**MICROSOFT ADO.NET PROGRAMMING**

# Introduction to ADO.NET

ActiveX Data Objects for the .NET Framework (ADO.NET) is a set of classes that expose data access services to the .NET programmer. ADO.NET provides a rich set of components for creating distributed, data-sharing applications. It is an integral part of the .NET Framework, providing access to relational data, XML, and application data. ADO.NET supports a variety of development needs, including the creation of front-end database clients and middle-tier business objects used by applications, tools, languages, or Internet browsers.

# Overview of ADO.NET

ADO.NET provides consistent access to data sources such as Microsoft SQL Server, as well as data sources exposed via OLE DB and XML. Data-sharing consumer applications can use ADO.NET to connect to these data sources and retrieve, manipulate, and update data.

ADO.NET cleanly factors data access from data manipulation into discrete components that can be used separately or in tandem. ADO.NET includes .NET data providers for connecting to a database, executing commands, and retrieving results. Those results are either processed directly, or placed in an ADO.NET **DataSet** object in order to be exposed to the user in an ad-hoc manner, combined with data from multiple sources, or remoted between tiers. The ADO.NET **DataSet** object can also be used independently of a .NET data provider to manage data local to the application or sourced from XML.

## Why ADO.NET?

As application development has evolved, new applications have become loosely coupled based on the Web application model. More and more of today's applications use XML to encode data to be passed over network connections. Web applications use HTTP as the fabric for communication between tiers, and therefore must explicitly handle maintaining state between requests. This new model is very different from the connected, tightly coupled style of programming that characterized the client/server era, where a connection was held open for the duration of the program's lifetime and no special handling of state was required.

In designing tools and technologies to meet the needs of today's developer, Microsoft recognized that an entirely new programming model for data access was needed, one that is built upon the .NET Framework. Building on the .NET Framework ensured that the data access technology would be uniform—components would share a common type system, design patterns, and naming conventions.

ADO.NET was designed to meet the needs of this new programming model: disconnected data architecture, tight integration with XML, common data representation with the ability to combine data from multiple and varied data sources, and optimized facilities for interacting with a database, all native to the .NET Framework.

## Design Goals

In creating ADO.NET, Microsoft embraced several design goals.

### Leverage Current ADO Knowledge

Microsoft's design for ADO.NET addresses many of the requirements of today's application development model. At the same time, the programming model stays as similar as possible to ADO, so current ADO developers do not have to start from scratch in learning a brand new data access technology. ADO.NET is an intrinsic part of the .NET Framework without seeming completely foreign to the ADO programmer.

ADO.NET coexists with ADO. While most new .NET applications will be written using ADO.NET, ADO remains available to the .NET programmer through .NET COM interoperability services. For more information about the similarities and the differences between ADO.NET and ADO, see [Comparison between ADO and ADO.NET](cpconcomparisonbetweenadoadonet.htm).

### Support the N-Tier Programming Model

ADO.NET provides first-class support for the disconnected, n-tier programming environment for which many new applications are written. The concept of working with a disconnected set of data has become a focal point in the programming model. The ADO.NET solution for n-tier programming is the **DataSet**.

### XML Support

XML and data access are intimately tied—XML is all about encoding data, and data access is increasingly becoming all about XML. The .NET Framework does not just support Web standards—it is built entirely on top of them.

XML support is built into ADO.NET at a very fundamental level. The XML class framework in .NET and ADO.NET are part of the same architecture—they integrate at many different levels. You no longer have to choose between the data access set of services and their XML counterparts; the ability to cross over from one to the other is inherent in the design of both.

Data processing has traditionally relied primarily on a connection-based, two-tier model. As data processing increasingly uses multi-tier architectures, programmers are switching to a disconnected approach to provide better scalability for their applications.

## XML and ADO.NET

ADO.NET leverages the power of XML to provide disconnected access to data. ADO.NET was designed hand-in-hand with the .NET XML framework—both are components of a single architecture.

ADO.NET and the .NET XML framework converge in the **DataSet** object. The **DataSet** can be populated with data from an XML source, whether it is a file or an XML stream. The **DataSet** can be written as W3C compliant XML, including its schema as XML Schema Definition language (XSD) schema, regardless of the source of the data in the **DataSet**. Because the native serialization format of the **DataSet** is XML, it is an excellent medium for moving data between tiers making the **DataSet** an optimal choice for remoting data and schema context to and from a Web Service.

The **DataSet** can also be synchronized with an **XmlDataDocument** to provide relational and hierarchical access to data in real time. For more information, see [Synchronizing a DataSet with an XmlDataDocument](cpconsynchronizingdatasetwithxmldatadocument.htm).

## ADO.NET Components

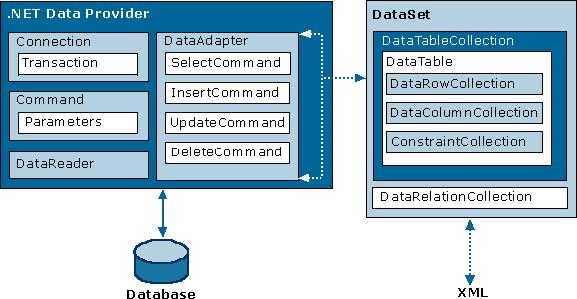
The ADO.NET components have been designed to factor data access from data manipulation. There are two central components of ADO.NET that accomplish this: the **DataSet**, and the .NET data provider, which is a set of components including the **Connection**, **Command**, **DataReader**, and **DataAdapter** objects.

The [DataSet](cpcontheadonetdataset.htm) is the core component of the disconnected architecture of ADO.NET. The **DataSet** is explicitly designed for data access independent of any data source. As a result it can be used with multiple and differing data sources, XML data, or used to manage data local to the application. The **DataSet** contains a collection of one or more **DataTable** objects made up of rows and columns of data, as well as primary key, foreign key, constraint and relation information about the data in the **DataTable** objects.

The other core element of the ADO.NET architecture is the [.NET data provider](cpconadonetproviders.htm), whose components are explicitly designed for data manipulation and fast, forward-only, read-only access to data. The **Connection** object provides connectivity to a data source. The **Command** object enables access to database commands to return data, modify data, run stored procedures, and send or retrieve parameter information. The **DataReader** provides a high-performance stream of data from the data source. Finally, the **DataAdapter** provides the bridge between the **DataSet** object and the data source. The **DataAdapter** uses **Command** objects to execute SQL commands at the data source to both load the **DataSet** with data, and reconcile changes made to the data in the **DataSet** back to the data source.You can write .NET data providers for any data source. The .NET Framework ships with two .NET data providers: the SQL Server .NET Data Provider and the OLE DB .NET Data Provider.

The following diagram illustrates the components of the ADO.NET architecture.

**Figure 1**



## ADO.NET Platform Requirements

The Microsoft .NET Framework SDK (including ADO.NET) is supported on Microsoft® Windows® 2000, Microsoft® Windows NT® 4 with Service Pack 6a, Microsoft® Windows® Millennium Edition, Microsoft® Windows® 98, Microsoft® Windows® SE, and Microsoft® Windows® 95. Use of the SQL Server .NET Data Provider or OLE DB .NET Data Provider requires the installation of Microsoft Data Access Components 2.6 or later.

To use ADO.NET, you will want to include the **System.Data** namespace in your applications.

[Visual Basic]

Imports System.Data

[C#]

using System.Data;

# ADO.NET Data Provider

A .NET data provider is used for connecting to a database, executing commands, and retrieving results. Those results are either processed directly, or placed in an ADO.NET **DataSet** in order to be exposed to the user in an ad-hoc manner, combined with data from multiple sources, or remoted between tiers. The .NET data provider is designed to be lightweight, creating a minimal layer between the data source and your code, increasing performance while not sacrificing functionality.

There are four core objects that make up a .NET data provider:

|  |  |
| --- | --- |
| **Object** | **Description** |
| **Connection** | Establishes a connection to a specific data source. |
| **Command** | Executes a command at a data source. Exposes **Parameters** and can enlist a **Transaction** from a **Connection**. |
| **DataReader** | Reads a forward-only, read-only stream of data from a data source. |
| **DataAdapter** | Populates a **DataSet** and resolves updates with the data source. |

The .NET Framework includes the SQL Server .NET Data Provider (for Microsoft SQL Server 7.0 or later), and the OLE DB .NET Data Provider.

## The SQL Server .NET Data Provider

The SQL Server .NET Data Provider uses its own protocol to communicate with SQL Server. The SQL Server .NET Data Provider is lightweight and performs well as it accesses a SQL Server data source directly without adding an OLE DB or Open Database Connectivity (ODBC) layer.

To use the SQL Server .NET Data Provider, you must have access to Microsoft SQL Server 7.0 or later. SQL Server .NET Data Provider classes are located in the **System.Data.SqlClient** namespace. For earlier versions of Microsoft SQL Server, use the OLE DB .NET Data Provider with the SQL Server OLE DB Provider (SQLOLEDB).

To use the SQL Server .NET Data Provider, you will want to include the **System.Data.SqlClient** namespace in your applications.

[Visual Basic]

Imports System.Data.SqlClient

[C#]

using System.Data.SqlClient;

The SQL Server .NET Data Provider requires the installation of MDAC 2.6 or later.

## The OLE DB .NET Data Provider

The OLE DB .NET Data Provider uses native OLE DB through COM interop to enable data access. The OLE DB .NET Data Provider supports both manual and automatic transactions. For automatic transactions, the OLE DB .NET Data Provider automatically enlists in a transaction and obtains transaction details from Windows 2000 Component Services.

To use the OLE DB .NET Data Provider, you must also use an OLE DB provider. The following providers are compatible with ADO.NET.

|  |  |
| --- | --- |
| **Driver** | **Provider** |
| SQLOLEDB | Microsoft OLE DB Provider for SQL Server |
| MSDAORA | Microsoft OLE DB Provider for Oracle |
| Microsoft.Jet.OLEDB.4.0 | OLE DB Provider for Microsoft Jet |

The OLE DB .NET Data Provider does not support OLE DB 2.5 interfaces. OLE DB Providers that require support for OLE DB 2.5 interfaces will not function properly with the OLE DB .NET Data Provider. This includes the Microsoft OLE DB Provider for Exchange and the Microsoft OLE DB Provider for Internet Publishing.

