Introduction to ADO.NET

Before ADO.NET we use ADO (Active Database Object) to access data from database. Basically ADO has automatic driver detection technique and it has only one drawback that it only provides a connected environment so efficiency of system may decrease.

ADO.NET is a new database technology used by .NET platform (introduced in 2002). In fact it is a set of classes used to communicate between an application front end and a database. It supports both **connected** & **disconnected** mode of data access.

Mandatory Namespaces used in ADO.NET

In any .NET data access page, before you connect to a database, you first have to import all the necessary namespaces that will allow you to work with the objects required. Namespaces used in ADO.Net are:

1. **System.Data**

   It contains the common classes for connecting, fetching data from database. Classes are like DataTable, DataSet, DataView etc.

2. **System.Data.SqlClient**

   It contains classes for connecting, fetching data from Sql Server database. Classes are like SqlDataAdapter, SqlDataReader etc.

3. **System.Data.OracleClient**

   It contains classes for connecting, fetching data from Oracle database. Classes are like OracleDataAdapter, OracleDataReader etc.

4. **System.Data.OleDb**

   It contains classes for connecting, fetching data from any database (like msaccess, db2, oracle, sqlserver, mysql). Classes are like OleDbDataAdapter, OleDbDataReader etc.

5. **System.Data.Odbc**

   It contains classes for connecting, fetching data from any database (like msaccess, db2, oracle, sqlserver, mysql). Classes are like OdbcDataAdapter,OdbcDataReader etc.

Component of ADO.NET architecture

1. **Data Provider**

   Data provider is a set of ADO.NET classes that allow us to access a database. Basically, it is a bridge between our application (We can say front-end) and data source. There are following Data Provider:

   SqlServer Data Provider:-It is used to access data from SqlServer database (for version 7.0 or later).

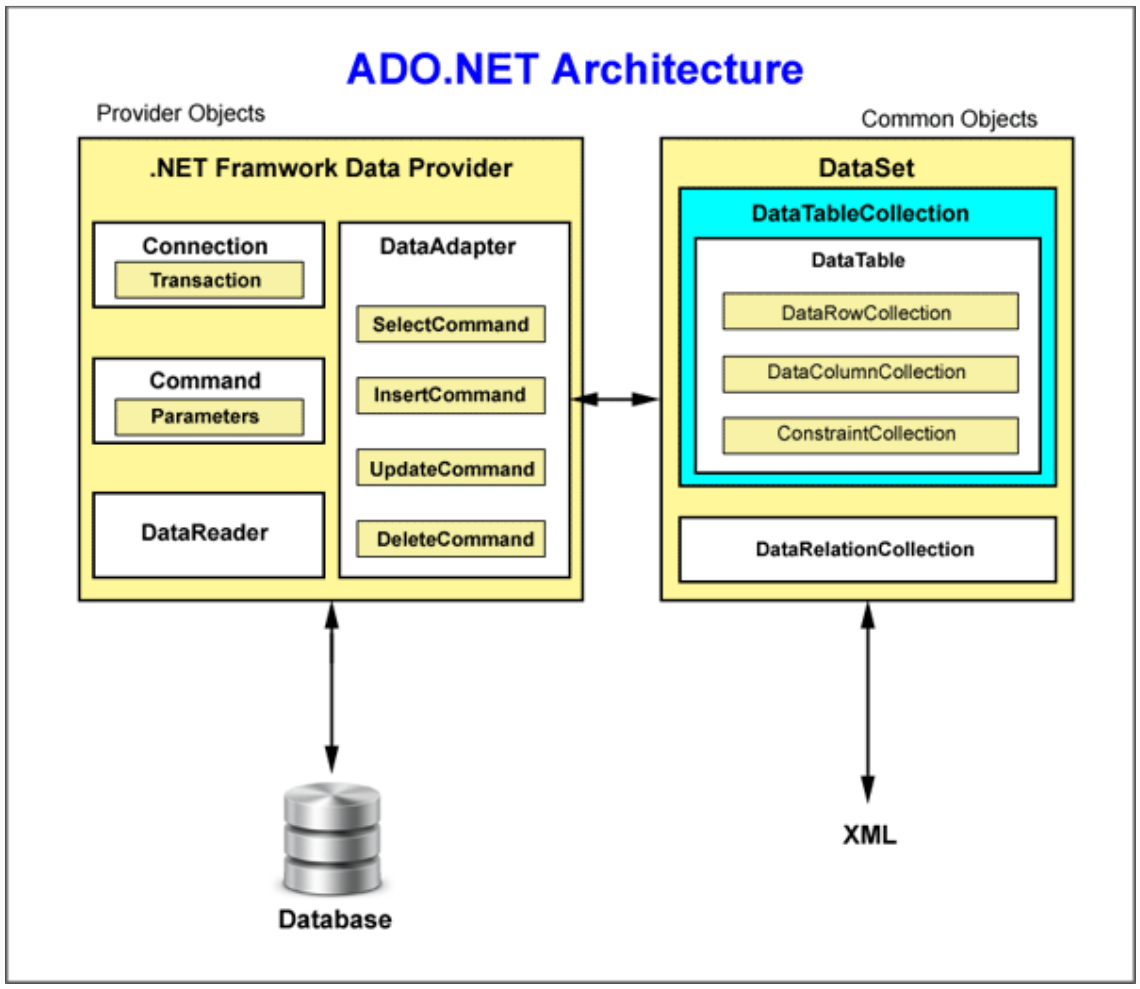
Oracle Data Provider:-It is used to access data from oracle database (for version 8i or later).

OleDb Data Provider:-It is used to access data from any database (msaccess, mysql, db2).

Odbc Data Provider :-It is used to access data from any database (msaccess, mysql, db2).

2. Data Set

Basically it is a small Data structure that may contain multiple data tables from multiple sources. The information in dataset is created inform of XML and is stored with .xsd extension. It support disconnected mode of data access. It has both scrolling mode means forward and backward scrolling mode (fetching of data). DataSet can have multiple DataTable from multiple sources but DataReader is able to read only single DataTable.



