evaluates the expression and returns a newly calculated value. In this way, if you update the value of any column in the expression, the value stored in the calculated column is accurate. Here are two code snippets illustrating the use of the *Expression* property:

Visual Basic

```
Dim col As New DataColumn()
...
With col
   .ColumnName = "ItemTotal"
   .DataType = GetType(Decimal)
   .Expression = "UnitPrice * Quantity"
End With
```

Visual C#

```
DataColumn col = new DataColumn();
col.ColumnName = "ItemTotal";
col.DataType = typeof(Decimal);
col.Expression = "UnitPrice * Quantity";
```

The *Columns* collection and *DataColumn* objects can be roughly compared to the *Fields* collection and *Field* objects in ADO and DAO.

Constraint Class

The *DataTable* class also provides a way for you to place constraints on the data stored locally within a *DataTable* object. For example, you can build a *Constraint* object that ensures that the values in a column, or multiple columns, are unique within the *DataTable*. *Constraint* objects are maintained in a *DataTable* object's *Constraints* collection.

DataRow Class

To access the actual values stored in a *DataTable* object, you use the object's *Rows* collection, which contains a series of *DataRow* objects. To examine the data stored in a specific column of a particular row, use the *Item* property of the appropriate *DataRow* object to read the value for any column in that row. The *DataRow* class provides several overloaded definitions of its *Item* property. You can specify which column to view by passing the column name, index value, or associated *DataColumn* object to a *DataRow* object's *Item* property. Because *Item* is the default property of the *DataRow* class, you can use it implicitly, as shown in the following code snippets:

Visual Basic

```
Dim row As DataRow
row = tbl.Rows(0)
Console.WriteLine(row(0))
Console.WriteLine(row("CustomerID"))
Console.WriteLine(row(tbl.Columns("CustomerID")))
```

Visual C#

```
DataRow row;
row = tbl.Rows[0];
Console.WriteLine(row[0]);
Console.WriteLine(row["CustomerID"]);
Console.WriteLine(row[MyTable.Columns["CustomerID"]]);
```

Rather than return the data for just the current row, the *DataTable* makes all rows of data available through a collection of *DataRows*. This is a marked change in behavior from the ADO and DAO *Recordset* objects, which expose only a single row of data at a time, thereby requiring you to navigate through its contents by using methods such as *MoveNext*. The following code snippet is an example of looping through the contents of an ADO *Recordset*:

Visual Basic "Classic"

```
Dim strConn As String, strSQL As String
Dim rs As ADODB.Recordset
strConn = "Provider=SQLOLEDB;Data Source=.\SQLExpress;" & _
          "Initial Catalog=Northwind;Integrated Security=SSPI;"
strSQL = "SELECT CustomerID, CompanyName FROM Customers"
Set rs = New ADODB.Recordset
rs.CursorLocation = adUseClient
rs.Open strSQL, strConn, adOpenStatic, adLockReadOnly, adCmdText
Do While Not rs.EOF
    Debug.Print rs("CustomerID")
    rs.MoveNext
Loop
```

To examine the contents of an ADO.NET *DataTable*, you loop through the *DataRow* objects contained in the *DataTable* object's *Rows* property, as shown in the following code snippet:

Visual Basic

```
Dim strSQL, strConn As String
strConn = "Data Source=.\SQLExpress;" & _
          "Initial Catalog=Northwind;Integrated Security=True;"
strSQL = "SELECT CustomerID, CompanyName FROM Customers"
Dim da As New SqlDataAdapter(strSQL, strConn)
Dim tbl As New DataTable()
da.Fill(tbl)
For Each row As DataRow In tbl.Rows
    Console.WriteLine(row("CustomerID"))
Next row
```

Visual C#

```
string strSQL, strConn;
strConn = @"Data Source=.\SQLExpress;" +
          "Initial Catalog=Northwind;Integrated Security=True;";
strSQL = "SELECT CustomerID, CompanyName FROM Customers";
SqlDataAdapter da = new SqlDataAdapter(strSQL, strConn);
DataTable tbl = new DataTable();
da.Fill(tbl);
foreach (DataRow row in tbl.Rows)
    Console.WriteLine(row["CustomerID"]);
```