The OLE DB .NET Data Provider does not work with the OLE DB Provider for ODBC (MSDASQL).

OLE DB .NET Data Provider classes are located in the **System.Data.OleDb** namespace. To use the OLE DB .NET Data Provider, you will want to include the **System.Data.OleDb** namespace in your applications.

[Visual Basic]

Imports System.Data.OleDb

[C#]

using System.Data.OleDb;

The OLE DB .NET Data Provider requires the installation of MDAC 2.6 or later.

## Choosing a .NET Data Provider

Depending on design and data source for your application, your choice of .NET data provider can improve the performance, capability, and integrity of your application. The following table discusses the advantages and limitations of each .NET data provider.

|  |  |
| --- | --- |
| **Provider** | **Notes** |
| **SQL Server** **.NET Data Provider** | Recommended for middle-tier applications using Microsoft SQL Server 7.0 or Later.  Recommended for single-tier applications using Microsoft Data Engine (MSDE) or Microsoft SQL Server 7.0 or later.  Recommended over use of the OLE DB Provider for SQL Server (SQLOLEDB) with the OLE DB .NET Data Provider.  For Microsoft SQL Server 6.5 and earlier, you must use the OLE DB Provider for SQL Server with the OLE DB .NET Data Provider. |
| **OLE DB .NET** **Data Provider** | Recommended for middle-tier applications using Microsoft SQL Server 6.5 or earlier, or Oracle.  For Microsoft SQL Server 7.0 or later, the SQL Server .NET Data Provider is recommended.  Recommended for single-tier applications using Microsoft Access databases.  Use of the OLE DB .NET Data Provider with a Microsoft Access database for a middle-tier application is not recommended.  Support for the OLE DB Provider for ODBC (MSDASQL) is disabled. |

## Common Model

ADO.NET exposes a common model for .NET data provider objects so that a single set of code can be written to work regardless of the .NET data provider. For example, the following will work with the SQL Server .NET Data Provider or the OLE DB .NET Data Provider.

[Visual Basic]

Dim myCommand As IDbCommand = myConn.CreateCommand()

myCommand.CommandText = "SELECT \* FROM Customers"

Dim myReader As IDataReader = myCommand.ExecuteReader()

Do While myReader.Read()

Console.WriteLine("{0}" & vbTab & "{1}", myReader.GetString(0), myReader.GetString(1))

Loop

[C#]

IDbCommand myCommand = myConn.CreateCommand();

myCommand.CommandText = "SELECT \* FROM Customers";

IDataReader myReader = myCommand.ExecuteReader();

while (myReader.Read())

Console.WriteLine("{0}\t{1}", myReader.GetString(0), myReader.GetString(1))

# ADO.NET DataSet

The **DataSet** object is central to supporting disconnected, distributed data scenarios with ADO.NET. The **DataSet** is a memory-resident representation of data that provides a consistent relational programming model regardless of the data source. The **DataSet** represents a complete set of data including related tables, constraints, and relationships among the tables. The **DataSet** object model is shown below.

The methods and objects in a **DataSet** are consistent with those in the relational database model.

The **DataSet** can also persist and reload its contents as XML and its schema as XML Schema Definition language (XSD) schema.

## The DataTableCollection

An ADO.NET **DataSet** contains a collection of zero or more tables represented by **DataTable** objects. The **DataTableCollection** contains all the **DataTable** objects in a **DataSet**.

A **DataTable** is defined in the **System.Data** namespace and represents a single table of memory-resident data. It contains a collection of columns represented by the **DataColumnCollection**, which defines the schema and rows of the table. It also contains a collection of rows represented by the **DataRowCollection**, which contains the data in the table. Along with the current state, a **DataRow** retains its original state and tracks changes that occur to the data.

## The DataRelationCollection

A **DataSet** contains relationships in its **DataRelationCollection** object. A relationship, represented by the **DataRelation** object, associates rows in one **DataTable** with rows in another **DataTable**. It is analogous to a join path that might exist between primary and foreign key columns in a relational database. A **DataRelation** identifies matching columns in two tables of a **DataSet**.

Relationships enable navigation from one table to another within a **DataSet**. The essential elements of a **DataRelation** are the name of the relationship, the two tables being related, and the related columns in each table. Relationships can be built with more than one column per table, with an array of **DataColumn** objects for the key columns. When a relationship is added to the **DataRelationCollection**, it may optionally add **ForeignKeyConstraints** that disallow any changes that would invalidate the relationship.

## ExtendedProperties

The **DataSet** (as well as the **DataTable** and **DataColumn**) has an **ExtendedProperties** property. **ExtendedProperties** is a **PropertyCollection** where a user can place customized information, such as the SELECT statement that was used to generate the resultset, or a date/time stamp of when the data was generated. The **ExtendedProperties** collection is persisted with the schema information for the **DataSet** (and **DataTable** and **DataColumn**).

# Comparing ADO with ADO.NET

ADO.NET is an evolutionary improvement on ADO. One way to quickly understand the advantages of ADO.NET is to compare its features to those of ADO.

|  |  |  |
| --- | --- | --- |
| **Feature** | **ADO** | **ADO.NET** |
| Memory-resident data representation | Uses the **RecordSet** object, which looks like a single table. | Uses the **DataSet** object, which can contain one or more tables represented by **DataTable** objects. |
| Relationships between multiple tables | Requires the JOIN query to assemble data from multiple database tables in a single result table. | Supports the **DataRelation** object to associate rows in one **DataTable** object with rows in another **DataTable** object. |
| Data visitation | Scans **RecordSet** rows sequentially. | Uses a navigation paradigm for non-sequential access to rows in a table. Follows relationships to navigate from rows in one table to corresponding rows in another table. |
| Disconnected access | Provided by the **RecordSet** but typically supports connected access, represented by the **Connection** object. You communicate to a database with calls to an OLE DB provider. | Communicates to a database with standardized calls to the **DataAdapter** object, which communicates to an OLE DB provider, or directly to SQL Server. |
| Cursors | Utilizes server-side and client-side cursors. | The architecture is disconnected so cursors are not applicable. |
| Programmability | Uses the **Connection** object to transmit commands that address underlying data structure of a data source. | Uses the strongly typed programming characteristic of XML. Data is self-describing because names for code items correspond to the "real world" problem solved by the code. Underlying data constructs such as tables, rows, and columns do not appear, making code easier to read and to write. |
| Sharing disconnected data between tiers or components | Uses COM marshalling to transmit a disconnected record set. This supports only those data types defined by the COM standard. Requires type conversions, which demand system resources. | Transmits a **DataSet** as XML. The XML format places no restrictions on data types and requires no type conversions. |
| Transmitting data through firewalls | Problematic, because firewalls are typically configured to prevent system-level requests such as COM marshalling. | Supported, because ADO.NET **DataSet** objects use XML, which can pass through firewalls. |
| Scalability | Database locks and active database connections for long durations contend for limited database resources. | Disconnected access to database data without retaining database locks or active database connections for lengthy periods limits contention for limited database resources. |

# Using .NET Data Providers to access the data

A data provider in the .NET Framework serves as a bridge between an application and a data source. A data provider is used to retrieve data from a data source and to reconcile changes to that data back to the data source. ADO.NET includes two .NET data providers:

|  |  |
| --- | --- |
| **.NET Data Provider** | **Description** |
| **SQL Server .NET Data Provider** | For Microsoft® SQL Server™ 7.0 or later. |
| **OLE DB .NET Data Provider** | For data sources exposed via OLE DB. |

The **Connection**, **Command**, **DataReader**, and **DataAdapter** objects represent the core elements of the .NET data provider model. The following table describes these objects.

|  |  |
| --- | --- |
| **Object** | **Description** |
| **Connection** | Establishes a connection to a specific data source. |
| **Command** | Executes a command at a data source. |
| **DataReader** | Reads a forward-only, read-only stream of data from a data source. |
| **DataAdapter** | Populates a **DataSet** and resolves updates with the data source. |

# ADO.NET Connections

In ADO.NET you use a data **Connection** object to connect to a specific data source. To connect to Microsoft SQL Server 7.0 or later, use the **SqlConnection** object of the SQL Server .NET Data Provider. SQL Server .NET Data Provider classes are located in the **System.Data.SqlClient** namespace. To connect to an OLE DB data source, or to Microsoft SQL Server 6.*x* or earlier using the OLE DB Provider for SQL Server (SQLOLEDB), use the **OleDbConnection** object of the OLE DB .NET Data Provider.

The following examples demonstrate how to create and open connections to SQL Server (SqlClient) and OLE DB (OleDb) databases.

## SqlClient

[Visual Basic]

Dim nwindConn As SqlConnection = New SqlConnection("Data Source=localhost;Integrated Security=SSPI;" & \_

"Initial Catalog=northwind")

nwindConn.Open()

[C#]

SqlConnection nwindConn = new SqlConnection("Data Source=localhost; Integrated Security=SSPI;" +

"Initial Catalog=northwind");

nwindConn.Open();

## OleDb

[Visual Basic]

Dim nwindConn As OleDbConnection = New OleDbConnection("Provider=SQLOLEDB;Data Source=localhost;" & \_

"Integrated Security=SSPI;Initial Catalog=northwind")

nwindConn.Open()

[C#]

OleDbConnection nwindConn = new OleDbConnection("Provider=SQLOLEDB;Data Source=localhost;" +

"Integrated Security=SSPI;Initial Catalog=northwind");

nwindConn.Open();

## Connection String Format – OleDbConnection

For the OLE DB .NET Data Provider, the connection string format is identical to the connection string format used in ADO, with the following exceptions:

* The **Provider** keyword is required.
* The **URL**, **Remote Provider** and **Remote Server** keywords are not supported.