ADO.NET Objects

ADO.NET includes many objects you can use to work with data. This section introduces some of the primary objects you will use. Over the course of this tutorial, you'll be exposed to many more ADO.NET objects from the perspective of how they are used in a particular lesson. The objects below are the ones you must know. Learning about them will give you an idea of the types of things you can do with data when using ADO.NET.

The SqlConnection Object

To interact with a database, you must have a connection to it. The connection helps identify the database server, the database name, user name, password, and other parameters that are required for connecting to the data base. A connection object is used by command objects so they will know which database to execute the command on.

The SqlCommand Object

The process of interacting with a database means that you must specify the actions you want to occur. This is done with a command object. You use a command object to send SQL statements to the database. A command object uses a connection object to figure out which database to communicate with. You can use a command object alone, to execute a command directly, or assign a reference to a command object to a SqlDataAdapter, which holds a set of commands that work on a group of data.

The SqlDataReader Object

Many data operations require that you only get a stream of data for reading. The data reader object allows you to obtain the results of a SELECT statement from a command object. For performance reasons, the data returned from a data reader is a fast forward-only stream of data. This means that you can only pull the data from the stream in a sequential manner. This is good for speed, but if you need to manipulate data, then a DataSet is a better object to work with.

The DataSet Object

DataSet objects are in-memory representations of data. They contain multiple DataTable objects, which contain columns and rows, just like normal database tables. You can even define relations between tables to create parent-child relationships. The DataSet is specifically designed to help manage data in memory and to support disconnected operations on data, when such a scenario make sense. The DataSet is an object that is used by all of the Data Providers, which is why it does not have a Data Provider specific prefix.

The SqlDataAdapter Object

Sometimes the data you work with is primarily read-only and you rarely need to make changes to the underlying data source. Some situations also call for caching data in memory to minimize the number of database calls for data that does not change. The data adapter makes it easy for you to accomplish these things by helping to manage data in a disconnected mode. The data adapter fills a DataSet object when reading the data and writes in a single batch when persisting changes back to the database. A data adapter contains a reference to the connection object and opens and closes the connection automatically when reading from or writing to the database. Additionally, the data adapter contains command object references for SELECT, INSERT, UPDATE, and DELETE operations on the data. You will have a data adapter defined for each table in a

DataSet and it will take care of all communication with the database for you. All you need to do is tell the data adapter when to load from or write to the database.

Summary

ADO.NET is the .NET technology for interacting with data sources. You have several Data Providers, which allow communication with different data sources, depending on the protocols they use or what the database is. Regardless, of which Data Provider used, you'll use a similar set of objects to interact with a data source. The SqlConnection object lets you manage a connection to a data source. SqlCommand objects allow you to talk to a data source and send commands to it. To have fast forward-only read access to data, use the SqlDataReader. If you want to work with disconnected data, use a DataSet and implement reading and writing to/from the data source with a SqlDataAdapter.

The SqlConnection Object

Introduction

The first thing you will need to do when interacting with a data base is to create a connection. The connection tells the rest of the ADO.NET code which database it is talking to. It manages all of the low level logic associated with the specific database protocols. This makes it easy for you because the most work you will have to do in code is instantiate the connection object, open the connection, and then close the connection when you are done. Because of the way that other classes in ADO.NET are built, sometimes you don't even have to do that much work.

Although working with connections is very easy in ADO.NET, you need to understand connections in order to make the right decisions when coding your data access routines. Understand that a connection is a valuable resource. Sure, if you have a stand-alone client application that works on a single database one machine, you probably don't care about this. However, think about an enterprise application where hundreds of users throughout a company are accessing the same database. Each connection represents overhead and there can only be a finite amount of them. To look at a more extreme case, consider a Web site that is being hit with hundreds of thousands of hits a day Applications that grab connections and don't let them go can have seriously negative impacts on performance and scalability.

Creating a SqlConnection Object

A SqlConnection is an object, just like any other VB.NET object. Most of the time, you just declare and instantiate the SqlConnection all at the same time, as shown below:

```
Dim conn As SqlConnection = New SqlConnection ("Data Source=(local);Initial Catalog=Northwind;Integrated Security=SSPI")
```

The SqlConnection object instantiated above uses a constructor with a single argument of type string. This argument is called a connection string. Table 1 describes common parts of a connection string.

Table 1. ADO.NET Connection Strings contain certain key/value pairs for specifying how to make a database connection. They include the location, name of the database, and security credentials.

Connection String Parameter Name	**Description**
Data Source	Identifies the server. Could be local machine, machine domain name, or IP Address.
Initial Catalog	Database name.
Integrated Security	Set to SSPI to make connection with user's Windows login

User ID	Name of user configured in SQL Server.
Password	Password matching SQL Server User ID.

Integrated Security is secure when you are on a single machine doing development. However, you will often want to specify security based on a SQL Server User ID with permissions set specifically for the application you are using. The following shows a connection string, using the User ID and Password parameters:

```
Dim conn As SqlConnection = New SqlConnection("Data Source=DatabaseServer;Initial Catalog=Northwind;User ID=YourUserID;Password=YourPassword")
```

Notice how the Data Source is set to DatabaseServer to indicate that you can identify a database located on a different machine, over a LAN, or over the Internet. Additionally, User ID and Password replace the Integrated Security parameter.

Using a SqlConnection

The purpose of creating a SqlConnection object is so you can enable other ADO.NET code to work with a database. Other ADO.NET objects, such as a SqlCommand and a SqlDataAdapter take a connection object as a parameter. The sequences of operations occurring in the lifetime of a SqlConnection are as follows:

1. Instantiate the SqlConnection.
2. Open the connection.
3. Pass the connection to other ADO.NET objects.
4. Perform database operations with the other ADO.NET objects.
5. Close the connection.

We've already seen how to instantiate a SqlConnection. The rest of the steps, opening, passing, using, and closing are shown in Listing 1.