The *DataRow* class is also the starting point for your updates. For example, you can call the *BeginEdit* method of a *DataRow* object, change the value of some columns in that row through the *Item* property, and then call the *EndEdit* method to save the changes to that row. Calling a *DataRow* object's *CancelEdit* method lets you cancel the changes made in the current editing session. The *DataRow* class also exposes methods to delete or remove an item from the *Data-Table* object's collection of *DataRows*.

When you change the contents of a row, the *DataRow* caches those changes so that you can later submit them to your database. Thus, when you change the value of a column in a row, the *DataRow* maintains that column's original value as well as its current value to successfully update the database. The *Item* property of a *DataRow* object also allows you to examine the original value of a column when the row has a pending change.

DataSet Class

A *DataSet* object, as its name indicates, contains a set of data. You can think of a *DataSet* object as the container for a number of *DataTable* objects (stored in the *DataSet* object's *Tables* collection). Remember that ADO.NET was created to help developers build large multi-tiered database applications. At times, you might want to access a component running on a middle-tier server to retrieve the contents of many tables. Rather than having to repeatedly call the server in order to fetch that data one table at a time, you can package all the data into a *DataSet* object and return it in a single call. But a *DataSet* object does a great deal more than act as a container for multiple *DataTable* objects.

The data stored in a *DataSet* object is disconnected from your database. Any changes you make to the data are simply cached in each *DataRow*. When it's time to send these changes to your database, it might not be efficient to send the entire *DataSet* back to your middle-tier server. You can use the *GetChanges* method to extract just the modified rows from your *DataSet*. In this way, you pass less data between the different processes or servers.

The *DataSet* class also exposes a *Merge* method, which can act as a complement to the *GetChanges* method. The middle-tier server you use to submit changes to your database, using the smaller *DataSet* returned by the *Merge* method, might return a *DataSet* that contains newly retrieved data. You can use the *DataSet* class's *Merge* method to combine the contents of two *DataSet* objects into a single *DataSet*. This is another example that shows how ADO.NET was developed with multi-tiered applications in mind. Previous Microsoft data access models have no comparable feature.

You can create a *DataSet* object and populate its *Tables* collection with information without having to communicate with a database. In previous data access models, you generally need to query a database before adding new rows locally, and then later submit them to the database. With ADO.NET, you don't need to communicate with your database until you're ready to submit the new rows.

The *DataSet* class also has features that allow you to write it to and read it from a file or an area of memory. You can save just the contents of the *DataSet* object, just the structure of the *DataSet* object, or both. ADO.NET stores this data as an XML document. Because ADO.NET and XML are so tightly coupled, moving data back and forth between ADO.NET *DataSet* objects and XML documents is a snap. You can thus take advantage of one of the most powerful features of XML: its ability to easily transform the structure of your data. For example, you can use an Extensible Stylesheet Language (XSL) transformation template to convert data exported to an XML document into HTML.

DataRelation Class

The tables in your database are usually related in some fashion. For example, in the Northwind database, each entry in the Orders table relates to an entry in the Customers table, so you can determine which customer placed which orders. You'll probably want to use related data from multiple tables in your application. The ADO.NET *DataSet* class handles data from related DataTable objects with a little help from *DataRelation* class.