For more information about OLE DB connection strings, see "Creating the Connection String" in the Platform SDK Documentation located in the MSDN library.

**Note**   **Using Universal Data Link (UDL) files may slow performance.** You can use UDL files to supply OLE DB connection information to the OLE DB .NET Data Provider. However, because UDL files can be modified externally to any ADO.NET client program, connection strings that contain references to UDL files will be parsed every time the connection is opened. This can slow performance and it is therefore recommended, for best performance, that you use a static connection string that does not include a UDL file.

## Connection String Format – SqlConnection

The SQL Server .NET Data Provider supports a connection string format that is similar to the OLE DB (ADO) connection string format. For valid string format names and values, see the SQLConnection.ConnectionString Property.

## Closing the Connection

You must always close the **Connection** when you are finished using it. This can be done using either the **Close** or **Dispose** methods of the **Connection** object. Connections are not implicitly released when the **Connection** object falls out of scope or is reclaimed by garbage collection.

# ADO.NET Commands

After establishing a connection to a data source, you can execute commands and return results from the data source using a **Command** object. A **Command** object can be created using the **Command** constructor, or by calling the **CreateCommand** method of the **Connection** object.

When creating a **Command** object using the **Command** constructor, specify an SQL statement to execute at the data source, and a **Connection** object. The SQL statement of the **Command** object can be queried and modified using the **CommandText** property.

The **Command** object exposes several **Execute** methods you can use to perform the intended action. When returning results as a stream of data, use **ExecuteReader** to return a **DataReader** object. Use **ExecuteScalar** to return a singleton value. Use **ExecuteNonQuery** to execute commands that do not return rows.

When using the **Command** object with a stored procedure, you may set the **CommandType** property of the **Command** object to **StoredProcedure**. With a **CommandType** of **StoredProcedure**, you may use the **Parameters** property of the **Command** to access input and output parameters and return values. The **Parameters** property can be accessed regardless of the **Execute** method called. However, when calling **ExecuteReader**, return values and output parameters will not be accessible until the **DataReader** is closed.

The following example demonstrates how to format a **Command** object to return a list of **Categories** from the **Northwind** database.

## SqlClient

[Visual Basic]

Dim catCMD As SqlCommand = New SqlCommand("SELECT CategoryID, CategoryName FROM Categories", nwindConn)

[C#]

SqlCommand catCMD = new SqlCommand("SELECT CategoryID, CategoryName FROM Categories", nwindConn);

## OleDb

[Visual Basic]

Dim catCMD As OleDbCommand = New OleDbCommand("SELECT CategoryID, CategoryName FROM Categories", nwindConn)

[C#]

OleDbCommand catCMD = new OleDbCommand("SELECT CategoryID, CategoryName FROM Categories", nwindConn);

# ADO.NET DataReader

You can use the ADO.NET **DataReader** to retrieve a read-only, forward-only stream of data from a database. Using the **DataReader** can increase application performance and reduce system overhead because only one row at a time is ever in memory.

After creating an instance of the **Command** object, you create a **DataReader** by calling **Command.ExecuteReader** to retrieve rows from a data source, as in the following example.

[Visual Basic]

Dim myReader As SqlDataReader = myCommand.ExecuteReader()

[C#]

SqlDataReader myReader = myCommand.ExecuteReader();

You use the **Read** method of the **DataReader** object to obtain a row from the results of the query. You can access each column of the returned row by passing the name or ordinal reference of the column to the **DataReader**. However, for best performance, the **DataReader** provides a series of methods that allow you to access column values in their native data types (**GetDateTime**, **GetDouble**, **GetGuid**, **GetInt32**, and so on). For a list of typed accessor methods, see the OleDBDataReader Class and the SqlDataReader Class. Using the typed accessor methods when the underlying data type is known will reduce the amount of type conversion required when retrieving the column value. The following example iterates through a **DataReader** object, and returns two columns from each row.

[Visual Basic]

Do While myReader.Read()

Console.WriteLine(vbTab & "{0}" & vbTab & "{1}", myReader.GetInt32(0), myReader.GetString(1))

Loop

myReader.Close()

[C#]

while (myReader.Read())

Console.WriteLine("\t{0}\t{1}", myReader.GetInt32(0), myReader.GetString(1));

myReader.Close();

The **DataReader** provides a non-buffered stream of data that allows procedural logic to efficiently process results from a data source sequentially. The **DataReader** is a good choice when retrieving large amounts of data because the data is not cached in memory.

## Closing the DataReader

You should always call the **Close** method when you have finished using the **DataReader** object.

If your **Command** contains output parameters or return values, they will not be available until the **DataReader** is closed.

## Multiple Result Sets

If multiple result sets are returned, the **DataReader** provides the **NextResult** method to iterate through the result sets in order, as shown in the following example.

[Visual Basic]

Dim myCMD As SqlCommand = New SqlCommand("SELECT CategoryID, CategoryName FROM Categories;" & \_

"SELECT EmployeeID, LastName FROM Employees", nwindConn)

nwindConn.Open()

Dim myReader As SqlDataReader = myCMD.ExecuteReader()

Dim fNextResult As Boolean = True

Do Until Not fNextResult

Console.WriteLine(vbTab & myReader.GetName(0) & vbTab & myReader.GetName(1))

Do While myReader.Read()

Console.WriteLine(vbTab & myReader.GetInt32(0) & vbTab & myReader.GetString(1))

Loop

fNextResult = myReader.NextResult()

Loop

myReader.Close()

nwindConn.Close()

[C#]

SqlCommand myCMD = new SqlCommand("SELECT CategoryID, CategoryName FROM Categories;" +

"SELECT EmployeeID, LastName FROM Employees", nwindConn);

nwindConn.Open();

SqlDataReader myReader = myCMD.ExecuteReader();

do

{

Console.WriteLine("\t{0}\t{1}", myReader.GetName(0), myReader.GetName(1));

while (myReader.Read())

Console.WriteLine("\t{0}\t{1}", myReader.GetInt32(0), myReader.GetString(1));

} while (myReader.NextResult());

myReader.Close();

nwindConn.Close();

## Getting Schema Information from the DataReader

While a **DataReader** is open, you can retrieve schema information about the current result set using the **GetSchemaTable** method. **GetSchemaTable** returns a **DataTable** object populated with rows and columns that contain the schema information for the current result set. The **DataTable** will contain one row for each column of the result set. Each column of the schema table row maps to a property of the column returned in the result set, where the **ColumnName** is the name of the property and the value of the column is the value of the property. The following example writes out the schema information for **DataReader**.

[Visual Basic]

Dim schemaTable As DataTable = myReader.GetSchemaTable()

Dim myRow As DataRow

Dim myCol As DataColumn

For Each myRow In schemaTable.Rows

For Each myCol In schemaTable.Columns

Console.WriteLine(myCol.ColumnName & " = " & myRow(myCol).ToString())

Next

Console.WriteLine()

Next

[C#]

DataTable schemaTable = myReader.GetSchemaTable();

foreach (DataRow myRow in schemaTable.Rows)

{

foreach (DataColumn myCol in schemaTable.Columns)

Console.WriteLine(myCol.ColumnName + " = " + myRow[myCol]);

Console.WriteLine();

}

## OLE DB Chapters

Hierarchical rowsets, or chapters (OLE DB type **DBTYPE\_HCHAPTER**, ADO type **adChapter**) can be retrieved using the **OleDbDataReader**. When a query that includes a chapter is returned as a **DataReader**, the chapter is returned as a column in that **DataReader**. The chapter is exposed as a **DataReader** object.

The ADO.NET **DataSet** can also be used to represent hierarchical rowsets using parent-child relationships between tables. For more information, see [Creating and Using DataSets](cpconcreatingusingdatasets.htm).

The following example uses the MSDataShape Provider to generate a chapter column of orders for each customer in a list of customers.

[Visual Basic]

Dim nwindConn As OleDbConnection = New OleDbConnection("Provider=MSDataShape;Data Provider=SQLOLEDB;" & \_

"Data Source=localhost;Integrated Security=SSPI;Initial Catalog=northwind")

Dim custCMD As OleDbCommand = New OleDbCommand("SHAPE {SELECT CustomerID, CompanyName FROM Customers} " & \_

" APPEND ({SELECT CustomerID, OrderID FROM Orders} AS CustomerOrders " & \_

" RELATE CustomerID TO CustomerID)", nwindConn)

nwindConn.Open()

Dim custReader As OleDbDataReader = custCMD.ExecuteReader()

Dim orderReader As OleDbDataReader

Do While custReader.Read()

Console.WriteLine("Orders for " & custReader.GetString(1)) ' custReader.GetString(1) = CompanyName

orderReader = custReader.GetValue(2) ' custReader.GetValue(2) = Orders chapter as DataReader

Do While orderReader.Read()

Console.WriteLine(vbTab & orderReader.GetInt32(1)) ' orderReader.GetInt32(1) = OrderID

Loop

orderReader.Close()

Loop

custReader.Close()

nwindConn.Close()

[C#]

OleDbConnection nwindConn = new OleDbConnection("Provider=MSDataShape;Data Provider=SQLOLEDB;" +

"Data Source=localhost;Integrated Security=SSPI;Initial Catalog=northwind");

OleDbCommand custCMD = new OleDbCommand("SHAPE {SELECT CustomerID, CompanyName FROM Customers} " +

" APPEND ({SELECT CustomerID, OrderID FROM Orders} AS CustomerOrders " +

" RELATE CustomerID TO CustomerID)", nwindConn);

nwindConn.Open();

OleDbDataReader custReader = custCMD.ExecuteReader();

OleDbDataReader orderReader;

while (custReader.Read())

{

Console.WriteLine("Orders for " + custReader.GetString(1));

// custReader.GetString(1) = CompanyName

orderReader = (OleDbDataReader)custReader.GetValue(2);

// custReader.GetValue(2) = Orders chapter as DataReader

while (orderReader.Read())

Console.WriteLine("\t" + orderReader.GetInt32(1));

// orderReader.GetInt32(1) = OrderID

orderReader.Close();

}

custReader.Close();

nwindConn.Close();

# Using Stored Procedures in Commands

Stored procedures offer many advantages in data-driven applications. Database operations can be encapsulated in a single command, optimized for best performance, and enhanced with additional security. While a stored procedure can be called by simply passing the stored procedure name followed by parameter arguments as an SQL statement, using the **Parameters** collection of the ADO.NET **Command** object enables you to more explicitly define stored procedure parameters as well as to access output parameters and return values.