Listing 1. Using a SqlConnection

```
Imports System.Data
Imports System.Data.SqlClient
''' <summary>
''' Demonstrates how to work with SqlConnection objects
''' </summary>
Class SqlConnectionDemo
    Private Shared Sub Main()
        ' 1. Instantiate the connection
        'Dim conn As New SqlConnection("Data Source=(local);Initial Catalog=Northwind;Integrated Security=SSPI")
        Dim conn As SqlConnection = New SqlConnection("Data Source=.\SQLEXPRESS;AttachDbFilename=D:\BasicDB.mdf;
Integrated Security=True; Connect Timeout=30; User Instance=True")
        Dim rdr As SqlDataReader = Nothing
```

```
        Try
            ' 2. Open the connection
            conn.Open()
            ' 3. Pass the connection to a command object
            Dim cmd As New SqlCommand("select * from Customers", conn)
            ' 4. Use the connection
            'get query results
            rdr = cmd.ExecuteReader()
            ' print the CustomerID of each record
            While rdr.Read()
                Console.WriteLine(rdr(0))
            End While
        Finally
            ' close the reader
            If rdr IsNot Nothing Then
                rdr.Close()
            End If
            ' 5. Close the connection
            If conn IsNot Nothing Then
                conn.Close()
            End If
        End Try
    End Sub
End Class
```

As shown in Listing 1, you open a connection by calling the *Open()* method of the *SqlConnection* instance, *conn*. Any operations on a connection that was not yet opened will generate an exception. So, you must open the connection before using it.

Before using a *SqlCommand*, you must let the ADO.NET code know which connection it needs. In Listing 1, we set the second parameter to the *SqlCommand* object with the *SqlConnection* object, *conn*. Any operations performed with the *SqlCommand* will use that connection.

The code that uses the connection is a *SqlCommand* object, which performs a query on the Customers table. The result set is returned as a *SqlDataReader* and the *while* loop reads the first column from each row of the result set, which is the CustomerID column. We'll discuss the *SqlCommand* and *SqlDataReader* objects in later lessons. For right now, it is important for you to understand that these objects are using the *SqlConnection* object so they know what database to interact with.

When you are done using the connection object, you must close it. Failure to do so could have serious consequences in the performance and scalability of your application. There are a couple points to be made about how we closed the connection in Listing 1: the *Close()*method is called in a *finally* block and we ensure that the connection is not null before closing it.

Notice that we wrapped the ADO.NET code in a *try/finally* block, *finally* blocks help guarantee that a certain piece of code will be executed, regardless of whether or not an exception is generated. Since connections are scarce system resources, you will want to make sure they are closed in *finally* blocks.

Another precaution you should take when closing connections is to make sure the connection object is not *null*. If something goes wrong when instantiating the connection, it will be *null* and you want to make sure you don't try to close an invalid connection, which would generate an exception.

This example showed how to use a SqlConnection object with a SqlDataReader, which required explicitly closing the connection. However, when using a disconnected data model, you don't have to open and close the connection yourself. We'll see how this works in a future lesson when we look at the SqlDataAdapter object.

Summary

SqlConnection objects let other ADO.NET code know what database to connect to and how to make the connection. They are instantiated by passing a connection string with a set of key/value pairs that define the connection. The steps you use to manage the lifetime of a connection are create, open, pass, use, and close. Be sure to close your connection properly when you are done with it to ensure you don't have a connection resource leak.

The SqlCommand Object

Introduction

A SqlCommand object allows you to specify what type of interaction you want to perform with a database. For example, you can do select, insert, modify, and delete commands on rows of data in a database table. The SqlCommand object can be used to support disconnected data management scenarios, but in this lesson we will only use the SqlCommand object alone. A later lesson on the SqlDataAdapter will explain how to implement an application that uses disconnected data. This lesson will also show you how to retrieve a single value from a database, such as the number of records in a table.

Creating a SqlCommand Object

Similar to other VB.NET objects, you instantiate a SqlCommand object via the new instance declaration, as follows:

```
Dim cmd As New SqlCommand("select CategoryName from Categories", conn)
```

The line above is typical for instantiating a SqlCommand object. It takes a string parameter that holds the command you want to execute and a reference to a SqlConnection object. SqlCommand has a few overloads, which you will see in the examples of this tutorial.

Querying Data

When using a SQL select command, you retrieve a data set for viewing. To accomplish this with a SqlCommand object, you would use the ExecuteReader method, which returns a SqlDataReader object. We'll discuss the SqlDataReader in a future lesson. The example below shows how to use the SqlCommand object to obtain a SqlDataReader object:

```
    ' 1. Instantiate a new command with a query and connection
    Dim cmd As New SqlCommand("select CategoryName from Categories", conn)
    ' 2. Call Execute reader to get query results
    Dim rdr As SqlDataReader = cmd.ExecuteReader()
```

In the example above, we instantiate a SqlCommand object, passing the command string and connection object to the constructor. Then we obtain a SqlDataReader object by calling the ExecuteReader method of the SqlCommand object, cmd.

Inserting Data

To insert data into a database, use the ExecuteNonQuery method of the SqlCommand object. The following code shows how to insert data into a database table:

```
    ' prepare command string
    Dim insertString As String = " insert into Categories (CategoryName, Description) values ('Miscellaneous', 'Whatever doesn''t fit elsewhere')"
    ' 1. Instantiate a new command with a query and connection
    Dim cmd As New SqlCommand(insertString, conn)
    ' 2. Call ExecuteNonQuery to send command
    cmd.ExecuteNonQuery()
```

The SqlCommand instantiation is just a little different from what you've seen before, but it is basically the same. Instead of a literal string as the first parameter of the SqlCommand constructor, we are using a variable, insertString. The insertString variable is declared just above the SqlCommand declaration.

Notice the two apostrophes (") in the insertString text for the word "doesn"t". This is how you escape the apostrophe to get the string to populate column properly.

Another observation to make about the insert command is that we explicitly specified the columns CategoryName and Description. The Categories table has a primary key field named CategoryID. We left this out of the list because SQL Server will add this field itself. trying to add a value to a primary key field, such as CategoryID, will generate an exception.

To execute this command, we simply call the ExecuteNonQuery method on the SqlCommand instance, cmd.