The DataSet class exposes a *Relations* property, which is a collection of *DataRelation* objects. You can use a *DataRelation* object to indicate a relationship between different *DataTable* objects in your *DataSet*. Once you've created your *DataRelation* object, you can use code such as the following to retrieve an array of *DataRow* objects for the orders that correspond to a particular customer:

Visual Basic

```
Dim ds As DataSet
Dim tblCustomers, tblOrders As DataTable
Dim rel As DataRelation

'The code for creating the DataSet goes here.

rel = ds.Relations.Add("Customers_Orders", _
                       tblCustomers.Columns("CustomerID"), _
                       tblOrders.Columns("CustomerID"))

For Each rowCustomer As DataRow In tblCustomers.Rows
    Console.WriteLine(rowCustomer("CompanyName"))
    For Each rowOrder As DataRow In rowCustomer.GetChildRows(rel)
        Console.WriteLine("  {0}", rowOrder("OrderID"))
    Next rowOrder
    Console.WriteLine()
Next rowCustomer
```

Visual C#

```
DataSet ds;
DataTable tblCustomers, tblOrders;
DataRelation rel;

//Create and initialize DataSet.

rel = ds.Relations.Add("Customers_Orders",
                       tblCustomers.Columns["CustomerID"],
                       tblOrders.Columns["CustomerID"]);

foreach (DataRow rowCustomer in tblCustomers.Rows) {
    Console.WriteLine(rowCustomer["CompanyName"]);
    foreach (DataRow rowOrder in rowCustomer.GetChildRows(rel))
        Console.WriteLine("  {0}", rowOrder["OrderID"]);
    Console.WriteLine();
}
```

DataRelation objects also expose properties that allow you to enforce referential integrity. For example, you can set a *DataRelation* object so that if you modify the value of the primary key field in the parent row, the change cascades down to the child rows automatically. You can also set your *DataRelation* object so that if you delete a row in one *DataTable*, the corresponding rows in any child *DataTable* objects, as defined by the relation, are automatically deleted as well.

DataView Class

Once you've retrieved the results of a query into a *DataTable* object, you can use a *DataView* object to view the data in different ways. If you want to sort the contents of a *DataTable* object based on a column, simply set the *DataView* object's *Sort* property to the name of that column. You can also set the *Filter* property on a *DataView* so that only the rows that match certain criteria are visible.

You can use multiple *DataView* objects to examine the same *DataTable* at the same time. For example, you can have two grids on a form, one showing all customers in alphabetical order, and the other showing the rows ordered by a different field, such as state or region. To show each view, you bind each grid to a different *DataView* object, but both *DataView* objects reference the same *DataTable*. This feature prevents you from having to maintain two copies of your data in separate structures. We'll discuss this in more detail in Chapter 8.

Metadata

ADO and DAO allow you to create a *Recordset* based on the results returned by your query. The data access engine examines the columns of data in the result set and populates the *Recordset* object's *Fields* collection based on this information, setting the name, data type, and so forth.

ADO.NET offers you a choice. You can use just a couple lines of code and let ADO.NET determine the structure of the results automatically, or you can use more code that includes metadata about the structure of the results of your query.

Why would you choose the option that involves writing more code? The main benefits are increased functionality and better performance. But how could having more code make your application run faster? That seems counterintuitive, doesn't it?

Unless you're writing an ad hoc query tool, you'll generally know what the structure of your query results will look like. For example, most ADO code looks something like the following:

```
Dim rs as Recordset
'Declare other variables here.
⋮
'Initialize variables and establish connection to database.
⋮
rs.Open strSQL, cnDatabase, adOpenStatic, adLockOptimistic, adCmdText
Do While Not rs.EOF
    List1.AddItem rs.Fields("UserName").Value
    rs.MoveNext
Loop
```

In this code snippet, the programmer knows that the query contains a column named UserName. The point is that as a developer, you generally know what columns your query will return and what data types those columns use. But ADO doesn't know in advance what the results of the query will look like. As a result, ADO has to query the OLE DB provider to ask questions such as "How many columns are there in the results of this query?" "What are the data types for each of those columns?" "Where did this data come from?" and "What are the primary key fields for each table referenced in this query?" The OLE DB provider can answer some of these questions, but in many cases it must call back to the database.

To retrieve the results of your query and store this data in a *DataSet* object, ADO.NET needs to know the answers to such questions. You can supply this information yourself or force ADO.NET to ask the provider for this information. Your code will run faster using the former option because asking the provider for this information at run time can result in a significant performance hit compared to supplying your own metadata through code.