To call a stored procedure, set the **CommandType** of the **Command** object to **StoredProcedure**. Once the **CommandType** is set to **StoredProcedure**, you can use the **Parameters** collection to define parameters, as in the following example.

## SqlClient

[Visual Basic]

Dim nwindConn As SqlConnection = New SqlConnection("Data Source=localhost;Integrated Security=SSPI;" & \_

"Initial Catalog=northwind")

Dim salesCMD As SqlCommand = New SqlCommand("SalesByCategory", nwindConn)

salesCMD.CommandType = CommandType.StoredProcedure

Dim myParm As SqlParameter = salesCMD.Parameters.Add("@CategoryName", SqlDbType.NVarChar, 15)

myParm.Value = "Beverages"

nwindConn.Open()

Dim myReader As SqlDataReader = salesCMD.ExecuteReader()

Console.WriteLine("{0}, {1}", myReader.GetName(0), myReader.GetName(1))

Do While myReader.Read()

Console.WriteLine("{0}, ${1}", myReader.GetString(0), myReader.GetDecimal(1))

Loop

myReader.Close()

nwindConn.Close()

[C#]

SqlConnection nwindConn = new SqlConnection("Data Source=localhost;Integrated Security=SSPI;Initial Catalog=northwind");

SqlCommand salesCMD = new SqlCommand("SalesByCategory", nwindConn);

salesCMD.CommandType = CommandType.StoredProcedure;

SqlParameter myParm = salesCMD.Parameters.Add("@CategoryName", SqlDbType.NVarChar, 15);

myParm.Value = "Beverages";

nwindConn.Open();

SqlDataReader myReader = salesCMD.ExecuteReader();

Console.WriteLine("{0}, {1}", myReader.GetName(0), myReader.GetName(1));

while (myReader.Read())

{

Console.WriteLine("{0}, ${1}", myReader.GetString(0), myReader.GetDecimal(1));

}

myReader.Close();

nwindConn.Close();

## OleDb

[Visual Basic]

Dim nwindConn As OleDbConnection = New OleDbConnection("Provider=SQLOLEDB;Data Source=localhost;Integrated Security=SSPI;" & \_

"Initial Catalog=northwind")

Dim salesCMD As OleDbCommand = New OleDbCommand("SalesByCategory", nwindConn)

salesCMD.CommandType = CommandType.StoredProcedure

Dim myParm As OleDbParameter = salesCMD.Parameters.Add("@CategoryName", OleDbType.VarChar, 15)

myParm.Value = "Beverages"

nwindConn.Open()

Dim myReader As OleDbDataReader = salesCMD.ExecuteReader()

Console.WriteLine("{0}, {1}", myReader.GetName(0), myReader.GetName(1))

Do While myReader.Read()

Console.WriteLine("{0}, ${1}", myReader.GetString(0), myReader.GetDecimal(1))

Loop

myReader.Close()

nwindConn.Close()

[C#]

OleDbConnection nwindConn = new OleDbConnection("Provider=SQLOLEDB;Data Source=localhost;Integrated Security=SSPI;" +

"Initial Catalog=northwind");

OleDbCommand salesCMD = new OleDbCommand("SalesByCategory", nwindConn);

salesCMD.CommandType = CommandType.StoredProcedure;

OleDbParameter myParm = salesCMD.Parameters.Add("@CategoryName", OleDbType.VarChar, 15);

myParm.Value = "Beverages";

nwindConn.Open();

OleDbDataReader myReader = salesCMD.ExecuteReader();

Console.WriteLine("\t{0}, {1}", myReader.GetName(0), myReader.GetName(1));

while (myReader.Read())

{

Console.WriteLine("\t{0}, ${1}", myReader.GetString(0), myReader.GetDecimal(1));

}

myReader.Close();

nwindConn.Close();

A **Parameter** object can be created using the **Parameter** constructor, or by calling the **Add** method of the **Parameters** collection of a **Command**. **Parameters.Add** will take as input either constructor arguments or an existing **Parameter** object. When setting the **Value** of a **Parameter** to a null value, use **DBNull.Value**.

For parameters other than **Input** parameters, you must set the **ParameterDirection** property to specify whether the parameter type is **InputOutput**, **Output**,or **ReturnValue**. The following example shows the difference between creating **Input**, **Output**, and **ReturnValue** parameters.

## SqlClient

[Visual Basic]

Dim sampleCMD As SqlCommand = New SqlCommand("SampleProc", nwindConn)

sampleCMD.CommandType = CommandType.StoredProcedure

Dim sampParm As SqlParameter = sampleCMD.Parameters.Add("RETURN\_VALUE", SqlDbType.Int)

sampParm.Direction = ParameterDirection.ReturnValue

sampParm = sampleCMD.Parameters.Add("@InputParm", SqlDbType.NVarChar, 12)

sampParm.Value = "Sample Value"

sampParm = sampleCMD.Parameters.Add("@OutputParm", SqlDbType.NVarChar, 28)

sampParm.Direction = ParameterDirection.Output

nwindConn.Open()

Dim sampReader As SqlDataReader = sampleCMD.ExecuteReader()

Console.WriteLine("{0}, {1}", sampReader.GetName(0), sampReader.GetName(1))

Do While sampReader.Read()

Console.WriteLine("{0}, {1}", sampReader.GetInt32(0), sampReader.GetString(1))

Loop

sampReader.Close()

nwindConn.Close()

Console.WriteLine(" @OutputParm: {0}", sampleCMD.Parameters("@OutputParm").Value)

Console.WriteLine("RETURN\_VALUE: {0}", sampleCMD.Parameters("RETURN\_VALUE").Value)

[C#]

SqlCommand sampleCMD = new SqlCommand("SampleProc", nwindConn);

sampleCMD.CommandType = CommandType.StoredProcedure;

SqlParameter sampParm = sampleCMD.Parameters.Add("RETURN\_VALUE", SqlDbType.Int);

sampParm.Direction = ParameterDirection.ReturnValue;

sampParm = sampleCMD.Parameters.Add("@InputParm", SqlDbType.NVarChar, 12);

sampParm.Value = "Sample Value";

sampParm = sampleCMD.Parameters.Add("@OutputParm", SqlDbType.NVarChar, 28);

sampParm.Direction = ParameterDirection.Output;

nwindConn.Open();

SqlDataReader sampReader = sampleCMD.ExecuteReader();

Console.WriteLine("{0}, {1}", sampReader.GetName(0), sampReader.GetName(1));

while (sampReader.Read())

{

Console.WriteLine("{0}, {1}", sampReader.GetInt32(0), sampReader.GetString(1));

}

sampReader.Close();

nwindConn.Close();

Console.WriteLine(" @OutputParm: {0}", sampleCMD.Parameters["@OutputParm"].Value);

Console.WriteLine("RETURN\_VALUE: {0}", sampleCMD.Parameters["RETURN\_VALUE"].Value);

## OleDb

[Visual Basic]

Dim sampleCMD As OleDbCommand = New OleDbCommand("SampleProc", nwindConn)

sampleCMD.CommandType = CommandType.StoredProcedure

Dim sampParm As OleDbParameter = sampleCMD.Parameters.Add("RETURN\_VALUE", OleDbType.Integer)

sampParm.Direction = ParameterDirection.ReturnValue

sampParm = sampleCMD.Parameters.Add("@InputParm", OleDbType.VarChar, 12)

sampParm.Value = "Sample Value"

sampParm = sampleCMD.Parameters.Add("@OutputParm", OleDbType.VarChar, 28)

sampParm.Direction = ParameterDirection.Output

nwindConn.Open()

Dim sampReader As OleDbDataReader = sampleCMD.ExecuteReader()

Console.WriteLine("{0}, {1}", sampReader.GetName(0), sampReader.GetName(1))

Do While sampReader.Read()

Console.WriteLine("{0}, {1}", sampReader.GetInt32(0), sampReader.GetString(1))

Loop

sampReader.Close()

nwindConn.Close()

Console.WriteLine(" @OutputParm: {0}", sampleCMD.Parameters("@OutputParm").Value)

Console.WriteLine("RETURN\_VALUE: {0}", sampleCMD.Parameters("RETURN\_VALUE").Value)

[C#]

OleDbCommand sampleCMD = new OleDbCommand("SampleProc", nwindConn);

sampleCMD.CommandType = CommandType.StoredProcedure;

OleDbParameter sampParm = sampleCMD.Parameters.Add("RETURN\_VALUE", OleDbType.Integer);

sampParm.Direction = ParameterDirection.ReturnValue;

sampParm = sampleCMD.Parameters.Add("@InputParm", OleDbType.VarChar, 12);

sampParm.Value = "Sample Value";

sampParm = sampleCMD.Parameters.Add("@OutputParm", OleDbType.VarChar, 28);

sampParm.Direction = ParameterDirection.Output;

nwindConn.Open();

OleDbDataReader sampReader = sampleCMD.ExecuteReader();

Console.WriteLine("{0}, {1}", sampReader.GetName(0), sampReader.GetName(1));

while (sampReader.Read())

{

Console.WriteLine("{0}, {1}", sampReader.GetInt32(0), sampReader.GetString(1));

}

sampReader.Close();

nwindConn.Close();

Console.WriteLine(" @OutputParm: {0}", sampleCMD.Parameters["@OutputParm"].Value);

Console.WriteLine("RETURN\_VALUE: {0}", sampleCMD.Parameters["RETURN\_VALUE"].Value);

### Using Parameters with an SqlCommand

When using parameters with an **SqlCommand**, the names of the parameters added to the **SqlParameterCollection** must match the names of the parameter markers in your stored procedure. The SQL Server .NET Data Provider treats parameters in the stored procedure as named parameters and searches for the matching parameter markers.