Updating Data

The ExecuteNonQuery method is also used for updating data. The following code shows how to update data:

```
' prepare command string
Dim updateString As String = " update Categories set CategoryName = 'Other' where CategoryName = 'Miscellaneous'"
' 1. Instantiate a new command with command text only
Dim cmd As New SqlCommand(updateString)
' 2. Set the Connection property
cmd.Connection = conn
' 3. Call ExecuteNonQuery to send command
cmd.ExecuteNonQuery()
```

Again, we put the SQL command into a string variable, but this time we used a different SqlCommand constructor that takes only the command. In step 2, we assign the SqlConnection object, conn, to the Connection property of the SqlCommand object, cmd.

This could have been done with the same constructor used for the insert command, with two parameters. It demonstrates that you can change the connection object assigned to a command at any time.

The ExecuteNonQuery method performs the update command.

Deleting Data

You can also delete data using the ExecuteNonQuery method. The following example shows how to delete a record from a database with the ExecuteNonQuery method:

```
' prepare command string
Dim deleteString As String = " delete from Categories where CategoryName = 'Other'"
' 1. Instantiate a new command
Dim cmd As New SqlCommand()
' 2. Set the CommandText property
cmd.CommandText = deleteString
```

```
' 3. Set the Connection property
cmd.Connection = conn
' 4. Call ExecuteNonQuery to send command
cmd.ExecuteNonQuery()
```

This example uses the SqlCommand constructor with no parameters. Instead, it explicitly sets the CommandText and Connection properties of the SqlCommand object, cmd.

We could have also used either of the two previous SqlCommand constructor overloads, used for the insert or update command, with the same result. This demonstrates that you can change both the command text and the connection object at any time.

The ExecuteNonQuery method call sends the command to the database.

Getting Single values

Sometimes all you need from a database is a single value, which could be a count, sum, average, or other aggregated value from a data set. Performing an ExecuteReader and calculating the result in your code is not the most efficient way to do this. The best choice is to let the database perform the work and return just the single value you need. The following example shows how to do this with the ExecuteScalar method:

```
' 1. Instantiate a new command
Dim cmd As New SqlCommand("select count(*) from Categories", conn)
' 2. Call ExecuteNonQuery to send command
Dim count As Integer = CInt(cmd.ExecuteScalar())
```

The query in the SqlCommand constructor obtains the count of all records from the Categories table. This query will only return a single value. The ExecuteScalar method in step 2 returns this value. Since the return type of ExecuteScalar is type object, we use a cast operator to convert the value to int.

This code is part of the GetNumberOfRecords method of Listing 2 in the Putting it All Together section later in this lesson.

Putting it All Together

For simplicity, we showed snippets of code in previous sections to demonstrate the applicable techniques. It is also useful to have an entire code listing to see how this code is used in a working program. Listing 1 shows all of the code used in this example, along with a driver in the Main method to produce formatted output.

Listing 2. SqlConnection Demo

```
Imports System.Data
Imports System.Data.SqlClient
''' <summary>
''' Demonstrates how to work with SqlCommand objects
''' </summary>
Class SqlCommandDemo
    Private conn As SqlConnection
    Public Sub New()
```

```
        ' Instantiate the connection
        conn = New SqlConnection("Data Source=.\SQLEXPRESS;AttachDbFilename=D:\BasicDB.mdf;Integrated
Security=True; Connect Timeout=30; User Instance=True")
    End Sub
    ' call methods that demo SqlCommand capabilities
    Private Shared Sub Main()
        Dim scd As New SqlCommandDemo()
        Console.WriteLine()
        Console.WriteLine("Categories Before Insert")
        Console.WriteLine("------------------------")
        ' use ExecuteReader method
        scd.ReadData()
        ' use ExecuteNonQuery method for Insert
        scd.Insertdata()
        Console.WriteLine()
        Console.WriteLine("Categories After Insert")
        Console.WriteLine("-----------------------------")
        scd.ReadData()
        ' use ExecuteNonQuery method for Update
        scd.UpdateData()
        Console.WriteLine()
        Console.WriteLine("Categories After Update")
        Console.WriteLine("-----------------------------")
        scd.ReadData()
        ' use ExecuteNonQuery method for Delete
        scd.DeleteData()
        Console.WriteLine()
        Console.WriteLine("Categories After Delete")
        Console.WriteLine("-----------------------------")
        scd.ReadData()
        ' use ExecuteScalar method
        Dim numberOfRecords As Integer = scd.GetNumberOfRecords()
        Console.WriteLine()
        Console.WriteLine("Number of Records: {0}", numberOfRecords)
    End Sub
    ''' <summary>
    ''' use ExecuteReader method
    ''' </summary>
    Public Sub ReadData()
        Dim rdr As SqlDataReader = Nothing
```

```
        Try
            ' Open the connection
            conn.Open()
            ' 1. Instantiate a new command with a query and connection
            Dim cmd As New SqlCommand("select CategoryName from Categories", conn)
            ' 2. Call Execute reader to get query results
            rdr = cmd.ExecuteReader()
            ' print the CategoryName of each record
            While rdr.Read()
                Console.WriteLine(rdr(0))
            End While
        Finally
            ' close the reader
            If rdr IsNot Nothing Then
                rdr.Close()
            End If
            ' Close the connection
            If conn IsNot Nothing Then
                conn.Close()
            End If
        End Try
    End Sub
    ''' <summary>
    ''' use ExecuteNonQuery method for Insert
    ''' </summary>
    Public Sub Insertdata()
        Try
            ' Open the connection
            conn.Open()
            ' prepare command string
            Dim insertString As String = "insert into Categories (CategoryName, Description) values ('Miscellaneous', 'Whatever
doesn''t fit elsewhere')"
            ' 1. Instantiate a new command with a query and connection
            Dim cmd As New SqlCommand(insertString, conn)
            ' 2. Call ExecuteNonQuery to send command
            cmd.ExecuteNonQuery()
        Finally
            ' Close the connection
            If conn IsNot Nothing Then
                conn.Close()
```

```
            End If
        End Try
    End Sub
    ''' <summary>
    ''' use ExecuteNonQuery method for Update
    ''' </summary>
    Public Sub UpdateData()
        Try
            ' Open the connection
            conn.Open()
            ' prepare command string
            Dim updateString As String = "update Categories set CategoryName = 'Other' where CategoryName = 'Miscellaneous'"
            ' 1. Instantiate a new command with command text only
            Dim cmd As New SqlCommand(updateString)
            ' 2. Set the Connection property
            cmd.Connection = conn
            ' 3. Call ExecuteNonQuery to send command
            cmd.ExecuteNonQuery()
        Finally
            ' Close the connection
            If conn IsNot Nothing Then
                conn.Close()
            End If
        End Try
    End Sub
    ''' <summary>
    ''' use ExecuteNonQuery method for Delete
    ''' </summary>
    Public Sub DeleteData()
        Try
            ' Open the connection
            conn.Open()
            ' prepare command string
            Dim deleteString As String = "delete from Categories  where CategoryName = 'Other'"
            ' 1. Instantiate a new command
            Dim cmd As New SqlCommand()
            ' 2. Set the CommandText property
            cmd.CommandText = deleteString
            ' 3. Set the Connection property
```

```
            cmd.Connection = conn
            ' 4. Call ExecuteNonQuery to send command
            cmd.ExecuteNonQuery()
        Finally
            ' Close the connection
            If conn IsNot Nothing Then
                conn.Close()
            End If
        End Try
    End Sub
    ''' <summary>
    ''' use ExecuteScalar method
    ''' </summary>
    ''' <returns>number of records</returns>
    Public Function GetNumberOfRecords() As Integer
        Dim count As Integer = -1
        Try
            ' Open the connection
            conn.Open()
            ' 1. Instantiate a new command
            Dim cmd As New SqlCommand("select count(*) from Categories", conn)
            ' 2. Call ExecuteScalar to send command
            count = CInt(cmd.ExecuteScalar())
        Finally
            ' Close the connection
            If conn IsNot Nothing Then
                conn.Close()
            End If
        End Try
        Return count
    End Function
End Class
```