Writing code to prepare the structure for your *DataSet* can become tedious, even if it improves the performance of your application. Thankfully, Visual Studio includes design-time data access features that offer the best of both worlds. For example, you can create a *DataSet* object based on a query, a table name, or a stored procedure, and then a configuration wizard will generate ADO.NET code to run the query and support submitting updates back to your database. We'll take a close look at many of these Visual Studio features in upcoming chapters.

Strongly Typed *DataSet* Classes

Visual Studio also helps you simplify the process of building data access applications by generating strongly typed *DataSet*. Let's say that we have a simple table named Orders that contains two columns, CustomerID and CompanyName. You don't have to write code such as the following.

Visual Basic

```
Dim ds As DataSet
'Create and fill DataSet.
Console.WriteLine(ds.Tables("Customers").Rows(0)("CustomerID"))
```

Visual C#

```
DataSet ds;
//Create and fill DataSet.
Console.WriteLine(ds.Tables["Customers"].Rows[0]["CustomerID"]);
```

Instead, we can write code like this:

Visual Basic

```
Dim ds As CustomersDataSet
'Create and fill DataSet.
Console.WriteLine(ds.Customers(0).CustomerID)
```

Visual C#

```
CustomersDataSet ds;
//Create and fill DataSet.
Console.WriteLine(ds.Customers[0].CustomerID);
```

The strongly typed *DataSet* is simply a class that Visual Studio builds with all the table and column information available through properties. Strongly typed *DataSet* objects also expose custom methods for such features as creating new rows. So instead of code that looks like the following:

Visual Basic

```
Dim ds as DataSet
'Code to create DataSet and customers DataTable
Dim rowNewCustomer As DataRow
rowNewCustomer = ds.Tables("Customers").NewRow()
rowNewCustomer("CustomerID") = "ALFKI"
rowNewCustomer("CompanyName") = "Alfreds Futterkiste"
ds.Tables("Customers").Rows.Add(rowNewCustomer)
```

Visual C#

```
DataSet ds;
//Code to create DataSet and customers DataTable
DataRow rowNewCustomer;
rowNewCustomer = ds.Tables["Customers"].NewRow();
rowNewCustomer["CustomerID"] = "ALFKI";
rowNewCustomer["CompanyName"] = "Alfreds Futterkiste";
ds.Tables["Customers"].Rows.Add(rowNewCustomer);
```

We can create and add a new row to our table in a single line of code, such as this:

```
ds.Customers.AddCustomersRow("ALFKI", "Alfreds Futterkiste")
```

We'll take a closer look at strongly typed *DataSet* objects in Chapter 9.

Questions That Should Be Asked More Frequently

Despite what its name implies, ADO.NET bears little resemblance to ADO. Although ADO.NET has classes that allow you to connect to your database, submit queries, and retrieve the results, the object model as a whole is very different from that of ADO. By now, you've probably picked up on many of those differences. In the coming chapters, we'll take a closer look at the main objects in the ADO.NET hierarchy. But before we do, it's worth addressing some of the questions that developers who are new to ADO.NET are likely to ask.

Q Why didn't you mention cursors?

A ADO.NET does not support server-side cursors. Future releases might include such functionality. Currently, no object in the ADO.NET hierarchy acts as an interface to a server-side cursor. The *DataSet* and *DataTable* classes most closely resemble a client-side ADO Recordset class. The *DataReader* class most closely resembles a server-side ADO *Recordset* class that uses a forward-only, read-only cursor.

Q How do I set the current position in a *DataTable* using ADO.NET? Previous object models exposed such methods as *MoveFirst* and *MoveNext*. Where are the positional properties and move methods?

A The *DataTable* class exposes a *Rows* collection (of *DataRow* objects) that you can use to reference any row in the table at any given time; therefore, the *DataTable* class has no concept of a current row. Because any row can be addressed directly, there is no need for positional properties or navigation methods such as *MoveFirst*, *MoveLast*, *MoveNext*, and *MovePrevious*.

These positional properties and move methods were used in ADO most often when displaying data on a form.