The SQL Server .NET Data Provider does not support the question mark (?) placeholder for passing parameters to an SQL statement or a stored procedure called by a **Command** of **CommandType.Text**. In this case, you must use named parameters, as in the following example.

SELECT \* FROM Customers WHERE CustomerID = @CustomerID

### Using Parameters with an OleDbCommand

When using parameters with an **OleDbCommand**, the names of the parameters added to the **OleDbParameterCollection** must match the names of the parameter markers in your stored procedure. The OLE DB .NET Data Provider treats parameters in the stored procedure as named parameters and searches for the matching parameter markers.

The OLE DB .NET Data Provider does not support named parameters for passing parameters to an SQL statement or a stored procedure called by a **Command** of **CommandType.Text**. In this case, you must use the question mark (?) placeholder, as in the following example.

SELECT \* FROM Customers WHERE CustomerID = ?

As a result, the order in which **Parameter** objects are added to the **Parameters** collection must directly correspond to the position of the question mark placeholder for the parameter.

# Single Result Valued Commands

You may need to return information from a database that is not in the form of a table or data stream. For example, you may want to return a single value such as the result of Count(\*), Sum(Price), or Avg(Quantity). The **Command** object provides the capability to return single values using the **ExecuteScalar** method. The **ExecuteScalar** method returns the value of the first column of the first row of the result set as a scalar value.

The following example returns the number of records in a table using the **Count** aggregate function.

[Visual Basic]

Dim ordersCMD As SqlCommand = New SqlCommand("SELECT Count(\*) FROM Orders", nwindConn)

Dim count As Int32 = CInt(ordersCMD.ExecuteScalar())

[C#]

SqlCommand ordersCMD = new SqlCommand("SELECT Count(\*) FROM Orders", nwindConn);

Int32 count = (Int32)ordersCMD.ExecuteScalar();

# Performing Database Operations

Using a .NET data provider, you can execute stored procedures or data definition language (DDL) statements (for example, CREATE TABLE and ALTER COLUMN) to perform schema manipulation on a database or catalog. These commands do not return rows as a query would, so the **Command** object provides an **ExecuteNonQuery** method to process them.

In addition to using **ExecuteNonQuery** to modify schema, you can also use this method to process SQL statements that modify data but that do not return rows, such as INSERT, UPDATE, and DELETE.

Although rows are not returned by the **ExecuteNonQuery** method, input and output parameters and return values can be passed and returned via the **Parameters** collection of the **Command** object.

# Performing Catalog Operations

To execute a command to modify a database or catalog, such as the CREATE TABLE or CREATE PROCEDURE statement, create a **Command** using the appropriate Transact-SQL statement(s) and **Connection**. Execute the command with the **ExecuteNonQuery** method of the **Command** object.

The following example creates a stored procedure in a Microsoft SQL Server database.

[Visual Basic]

Dim createStr As String = "CREATE PROCEDURE InsertCategory " & \_

" @CategoryName nchar(15), " & \_

" @Identity int OUT " & \_

"AS " & \_

"INSERT INTO Categories (CategoryName) VALUES(@CategoryName) " & \_

"SET @Identity = @@Identity " & \_

"RETURN @@ROWCOUNT"

Dim createCMD As SqlCommand = New SqlCommand(createStr, nwindConn)

createCMD.ExecuteNonQuery()

[C#]

string createStr = "CREATE PROCEDURE InsertCategory " +

" @CategoryName nchar(15), " +

" @Identity int OUT " +

"AS " +

"INSERT INTO Categories (CategoryName) VALUES(@CategoryName) " +

"SET @Identity = @@Identity " +

"RETURN @@ROWCOUNT";

SqlCommand createCMD = new SqlCommand(createStr, nwindConn);

createCMD.ExecuteNonQuery();

# Modifying Data in the Database

SQL statements that modify data (such as INSERT, UPDATE, or DELETE) do not return rows. Similarly, many stored procedures perform an action but do not return rows. To execute commands that do not return rows, create a **Command** object with the appropriate SQL command and **Connection** (and any required **Parameters**), and use the **ExecuteNonQuery** method of the **Command** object.

The following example executes an INSERT statement to insert a record into a database using **ExecuteNonQuery**.

[Visual Basic]

Dim nwindConn As SqlConnection = New SqlConnection("Data Source=localhost;Integrated Security=SSPI;" & \_

"Initial Catalog=northwind")

nwindConn.Open()

Dim insertStr As String = "INSERT INTO Customers (CustomerID, CompanyName) Values('NWIND', 'Northwind Traders')"

Dim insertCMD As SqlCommand = New SqlCommand(insertStr, nwindConn)

Dim recordsAffected As Int32 = insertCMD.ExecuteNonQuery()

[C#]

SqlConnection nwindConn = new SqlConnection("Data Source=localhost;Integrated Security=SSPI;" +

"Initial Catalog=northwind");

nwindConn.Open();

string insertStr = "INSERT INTO Customers (CustomerID, CompanyName) Values('NWIND', 'Northwind Traders')";

SqlCommand insertCMD = new SqlCommand(insertStr, nwindConn);

Int32 recordsAffected = insertCMD.ExecuteNonQuery();

The following example executes the stored procedure created by the sample code in [Performing Catalog Operations](cpconperformingcatalogoperations.htm). No rows are returned by the stored procedure, so the **ExecuteNonQuery** method is used, but the stored procedure does receive an input parameter and returns an output parameter and a return value.

For the **OleDbCommand** object, the **ReturnValue** parameter must be added to the **Parameters** collection first.

[Visual Basic]

Dim insertCatCMD As SqlCommand = New SqlCommand("InsertCategory" , nwindConn)

insertCatCMD.CommandType = CommandType.StoredProcedure

Dim workParm As SqlParameter

workParm = insertCatCMD.Parameters.Add("@RowCount", SqlDbType.Int)

workParm.Direction = ParameterDirection.ReturnValue

workParm = insertCatCMD.Parameters.Add("@CategoryName", SqlDbType.NChar, 15)

workParm = insertCatCMD.Parameters.Add("@Identity", SqlDbType.Int)

workParm.Direction = ParameterDirection.Output

insertCatCMD.Parameters("@CategoryName").Value = "New Category"

insertCatCMD.ExecuteNonQuery()

Dim catID As Int32 = CInt(insertCatCMD.Parameters("@Identity").Value)

Dim rowCount As Int32 = CInt(insertCatCMD.Parameters("@RowCount").Value)

[C#]

SqlCommand insertCatCMD = new SqlCommand("InsertCategory" , nwindConn);

insertCatCMD.CommandType = CommandType.StoredProcedure;

SqlParameter workParm;

workParm = insertCatCMD.Parameters.Add("@RowCount", SqlDbType.Int);

workParm.Direction = ParameterDirection.ReturnValue;

workParm = insertCatCMD.Parameters.Add("@CategoryName", SqlDbType.NChar, 15);

workParm = insertCatCMD.Parameters.Add("@Identity", SqlDbType.Int);

workParm.Direction = ParameterDirection.Output;

insertCatCMD.Parameters["@CategoryName"].Value = "New Category";

insertCatCMD.ExecuteNonQuery();

Int32 catID = (Int32)insertCatCMD.Parameters["@Identity"].Value;

Int32 rowCount = (Int32)insertCatCMD.Parameters["@RowCount"].Value;

# Obtaining Data as XML from Data Provider

Microsoft SQL Server 2000 introduces support for XML functionality when retrieving data. To enable you to return an XML stream directly from Microsoft SQL Server 2000, the SQL Server .NET Data Provider **SqlCommand** object has the **ExecuteXmlReader** method. **ExecuteXmlReader** returns a **System.Xml.XmlReader** object populated with the results of the SQL statement specified for an **SqlCommand**. For more information about the **XmlReader**, see the XmlReader Class. **ExecuteXmlReader** can only be used with a statement that returns results as XML data, such as statements that include the SQL Server 2000 **FOR XML** clause, as shown in the following example.

[Visual Basic]

Dim custCMD As SqlCommand = New SqlCommand("SELECT \* FROM Customers FOR XML AUTO, ELEMENTS", nwindConn)

Dim myXR As System.Xml.XmlReader = custCMD.ExecuteXmlReader()

[C#]

SqlCommand custCMD = new SqlCommand("SELECT \* FROM Customers FOR XML AUTO, ELEMENTS", nwindConn);

System.Xml.XmlReader myXR = custCMD.ExecuteXmlReader();

The **DataSet** can also be used to write relational data as XML and can be synchronized with an **XmlDataDocument** to provide a real-time relational and hierarchical view of a single set of data in memory. For more information, see [Populating a DataSet from a DataAdapter](cpconpopulatingdatasetfromdataadapter.htm) and [XML and the DataSet](cpconinferringdatasetrelationalstructurefromxml.htm).

If there is no need for an in-memory relational view of the data using the **DataSet**, the **ExecuteXmlReader** method is well suited for retrieving XML data, especially for large quantities of data. Because **ExecuteXmlReader** is a streaming API, it does not have to retrieve and cache all of the data before exposing it to the caller, as would be the case if a **DataSet** were used to convert relational data into XML.

# Populating DataSet from a DataAdapter

The ADO.NET **DataSet** is a memory-resident representation of data that provides a consistent relational programming model independent of the data source. The **DataSet** represents a complete set of data including tables, constraints, and relationships among the tables. Because the **DataSet** is independent of the data source, a **DataSet** can include data local to the application, as well as data from multiple data sources. Interaction with existing data sources is controlled through the **DataAdapter**.