In Listing 2, the SqlConnection object is instantiated in the SqlCommandDemo structure. This is okay because the object itself will be cleaned up when the CLR garbage collector executes. What is important is that we close the connection when we are done using it. This program opens the connection in a try block and closes it in a finally block in each method.

The ReadData method displays the contents of the CategoryName column of the Categories table. We use it several times in the Main method to show the current status of the Categories table, which changes after each of the insert, update, and delete commands. Because of this, it is convenient to reuse to show you the effects after each method call.

Summary

A SqlCommand object allows you to query and send commands to a database. It has methods that are specialized for different commands. The ExecuteReader method returns a SqlDataReader object for viewing the results of a select query. For insert, update, and delete SQL commands, you use the ExecuteNonQuery method. If you only need a single aggregate value from a query, the ExecuteScalar is the best choice.

Reading Data with the SqlDataReader

Introduction

A SqlDataReader is a type that is good for reading data in the most efficient manner possible. You can *not* use it for writing data. SqlDataReaders are often described as fast-forward fire hose-like streams of data.

You can read from SqlDataReader objects in a forward-only sequential manner. Once you've read some data, you must save it because you will not be able to go back and read it again.

The forward only design of the SqlDataReader is what enables it to be fast. It doesn't have overhead associated with traversing the data or writing it back to the data source. Therefore, if your only requirement for a group of data is for reading one time and you want the fastest method possible, the SqlDataReader is the best choice. Also, if the amount of data you need to read is larger than what you would prefer to hold in memory beyond a single call, then the streaming behavior of the SqlDataReader would be a good choice.

Note: Observe that I used the term "one time" in the previous paragraph when discussing the reasons why you would use a SqlDataReader. As with anything, there are exceptions. In many cases, it is more efficient to use a cached DataSet. While caching is outside the scope of this tutorial, we will discuss using DataSet objects in the next lesson.

Creating a SqlDataReader Object

Getting an instance of a SqlDataReader is a little different than the way you instantiate other ADO.NET objects. You must call*ExecuteReader* on a command object, like this:

```
Dim rdr As SqlDataReader = cmd.ExecuteReader()
```

The *ExecuteReader* method of the SqlCommand object, *cmd* , returns a SqlDataReader instance. Creating a SqlDataReader with the new operator doesn't do anything for you. As you learned in previous lessons, the SqlCommand object references the connection and the SQL statement necessary for the SqlDataReader to obtain data.

Reading Data

As explained earlier, the SqlDataReader returns data via a sequential stream. To read this data, you must pull data from a table row-by-row. Once a row has been read, the previous row is no longer available. To read that row again, you would have to create a new instance of the SqlDataReader and read through the data stream again.

The typical method of reading from the data stream returned by the SqlDataReader is to iterate through each row with a while loop. The following code shows how to accomplish this:

```
While rdr.Read()
    ' get the results of each column
    Dim contact As String = DirectCast(rdr("ContactName"), String)
    Dim company As String = DirectCast(rdr("CompanyName"), String)
    Dim city As String = DirectCast(rdr("City"), String)
    ' print out the results
```

```
            Console.Write("{0,-25}", contact)
            Console.Write("{0,-20}", city)
            Console.Write("{0,-25}", company)
            Console.WriteLine()
        End While
```

Notice the call to *Read* on the SqlDataReader, *rdr*, in the *while* loop condition in the code above. The return value of *Read* is type bool and returns true as long as there are more records to read. After the last record in the data stream has been read, *Read* returns false.

In previous examples we extracted the first column from the row by using the SqlDataReader indexer, i.e. *rdr[0]*. You can extract each column of the row with a numeric indexer like this, but it isn't very readable. The example above uses a string indexer, where the string is the column name from the SQL query (the table column name if you used an asterisk, *. String indexers are much more readable, making the code easier to maintain.

Regardless of the type of the indexer parameter, a SqlDataReader indexer will return type object. This is why the example above casts results to a string. Once the values are extracted, you can do whatever you want with them, such as printing them to output with Console type methods.

Finishing Up

Always remember to close your SqlDataReader, just like you need to close the SqlConnection. Wrap the data access code in a try block and put the close operation in the finally block, like this:

```
        ' data access code
        Try
        Finally
            ' 3. close the reader
            If rdr IsNot Nothing Then
                rdr.Close()
             ' close the connection too
            End If
        End Try
```

The code above checks the SqlDataReader to make sure it isn't null. After the code knows that a good instance of the SqlDataReader exists, it can close it. Listing 3 shows the code for the previous sections in its entirety.

Listing 3: Using the SqlDataReader

```
Imports System.Data.SqlClient
Module M2
    Sub Main()
        ' declare the SqlDataReader, which is used in
        ' both the try block and the finally block
        Dim rdr As SqlDataReader = Nothing
        ' create a connection object
        Dim conn As New SqlConnection("Data Source=(local);Initial Catalog=Northwind;Integrated Security=SSPI")
```

```
        ' create a command object
        Dim cmd As New SqlCommand("select * from Customers", conn)
        Try
            ' open the connection
            conn.Open()
            ' 1. get an instance of the SqlDataReader
            rdr = cmd.ExecuteReader()
            ' print a set of column headers
            Console.WriteLine("Contact Name           City              Company Name")
            Console.WriteLine("-------------          ------------      ------------")
            ' 2. print necessary columns of each       record
            While rdr.Read()
                ' get the results of each column
                Dim contact As String = DirectCast(rdr("ContactName"), String)
                Dim company As String = DirectCast(rdr("CompanyName"), String)
                Dim city As String = DirectCast(rdr("City"), String)
                ' print out the results
                Console.Write("{0,-25}", contact)
                Console.Write("{0,-20}", city)
                Console.Write("{0,-25}", company)
                Console.WriteLine()
            End While
        Finally
            ' 3. close the reader
            If rdr IsNot Nothing Then
                rdr.Close()
            End If
            ' close the connection
            If conn IsNot Nothing Then
                conn.Close()
            End If
        End Try
    End Sub
End Module
```

Summary

SqlDataReader objects allow you to read data in a fast forward-only manner. You obtain data by reading each row from the data stream. Call the Close method of the SqlDataReader to ensure there are not any resource leaks.

Working with Disconnected Data – The DataSet and SqlDataAdapter

Introduction

In previous lessons we discussed a fully connected mode of operation for interacting with a data source by using the SqlCommand object. We also learned about how to read data quickly and let go of the connection with the SqlDataReader. This Lesson shows how to accomplish something in-between SqlConnection and SqlDataReader interaction by using the DataSet and SqlDataAdapter objects.

A DataSet is an in-memory data store that can hold numerous tables. DataSet only hold data and do not interact with a data source. It is the SqlDataAdapter that manages connections with the data source and gives us disconnected behavior. The SqlDataAdapter opens a connection only when required and closes it as soon as it has performed its task. For example, the SqlDataAdapter performs the following tasks when filling a DataSet with data:

1. Open connection
2. Retrieve data into DataSet
3. Close connection

and performs the following actions when updating data source with DataSet changes:

1. Open connection
2. Write changes from DataSet to data source
3. Close connection

In between the Fill and Update operations, data source connections are closed and you are free to read and write data with the DataSet as you need. These are the mechanics of working with disconnected data. Because the application holds on to connections only when necessary, the application becomes more scalable.

A couple scenarios illustrate why you would want to work with disconnected data: people working without network connectivity and making Web sites more scalable. Consider sales people who need customer data as they travel. At the beginning of the day, they'll need to sync up with the main database to have the latest information available. During the day, they'll make modifications to existing customer data, add new customers, and input new orders. This is okay because they have a given region or customer base where other people won't be changing the same records. At the end of the day, the sales person will connect to the network and update changes for overnight processing.

Another scenario is making a Web site more scalable. With a SqlDataReader, you have to go back to the database for records every time you show a page. This requires a new connection for each page load, which will hurt scalability as the number of users increase. One way to relieve this is to use a DataSet that is updated one time and stored in cache. Every request for the page checks the cache and loads the data if it isn't there or just pulls the data out of cache and displays it. This avoids a trip to the database, making your application more efficient.

Exceptions to the scenario above include situations where you need to update data. You then have to make a decision, based on the nature of how the data will be used as to your strategy. Use disconnected data when your information is primarily read only, but consider other alternatives (such as using SqlCommand object for immediate update) when your requirements call for something more dynamic. Also, if the amount of data is so large that holding it in memory is impractical, you will need to use SqlDataReader for read-only data. Really, one could come up with all kinds of exceptions, but the true guiding force should be the requirements of your application which will influence what your design should be.

Creating a DataSet Object

There isn't anything special about instantiating a DataSet. You just create a new instance, just like any other object:

```
Dim dsCustomers As New DataSet()
```

The DataSet constructor doesn't require parameters. However there is one overload that accepts a string for the name of the DataSet, which is used if you were to serialize the data to XML. Since that isn't a requirement for this example, I left it out. Right now, the DataSet is empty and you need a SqlDataAdapter to load it.

Creating a SqlDataAdapter

The SqlDataAdapter holds the SQL commands and connection object for reading and writing data. You initialize it with a SQL select statement and connection object:

```
Dim daCustomers As New SqlDataAdapter("select CustomerID, CompanyName from Customers", conn)
```

The code above creates a new SqlDataAdapter, *daCustomers*. The SQL select statement specifies what data will be read into a DataSet. The connection object, *conn*, should have already been instantiated, but not opened. It is the SqlDataAdapter's responsibility to open and close the connection during Fill and Update method calls.

As indicated earlier, the SqlDataAdapter contains all of the commands necessary to interact with the data source. The code showed how to specify the select statement, but didn't show the insert, update, and delete statements. These are added to the SqlDataAdapter after it is instantiated.

There are two ways to add insert, update, and delete commands: via SqlDataAdapter properties or with a SqlCommandBuilder. In this lesson, I'm going to show you the easy way of doing it with the SqlCommandBuilder. Here's how to add commands to the SqlDataAdapter with the SqlCommandBuilder:

```
Dim cmdBldr As New SqlCommandBuilder(daCustomers)
```

Notice in the code above that the SqlCommandBuilder is instantiated with a single constructor parameter of the SqlDataAdapter,*daCustomers*, instance. This tells the SqlCommandBuilder what SqlDataAdapter to add commands to. The SqlCommandBuilder will read the SQL select statement (specified when the SqlDataAdapter was instantiated), infer the insert, update, and delete commands, and assign the new commands to the Insert, Update, and Delete properties of the SqlDataAdapter, respectively.