Each .NET data provider included with the .NET Framework has a **DataAdapter** object: the OLE DB .NET Data Provider includes an **OleDbDataAdapter** object, and the SQL Server .NET Data Provider includes an **SqlDataAdapter** object. A **DataAdapter** is used to retrieve data from a data source and populate tables and constraints within a **DataSet**. The **DataAdapter** also resolves changes made to the **DataSet** back to the data source. The **DataAdapter** uses the **Connection** object of the .NET data provider to connect to a data source, and **Command** objects to retrieve data from and resolve changes to the data source.

The **SelectCommand** property of the **DataAdapter** is a **Command** object that retrieves data from the data source. The **InsertCommand**, **UpdateCommand**, and **DeleteCommand** properties of the **DataAdapter** are **Command** objects that manage updates to the data in the data source according to modifications made to the data in the **DataSet**. These properties are covered in more detail in [Updating the Database with a DataAdapter and the DataSet](cpconupdatingdatabasewithdataadapterdataset.htm).

The **Fill** method of the **DataAdapter** is used to populate a **DataSet** with the results of the **SelectCommand** of the **DataAdapter**. **Fill** takes as its arguments a **DataSet** to be populated, and a **DataTable** object, or the name of the **DataTable** to be filled with the rows returned from the **SelectCommand**. If the **DataTable** does not exist, the **Fill** method will create a new **DataTable** in the **DataSet**.

The following example creates an instance of a **DataAdapter** that will use a **Connection** to the Microsoft SQL Server **Northwind** database and populate a **DataTable** in a **DataSet** with the list of customers. The SQL statement and **Connection** arguments passed to the **DataAdapter** constructor will be used to create the **SelectCommand** property of the **DataAdapter**.

## SqlClient

[Visual Basic]

Dim nwindConn As SqlConnection = New SqlConnection("Data Source=localhost;Integrated Security=SSPI;Initial Catalog=northwind")

Dim selectCMD As SqlCommand = New SqlCommand("SELECT CustomerID, CompanyName FROM Customers", nwindConn)

selectCMD.CommandTimeout = 30

Dim custDA As SqlDataAdapter = New SqlDataAdapter

custDA.SelectCommand = selectCMD

nwindConn.Open()

Dim custDS As DataSet = New DataSet

custDA.Fill(custDS, "Customers")

nwindConn.Close()

[C#]

SqlConnection nwindConn = new SqlConnection("Data Source=localhost;Integrated Security=SSPI;Initial Catalog=northwind");

SqlCommand selectCMD = new SqlCommand("SELECT CustomerID, CompanyName FROM Customers", nwindConn);

selectCMD.CommandTimeout = 30;

SqlDataAdapter custDA = new SqlDataAdapter();

custDA.SelectCommand = selectCMD;

nwindConn.Open();

DataSet custDS = new DataSet();

custDA.Fill(custDS, "Customers");

nwindConn.Close();

## OleDb

[Visual Basic]

Dim nwindConn As OleDbConnection = New OleDbConnection("Provider=SQLOLEDB;Data Source=localhost;" & \_

"Integrated Security=SSPI;Initial Catalog=northwind")

Dim selectCMD As OleDbCommand = New OleDbCommand("SELECT CustomerID, CompanyName FROM Customers", nwindConn)

selectCMD.CommandTimeout = 30

Dim custDA As OleDbDataAdapter = New OleDbDataAdapter

custDA.SelectCommand = selectCMD

nwindConn.Open()

Dim custDS As DataSet = New DataSet

custDA.Fill(custDS, "Customers")

nwindConn.Close()

[C#]

OleDbConnection nwindConn = new OleDbConnection("Provider=SQLOLEDB;Data Source=localhost;" +

"Integrated Security=SSPI;Initial Catalog=northwind");

OleDbCommand selectCMD = new OleDbCommand("SELECT CustomerID, CompanyName FROM Customers", nwindConn);

selectCMD.CommandTimeout = 30;

OleDbDataAdapter custDA = new OleDbDataAdapter();

custDA.SelectCommand = selectCMD;

nwindConn.Open();

DataSet custDS = new DataSet();

custDA.Fill(custDS, "Customers");

nwindConn.Close();

Notice that the **Connection** does not need to be opened until the **DataAdapter** requests data from the source in order to **Fill** the **DataSet**. Also, because the **DataSet** is independent of the data source, it requires no persistent **Connection** to the database and the **Connection** can be closed once the **DataSet** has been filled.

## Multiple Result Sets

If the **DataAdapter** encounters multiple result sets, it will create multiple tables in the **DataSet**. The tables will be given an incremental default name of Table*N*, starting with "Table" for Table0. If a table name is passed as an argument to the **Fill** method, the tables will be given an incremental default name of TableName*N*, starting with "TableName" for TableName0.

## Populating a DataSet from Multiple DataAdapters

Any number of **DataAdapters** can be used in conjunction with a **DataSet**. Each **DataAdapter** can be used to fill one or more **DataTables** and resolve updates back to the relevant data source. **DataRelations** and **Constraints** can be added to the **DataSet** locally, enabling you to relate data from multiple dissimilar data sources. For example, a **DataSet** could contain data from a Microsoft SQL Server database, an IBM DB2 database exposed via OLE DB, and a data source that streams XML. One or more **DataAdapters** would handle communication to each data source.

The following example populates a list of customers from the **Northwind** database on Microsoft SQL Server 2000, and a list of orders from the **Northwind** database stored in Microsoft Access 2000. The filled tables are related with a **DataRelation**, and the list of customers is then displayed with the orders for that customer. For more information about **DataRelations**, see [Adding a Relationship between Tables](cpconaddingrelationshipbetweentwotables.htm) and [Navigating a Relationship between Tables](cpconnavigatingrelationshipbetweentwotables.htm).

[Visual Basic]

Dim custConn As SqlConnection= New SqlConnection("Data Source=localhost;Integrated Security=SSPI;" & \_

"Initial Catalog=northwind;")

Dim custDA As SqlDataAdapter = New SqlDataAdapter("SELECT \* FROM Customers", custConn)

Dim orderConn As OleDbConnection = New OleDbConnection("Provider=Microsoft.Jet.OLEDB.4.0;" & \_

"Data Source=c:\Program Files\Microsoft Office\" & \_

"Office\Samples\northwind.mdb;")

Dim orderDA As OleDbDataAdapter = New OleDbDataAdapter("SELECT \* FROM Orders", orderConn)

custConn.Open()

orderConn.Open()

Dim custDS As DataSet = New DataSet()

custDA.Fill(custDS, "Customers")

orderDA.Fill(custDS, "Orders")

custConn.Close()

orderConn.Close()

Dim custOrderRel As DataRelation = custDS.Relations.Add("CustOrders", \_

custDS.Tables("Customers").Columns("CustomerID"), \_

custDS.Tables("Orders").Columns("CustomerID"))

Dim pRow, cRow As DataRow

For Each pRow In custDS.Tables("Customers").Rows

Console.WriteLine(pRow("CustomerID").ToString())

For Each cRow In pRow.GetChildRows(custOrderRel)

Console.WriteLine(vbTab & cRow("OrderID").ToString())

Next

Next

[C#]

SqlConnection custConn = new SqlConnection("Data Source=localhost;Integrated Security=SSPI;Initial Catalog=northwind;");

SqlDataAdapter custDA = new SqlDataAdapter("SELECT \* FROM Customers", custConn);

OleDbConnection orderConn = new OleDbConnection("Provider=Microsoft.Jet.OLEDB.4.0;" +

"Data Source=c:\\Program Files\\Microsoft Office\\Office\\Samples\\northwind.mdb;");

OleDbDataAdapter orderDA = new OleDbDataAdapter("SELECT \* FROM Orders", orderConn);

custConn.Open();

orderConn.Open();

DataSet custDS = new DataSet();

custDA.Fill(custDS, "Customers");

orderDA.Fill(custDS, "Orders");

custConn.Close();

orderConn.Close();

DataRelation custOrderRel = custDS.Relations.Add("CustOrders",

custDS.Tables["Customers"].Columns["CustomerID"],

custDS.Tables["Orders"].Columns["CustomerID"]);

foreach (DataRow pRow in custDS.Tables["Customers"].Rows)

{

Console.WriteLine(pRow["CustomerID"]);

foreach (DataRow cRow in pRow.GetChildRows(custOrderRel))

Console.WriteLine("\t" + cRow["OrderID"]);

}

## SQL Server Decimal Type

The **DataSet** stores data using .NET Framework data types. For most applications, these provide a convenient representation of data source information. However, this representation may cause a problem when the data type in the data source is a SQL Server **decimal**. The .NET Framework **decimal** data type allows a maximum of 28 significant digits, while the SQL Server **decimal** data type allows 38 significant digits. If the **SqlDataAdapter** determines, during a **Fill** operation, that the precision of a SQL Server **decimal** field is greater than 28 characters, the current row will not be added to the **DataTable**. Instead the **FillError** event will occur, which enables you to determine if a loss of precision will occur, and respond appropriately. For more information about the **FillError** event, see [Adding and Removing .NET Data Provider Events](cpconaddingremovingadonetproviderevents.htm). To get the SQL Server **decimal** value, you can also use an **SqlDataReader** object and call the **GetSqlDecimal** method.

# Setting up DataTable and DataColumn Mappings

A **DataAdapter** contains a collection of zero or more named **TableMappings**. A **TableMapping** provides a master mapping between the data returned from a query against a data source, and a **DataTable**. The **TableMapping** name can be passed in place of the **DataTable** name to the **Fill** method of the **DataAdapter**. The following example creates a **TableMapping** named AuthorsMapping for the **MyAuthors** **DataTable**.

[Visual Basic]

workAdapter.TableMappings.Add("AuthorsMapping", "MyAuthors")

[C#]

workAdapter.TableMappings.Add("AuthorsMapping", "MyAuthors");

A **TableMapping** enables you to use column names in a **DataTable** that are different from those in the database. The **DataAdapter** uses the mapping to match the columns when the table is updated.