As I mentioned earlier, the SqlCommandBuilder has limitations. It works when you do a simple select statement on a single table. However, when you need a join of two or more tables or must do a stored procedure, it won't work.

Filling the DataSet

Once you have a DataSet and SqlDataAdapter instances, you need to fill the DataSet. Here's how to do it, by using the Fill method of the SqlDataAdapter:

```
daCustomers.Fill (dsCustomers, "Customers")
```

The *Fill* method, in the code above, takes two parameters: a DataSet and a table name. The DataSet must be instantiated before trying to fill it with data. The second parameter is the name of the table that will be created in the DataSet. You can name the table anything you want. Its purpose is so you can identify the table with a meaningful name later on. Typically, I'll give it the same name as the database table. However, if the SqlDataAdapter's select command contains a join, you'll need to find another meaningful name.

The *Fill* method has an overload that accepts one parameter for the DataSet only. In that case, the table created has a default name of "table1" for the first table. The number will be incremented (table2, table3, …, tableN) for each table added to the DataSet where the table name was not specified in the Fill method.

Using the DataSet

A DataSet will bind with both ASP.NET and Windows forms DataGrids. Here's an example that assigns the DataSet to a Windows forms DataGrid:

```
dgCustomers.DataSource = dsCustomers
dgCustomers.DataMember = "Customers"
```

The first thing we do, in the code above, is assign the DataSet to the DataSource property of the DataGrid. This lets the DataGrid know that it has something to bind to, but you will get a '+' sign in the GUI because the DataSet can hold multiple tables and this would allow you to expand each available table. To specify exactly which table to use, set the DataGrid's *DataMember* property to the name of the table. In the example, we set the name to *Customers*, which is the same name used as the second parameter to the SqlDataAdapter Fill method. This is why I like to give the table a name in the *Fill* method, as it makes subsequent code more readable.

Updating Changes

After modifications are made to the data, you'll want to write the changes back to the database. Refer to previous discussion in the Introduction of this article on update guidance. The following code shows how to use the *Update* method of the SqlDataAdapter to push modifications back to the database.

```
daCustomers.Update(dsCustomers, "Customers")
```

The *Update* method, above, is called on the SqlDataAdapter instance that originally filled the *dsCustomers* DataSet. The second parameter to the *Update* method specifies which table, from the DataSet, to update. The table contains a list of records that have been modified and the Insert, Update, and Delete properties of the SqlDataAdapter contain the SQL statements used to make database modifications.

Putting it All Together

Until now, you've seen the pieces required to implement disconnected data management. What you really need is to see all this implemented in an application. Listing 4 shows how the code

from all the previous sections is used in a working program that has been simplified to enhance the points of this lesson:

Listing 4: Implementing a Disconnected Data Management Strategy

```
Imports System.Data
Imports System.Data.SqlClient
Imports System.Drawing
Imports System.Windows.Forms

Class DisconnectedDataform
   Inherits Form
   Private conn As SqlConnection
   Private daCustomers As SqlDataAdapter
   Private dsCustomers As DataSet
   Private dgCustomers As DataGrid
   Private Const tableName As String = "Customers"
   ' initialize form with DataGrid and Button
   Public Sub New()
      ' fill dataset
      Initdata()
      ' set up datagrid
      dgCustomers = New DataGrid()
      dgCustomers.Location = New Point(5, 5)
      dgCustomers.Size = New Size(Me.ClientRectangle.Size.Width - 10, Me.ClientRectangle.Height - 50)
      dgCustomers.DataSource = dsCustomers
      dgCustomers.DataMember = tableName
      ' create update button
      Dim btnUpdate As New Button()
      btnUpdate.Text = "Update"
      btnUpdate.Location = New Point(Me.ClientRectangle.Width / 2 - btnUpdate.Width / 2, Me.ClientRectangle.Height -
(btnUpdate.Height + 10))
      btnUpdate.Click += New EventHandler(AddressOf btnUpdateClicked)
      ' make sure controls appear on form
      Controls.AddRange(New Control() {dgCustomers, btnUpdate})
   End Sub
   ' set up ADO.NET objects
   Public Sub Initdata()
      ' instantiate the connection
      conn = New SqlConnection("Server=(local);DataBase=Northwind;Integrated Security=SSPI")
      ' 1. instantiate a new DataSet
      dsCustomers = New DataSet()
```

```
        ' 2. init SqlDataAdapter with select command and connection
        daCustomers = New SqlDataAdapter("select CustomerID, CompanyName from Customers", conn)
        ' 3. fill in insert, update, and delete commands
        Dim cmdBldr As New SqlCommandBuilder(daCustomers)
        ' 4. fill the dataset
        daCustomers.Fill(dsCustomers, tableName)
    End Sub
    ' Update button was clicked
    Public Sub btnUpdateClicked(ByVal sender As Object, ByVal e As EventArgs)
        ' write changes back to DataBase
        daCustomers.Update(dsCustomers, tableName)
    End Sub
    ' start the Windows form
    Private Shared Sub Main()
        Application.Run(New DisconnectedDataform())
    End Sub
End Class
```

The *Initdata* method in Listing 4 contains the methods necessary to set up the SqlDataAdapter and DataSet. Notice that various data objects are defined at class level so they can be used in multiple methods. The DataGrid's *DataSource* property is set in the constructor. Whenever a user clicks the Update button, the *Update* method in the *btnUpdateClicked* event handler is called, pushing modifications back to the database.

Summary

DataSets hold multiple tables and can be kept in memory and reused. The SqlDataAdapter enables you to fill a DataSet and Update changes back to the database. You don't have to worry about opening and closing the SqlConnection because the SqlDataAdapter does it automatically. A SqlCommandBuilder populates insert, update, and delete commands based on the SqlDataAdapter's select statement. Use the *Fill* method of the SqlDataAdapter to fill a DataSet with data. Call the SqlDataAdapter's *Update* method to push changes back to a database.