If you do not specify a **TableName** or a **TableMapping** name when calling the **Fill** or **Update** method of the **DataAdapter**, the **DataAdapter** will look for a **TableMapping** named "Table". If that **TableMapping** does not exist, the **TableName** of the **DataTable** will be "Table". You can specify a default **TableMapping** by creating a **TableMapping** with the name of "Table".

The following example creates a **TableMapping** and makes it the default by naming it "Table". The example then maps the columns from the **Customers** table of the **Northwind** database to a set of more user-friendly names in the **DataSet**. For columns that are not mapped, the name of the column from the data source is used.

[Visual Basic]

Dim custMap As ITableMapping = custDA.TableMappings.Add("Table", "Customers")

custMap.ColumnMappings.Add( "CompanyName", "Company")

custMap.ColumnMappings.Add( "ContactName", "Contact")

custMap.ColumnMappings.Add( "PostalCode", "ZIPCode")

custDA.Fill(custDS)

[C#]

ITableMapping custMap = custDA.TableMappings.Add("Table", "Customers");

custMap.ColumnMappings.Add( "CompanyName", "Company");

custMap.ColumnMappings.Add( "ContactName", "Contact");

custMap.ColumnMappings.Add( "PostalCode", "ZIPCode");

custDA.Fill(custDS);

In more advanced situations, you may decide that you want the same **DataAdapter** to support loading different tables with different mappings. To do this, simply add additional **TableMappings**.

When the **Fill** method is passed an instance of a **DataSet** and a **TableMapping** name, if a mapping with that name exists, it is used, otherwise a **DataTable** with that name is used.

The following examples create a **TableMapping** with a name of Customers and a **DataTable** name of BizTalkSchema. The example then maps the rows returned by the SELECT statement to the BizTalkSchema **DataTable**.

[Visual Basic]

Dim bizMap As ITableMapping = custDA.TableMappings.Add("Customers", "BizTalkSchema")

bizMap.ColumnMappings.Add( "CustomerID", "ClientID")

bizMap.ColumnMappings.Add( "CompanyName", "ClientName")

bizMap.ColumnMappings.Add( "ContactName", "Contact")

bizMap.ColumnMappings.Add( "PostalCode", "ZIP")

custDA.Fill(custDS, "Customers")

[C#]

ITableMapping bizMap = custDA.TableMappings.Add("Customers", "BizTalkSchema");

bizMap.ColumnMappings.Add( "CustomerID", "ClientID");

bizMap.ColumnMappings.Add( "CompanyName", "ClientName");

bizMap.ColumnMappings.Add( "ContactName", "Contact");

bizMap.ColumnMappings.Add( "PostalCode", "ZIP");

custDA.Fill(custDS, "Customers");

**CAUTION**   If a source column name is not supplied for a column mapping or a source table name is not supplied for a table mapping, default names will be automatically generated. If no source column is supplied for a column mapping, the column mapping is given an incremental default name of SourceColumn*N,* starting with "SourceColumn1". If no source table name is supplied for a table mapping, the table mapping is given an incremental default name of SourceTable*N*, starting with "SourceTable1".

It is recommended that you avoid the naming convention of "SourceColumn*N*" when you supply a source column name for a column mapping, or "SourceTable*N*" when you supply a source table name for a table mapping, because the name you supply may conflict with an existing default column mapping name in the **ColumnMappingCollection** or table mapping name in the **TableMappingCollection**. If the supplied name already exists, an exception will be thrown.

# Input, Output Parameters and Return Values

Stored procedures can have return values in addition to input and output parameters. The sample below illustrates how ADO.NET sends and receives input parameters, output parameters, and return values using a common scenario of inserting a new record into a table where the primary key column is an autonumber field. The sample uses an output parameter to return the **@@Identity** of the autonumber field and the **DataAdapter** binds it to the column of the **DataTable** so that the **DataSet** will reflect the resulting primary key value.

The sample uses the following stored procedure to insert a new category into the **Northwind** **Categories** table, which takes the value in the **CategoryName** column as an input parameter, returns the value of the autonumber identity field, **CategoryID**, from **@@Identity** as an output parameter, and has a return value of the rows affected.

CREATE PROCEDURE InsertCategory

@CategoryName nchar(15),

@Identity int OUT

AS

INSERT INTO Categories (CategoryName) VALUES(@CategoryName)

SET @Identity = @@Identity

RETURN @@ROWCOUNT

The following example uses the **InsertCategory** stored procedure as the source for the **InsertCommand** of the **DataAdapter**. By specifying the **CategoryID** column as the **SourceColumn** for the *@Identity* output parameter, the resulting autonumber value will be reflected in the **DataSet** after the record has been inserted into the database when the **Update** method of the **DataAdapter** is called.

For the **OleDbDataAdapter**, parameters with a **ParameterDirection** of **ReturnValue** must be specified before the other parameters.

## SqlClient

[Visual Basic]

Dim nwindConn As SqlConnection = New SqlConnection("Data Source=localhost;Integrated Security=SSPI;" & \_

"Initial Catalog=northwind")

Dim catDA As SqlDataAdapter = New SqlDataAdapter("SELECT CategoryID, CategoryName FROM Categories", nwindConn)

catDA.InsertCommand = New SqlCommand("InsertCategory" , nwindConn)

catDA.InsertCommand.CommandType = CommandType.StoredProcedure

Dim myParm As SqlParameter = catDA.InsertCommand.Parameters.Add("@RowCount", SqlDbType.Int)

myParm.Direction = ParameterDirection.ReturnValue

catDA.InsertCommand.Parameters.Add("@CategoryName", SqlDbType.NChar, 15, "CategoryName")

myParm = catDA.InsertCommand.Parameters.Add("@Identity", SqlDbType.Int, 0, "CategoryID")

myParm.Direction = ParameterDirection.Output

Dim catDS As DataSet = New DataSet()

catDA.Fill(catDS, "Categories")

Dim newRow As DataRow = catDS.Tables("Categories").NewRow()

newRow("CategoryName") = "New Category"

catDS.Tables("Categories").Rows.Add(newRow)

catDA.Update(catDS, "Categories")

Dim rowCount As Int32 = CInt(catDA.InsertCommand.Parameters("@RowCount").Value)

[C#]

SqlConnection nwindConn = new SqlConnection("Data Source=localhost;Integrated Security=SSPI;" +

"Initial Catalog=northwind");

SqlDataAdapter catDA = new SqlDataAdapter("SELECT CategoryID, CategoryName FROM Categories", nwindConn);

catDA.InsertCommand = new SqlCommand("InsertCategory", nwindConn);

catDA.InsertCommand.CommandType = CommandType.StoredProcedure;

SqlParameter myParm = catDA.InsertCommand.Parameters.Add("@RowCount", SqlDbType.Int);

myParm.Direction = ParameterDirection.ReturnValue;

catDA.InsertCommand.Parameters.Add("@CategoryName", SqlDbType.NChar, 15, "CategoryName");

myParm = catDA.InsertCommand.Parameters.Add("@Identity", SqlDbType.Int, 0, "CategoryID");

myParm.Direction = ParameterDirection.Output;

DataSet catDS = new DataSet();

catDA.Fill(catDS, "Categories");

DataRow newRow = catDS.Tables["Categories"].NewRow();

newRow["CategoryName"] = "New Category";

catDS.Tables["Categories"].Rows.Add(newRow);

catDA.Update(catDS, "Categories");

Int32 rowCount = (Int32)catDA.InsertCommand.Parameters["@RowCount"].Value;

## OleDb

[Visual Basic]

Dim nwindConn As OleDbConnection = New OleDbConnection("Provider=SQLOLEDB;Data Source=localhost;" & \_

"Integrated Security=SSPI;Initial Catalog=northwind")

Dim catDA As OleDbDataAdapter = New OleDbDataAdapter("SELECT CategoryID, CategoryName FROM Categories", \_

nwindConn)

catDA.InsertCommand = New OleDbCommand("InsertCategory" , nwindConn)

catDA.InsertCommand.CommandType = CommandType.StoredProcedure

Dim myParm As OleDbParameter = catDA.InsertCommand.Parameters.Add("@RowCount", OleDbType.Integer)

myParm.Direction = ParameterDirection.ReturnValue

catDA.InsertCommand.Parameters.Add("@CategoryName", OleDbType.Char, 15, "CategoryName")

myParm = catDA.InsertCommand.Parameters.Add("@Identity", OleDbType.Integer, 0, "CategoryID")

myParm.Direction = ParameterDirection.Output

Dim catDS As DataSet = New DataSet()

catDA.Fill(catDS, "Categories")

Dim newRow As DataRow = catDS.Tables("Categories").NewRow()

newRow("CategoryName") = "New Category"

catDS.Tables("Categories").Rows.Add(newRow)

catDA.Update(catDS, "Categories")

Dim rowCount As Int32 = CInt(catDA.InsertCommand.Parameters("@RowCount").Value)

[C#]

OleDbConnection nwindConn = new OleDbConnection("Provider=SQLOLEDB;Data Source=localhost;" +

"Integrated Security=SSPI;Initial Catalog=northwind");

OleDbDataAdapter catDA = new OleDbDataAdapter("SELECT CategoryID, CategoryName FROM Categories", nwindConn);

catDA.InsertCommand = new OleDbCommand("InsertCategory", nwindConn);

catDA.InsertCommand.CommandType = CommandType.StoredProcedure;

OleDbParameter myParm = catDA.InsertCommand.Parameters.Add("@RowCount", OleDbType.Integer);

myParm.Direction = ParameterDirection.ReturnValue;

catDA.InsertCommand.Parameters.Add("@CategoryName", OleDbType.Char, 15, "CategoryName");

myParm = catDA.InsertCommand.Parameters.Add("@Identity", OleDbType.Integer, 0, "CategoryID");

myParm.Direction = ParameterDirection.Output;

DataSet catDS = new DataSet();

catDA.Fill(catDS, "Categories");

DataRow newRow = catDS.Tables["Categories"].NewRow();

newRow["CategoryName"] = "New Category";

catDS.Tables["Categories"].Rows.Add(newRow);

catDA.Update(catDS, "Categories");

Int32 rowCount = (Int32)catDA.InsertCommand.Parameters["@RowCount"].Value;

# Command Generator

If your **DataTable** maps to or is generated from a single database table, you can take advantage of the **CommandBuilder** object to automatically generate the **DeleteCommand**, **InsertCommand**, and **UpdateCommand** of the **DataAdapter**. This simplifies and reduces the code required to perform INSERT, UDPATE, and DELETE operations.

As a minimum requirement, you must set the **SelectCommand** property in order for automatic command generation to work. The table schema retrieved by the **SelectCommand** determines the syntax of the automatically generated INSERT, UPDATE, and DELETE statements.

The **SelectCommand** must also return at least one primary key or unique column. If none are present, an **InvalidOperation** exception is generated, and the commands are not generated.

The **InsertCommand**, **UpdateCommand**, and **DeleteCommand** properties are automatically generated if they are null values. If a **Command** exists for the property, the existing **Command** is used.

## Rules for Automatically Generated Commands

The following table shows the rules for how automatically generated commands are generated.

|  |  |
| --- | --- |
| **Command** | **Rule** |
| **InsertCommand** | Inserts a row at the data source for all rows in the table with a **RowState** of **DataRowState.Added**. Inserts values for all columns that are updateable (but not columns such as identities, expressions, or timestamps). |
| **UpdateCommand** | Updates rows at the data source for all rows in the table with a **RowState** of **DataRowState.Modified**. Updates the values of all columns except for non-updateable columns, such as identities or expressions. Updates all rows where the column values at the data source match the primary key column values of the row, and where the remaining columns at the data source match the original values of the row (see the section in this topic on Optimistic Concurrency Model for Updates). |
| **DeleteCommand** | Deletes rows at the data source for all rows in the table with a **RowState** of **DataRowState.Deleted**. Deletes all rows where the column values match the primary key values of the row. |

## Optimistic Concurrency Model for Updates

The logic for generating commands automatically for UPDATE statements is based on optimistic concurrency. That is, records are not locked for editing and can be modified by other users or processes at any time. Because a record could have been modified after it was returned from the SELECT statement, but before the UPDATE statement is issued, the automatically generated UPDATE statement contains a WHERE clause such that a row is only updated if it contains all original values and has not been deleted. This is done to avoid new data being overwritten. In cases where an automatically generated update attempts to update a row that has been deleted or does not contain the original values found in the **DataSet**, the command will not affect any records and a **DBConcurrencyException** will be thrown.

If you want the UPDATE to complete regardless of original values, you will need to explicitly set the **UpdateCommand** for the **DataAdapter** and not rely on automatic command generation.

## Limitations of Automatic Command Generation Logic

The following limitations apply to automatic command generation.

**Unrelated tables only**

Automatic command generation generates INSERT, UPDATE, or DELETE statements for tables that have no relationships to other tables within a **DataSet**. The automatic command generation logic relies on primary key information to generate the WHERE clause for the various statements. This is insufficient information when joined tables are involved. In this case, the user must specify the statements used to perform the update.

**Table and column names**

Automatic command generation logic fails if column names or table names contain any special characters, such as spaces, periods, quotation marks, or other non-alphanumeric characters, even if delimited by brackets. Fully qualified table names in the form of schema.owner.table are supported.

## Using the CommandBuilder to Automatically Generate an SQL Statement

To automatically generate SQL statements for a **DataAdapter**, first set the **SelectCommand** property of the **DataAdapter**. Then create a **CommandBuilder** object and specify as an argument the **DataAdapter** for which the **CommandBuilder** will automatically generate SQL statements.

[Visual Basic]

Dim custDA As SqlDataAdapter = New SqlDataAdapter("SELECT \* FROM Customers", nwindConn)

Dim custCB As SqlCommandBuilder = New SqlCommandBuilder(custDA)

nwindConn.Open()

Dim custDS As DataSet = New DataSet

custDA.Fill(custDS, "Customers")

' Code to modify data in DataSet here.

' Without the SqlCommandBuilder, this line would fail.

custDA.Update(custDS, "Customers")

nwindConn.Close()

[C#]

SqlDataAdapter custDA = new SqlDataAdapter("SELECT \* FROM Customers", nwindConn);

SqlCommandBuilder custCB = new SqlCommandBuilder(custDA);

nwindConn.Open();

DataSet custDS = new DataSet();

custDA.Fill(custDS, "Customers");

// Code to modify data in DataSet here.

// Without the SqlCommandBuilder, this line would fail.

custDA.Update(custDS, "Customers");

nwindConn.Close();

## Modifying the SelectCommand

If you modify the **CommandText** of the **SelectCommand** after the insert, update, or delete commands have been automatically generated, an exception may occur. If the modified **SelectCommand.CommandText** contains schema information that is inconsistent with the **SelectCommand.CommandText** used when the insert, update, or delete commands were automatically generated, future calls to the **DataAdapter.Update** method may attempt to access columns that no longer exist in the current table referenced by the **SelectCommand**, and an exception will be thrown.

You can refresh the schema information used by the **CommandBuilder** to automatically generate commands by calling the **RefreshSchema** method of the **CommandBuilder**.

If you want to find out what command was automatically generated, you can obtain a reference to the automatically generated commands using the **GetInsertCommand**, **GetUpdateCommand**, and **GetDeleteCommand** methods of the **CommandBuilder** object, and check the **CommandText** property of the associated **Command**. For example:

Console.WriteLine(custCB.GetUpdateCommand().CommandText)

The following example continues the code from the previous example and recreates the **Customers** table, replacing the **CompanyName** column with the **ContactName** column. The **RefreshSchema** method is called to refresh the automatically generated commands with this new column information.

[Visual Basic]

custDA.SelectCommand.CommandText = "SELECT CustomerID, ContactName FROM Customers"

custCB.RefreshSchema()

custDS.Tables.Remove(custDS.Tables("Customers"))

custDA.Fill(custDS, "Customers")

' Code to modify the new table in the DataSet here.

nwindConn.Open()

' Without the call to RefreshSchema, this line would fail.

custDA.Update(custDS, "Customers")

nwindConn.Close()

[C#]

custDA.SelectCommand.CommandText = "SELECT CustomerID, ContactName FROM Customers";

custCB.RefreshSchema();

custDS.Tables.Remove(custDS.Tables["Customers"]);

custDA.Fill(custDS, "Customers");

// Code to modify the new table in the DataSet here.

nwindConn.Open();

// Without the call to RefreshSchema, this line would fail.

custDA.Update(custDS, "Customers");

nwindConn.Close();

# Obtaining Schema Information from the Database

You can obtain schema information from your data source using both the SQL Server .NET Data Provider and the OLE DB .NET Data Provider. Schema information in a data source includes databases or catalogs available from the data source, tables and views in a database, constraints that exist, and so on. The SQL Server .NET Data Provider exposes schema information through stored procedures and informational views. For information about views and stored procedures available through Microsoft SQL Server, see the Transact-SQL reference located in the MSDN library at http://msdn.microsoft.com/library.

The OLE DB .NET Data Provider exposes schema information using the **GetOleDbSchemaTable** method of the **OleDbConnection** object. **GetOleDbSchemaTable** takes as arguments an **OleDbSchemaGuid** that identifies which schema information to return, and an array of restrictions on those returned columns. **GetOleDbSchemaTable** returns a **DataTable** populated with the schema information.

The following example returns the list of tables in the **Northwind** database. The SQL Server .NET Data Provider example selects the tables from an informational view provided by Microsoft SQL Server and populates a **DataTable** using a **DataAdapter**. The OLE DB .NET Data Provider example uses **GetOleDbSchemaTable** to return a **DataTable** with the schema information.

## SqlClient

[Visual Basic]

Dim nwindConn As SqlConnection = New SqlConnection("Data Source=localhost;Integrated Security=SSPI;Initial Catalog=northwind")

Dim schemaDA As SqlDataAdapter = New SqlDataAdapter("SELECT \* FROM INFORMATION\_SCHEMA.TABLES " & \_

"WHERE TABLE\_TYPE = 'BASE TABLE' " & \_

"ORDER BY TABLE\_TYPE", \_

nwindConn)

Dim schemaTable As DataTable = New DataTable

nwindConn.Open()

schemaDA.Fill(schemaTable)

nwindConn.Close()

[C#]

SqlConnection nwindConn = new SqlConnection("Data Source=localhost;Integrated Security=SSPI;Initial Catalog=northwind");

SqlDataAdapter schemaDA = new SqlDataAdapter("SELECT \* FROM INFORMATION\_SCHEMA.TABLES " +

"WHERE TABLE\_TYPE = 'BASE TABLE' " +

"ORDER BY TABLE\_TYPE",

nwindConn);

DataTable schemaTable = new DataTable();

nwindConn.Open();

schemaDA.Fill(schemaTable);

nwindConn.Close();

## OleDb

[Visual Basic]

Dim nwindConn As OleDbConnection = New OleDbConnection("Provider=SQLOLEDB;Data Source=localhost;Integrated Security=SSPI;Initial Catalog=northwind")

nwindConn.Open()

Dim schemaTable As DataTable = nwindConn.GetOleDbSchemaTable(OleDbSchemaGuid.Tables,

New Object() {Nothing, Nothing, Nothing, "TABLE"})

nwindConn.Close()

[C#]

OleDbConnection nwindConn = new OleDbConnection("Provider=SQLOLEDB;Data Source=localhost;Integrated Security=SSPI;Initial Catalog=northwind");

nwindConn.Open();

DataTable schemaTable = nwindConn.GetOleDbSchemaTable(OleDbSchemaGuid.Tables,

new object[] {null, null, null, "TABLE"});

nwindConn.Close();