Adding Parameters to Commands

Introduction

When working with data, you'll often want to filter results based on some criteria. Typically, this is done by accepting input from a user and using that input to form a SQL query. For example, a sales person may need to see all orders between specific dates. Another query might be to filter customers by city.

As you know, the SQL query assigned to a SqlCommand object is simply a string. So, if you want to filter a query, you could build the string dynamically, but you wouldn't want to. Here is a bad example of filtering a query.

```
' don't ever do this

Dim cmd As New SqlCommand("select * from Customers where city = '" & inputCity & "'", conn)
```

Don't ever build a query this way! The input variable, *inputCity*, is typically retrieved from a TextBox control on either a Windows form or a Web Page. Anything placed into that TextBox control will be put into *inputCity* and added to your SQL string. This situation invites a hacker to replace that string with something malicious. In the worst case, you could give full control of your computer away.

Instead of dynamically building a string, as shown in the bad example above, use parameters. Anything placed into a parameter will be treated as field data, not part of the SQL statement, which makes your application much more secure.

Using parameterized queries is a three step process:

1. Construct the SqlCommand command string with parameters.
2. Declare a SqlParameter object, assigning values as appropriate.
3. Assign the SqlParameter object to the SqlCommand object's Parameters property.

The following sections take you step-by-step through this process.

Preparing a SqlCommand Object for Parameters

The first step in using parameters in SQL queries is to build a command string containing parameter placeholders. These placeholders are filled in with actual parameter values when the SqlCommand executes. Proper syntax of a parameter is to use a '@' symbol prefix on the parameter name as shown below:

```
Dim cmd As New SqlCommand("select * from Customers where city = @City", conn)
```

In the SqlCommand constructor above, the first argument contains a parameter declaration, *@City*. This example used one parameter, but you can have as many parameters as needed to customize the query. Each parameter will match a SqlParameter object that must be assigned to this SqlCommand object.

Declaring a SqlParameter Object

Each parameter in a SQL statement must be defined. This is the purpose of the SqlParameter type. Your code must define a SqlParameter instance for each parameter in a SqlCommand

object's SQL command. The following code defines a parameter for the *@City* parameter from the previous section:

```
' 2. define parameters used in command object
Dim param As New SqlParameter()
param.ParameterName = "@City"
param.Value = inputCity
```

Notice that the ParameterName property of the SqlParameter instance must be spelled exactly as the parameter that is used in the SqlCommand SQL command string. You must also specify a value for the command. When the SqlCommand object executes, the parameter will be replaced with this value.

Associate a SqlParameter Object with a SqlCommand Object

For each parameter defined in the SQL command string argument to a SqlCommand object, you must define a SqlParameter. You must also let the SqlCommand object know about the SqlParameter by assigning the SqlParameter instance to the Parameters property of the SqlCommand object. The following code shows how to do this:

```
' 3. add new parameter to command object
        cmd.Parameters.Add(param);
```

The SqlParameter instance is the argument to the Add method of the Parameters property for the SqlCommand object above. You must add a unique SqlParameter for each parameter defined in the SqlCommand object's SQL command string.

Putting it All Together

You already know how to use SqlCommand and SqlDataReader objects. The following code demonstrates a working program that uses SqlParameter objects. So, everything should be familiar by now, except for the new parts presented in this article:

Listing 5: Adding Parameters to Queries

```
Imports System.Data
Imports System.Data.SqlClient
Class ParamDemo
  Private Shared Sub Main()
    ' conn and reader declared outside try
    ' block for visibility in finally block
    Dim conn As SqlConnection = Nothing
    Dim reader As SqlDataReader = Nothing
    Dim inputCity As String = "London"
    Try
      ' instantiate and open connection
      conn = New SqlConnection("Server=(local);DataBase=Northwind;Integrated Security=SSPI")
      conn.Open()
      ' don't ever do this
```

```
        ' Dim cmd As New SqlCommand("select * from Customers where city = '" & inputCity & "'",
conn)
        ' 1. declare command object with parameter
        Dim cmd As New SqlCommand("select * from Customers where city = @City", conn)
        ' 2. define parameters used in command object
        Dim param As New SqlParameter()
        param.ParameterName = "@City"
        param.Value = inputCity
        ' 3. add new parameter to command object
        cmd.Parameters.Add(param)
        ' get data stream
        reader = cmd.ExecuteReader()
        ' write each record
        While reader.Read()
            Console.WriteLine("{0}, {1}", reader("CompanyName"), reader("ContactName"))
        End While
      Finally
        ' close reader
        If reader IsNot Nothing Then
            reader.Close()
        End If
        ' close connection
        If conn IsNot Nothing Then
            conn.Close()
        End If
      End Try
    End Sub
End Class
```

The code in Listing 5 retrieves records for each customer that lives in London. This was made more secure through the use of parameters. Besides using parameters, all of the other code contains techniques you've learned in previous lessons.

Summary

You should use parameters to filter queries in a secure manner. The process of using parameter contains three steps: define the parameter in the SqlCommand command string, declare the SqlParameter object with applicable properties, and assign the SqlParameter object to the SqlCommand object. When the SqlCommand executes, parameters will be replaced with values specified by the SqlParameter object.

Using Stored Procedures

Introduction

A stored procedure is a pre-defined, reusable routine that is stored in a database. SQL Server compiles stored procedures, which makes them more efficient to use. Therefore, rather than dynamically building queries in your code, you can take advantage of the reuse and performance benefits of stored procedures. The following sections will show you how to modify the SqlCommand object to use stored procedures. Additionally, you'll see another reason why parameter support is an important part of the ADO.NET libraries.

Executing a Stored Procedure

In addition to commands built with strings, the SqlCommand type can be used to execute stored procedures. There are two tasks require to make this happen: let the SqlCommand object know which stored procedure to execute and tell the SqlCommand object that it is executing a stored procedure. These two steps are shown below:

```
' 1. create a command object identifying the stored procedure
Dim cmd As New SqlCommand("Ten Most Expensive Products", conn)
' 2. set the command object so it knows to execute a stored procedure
cmd.CommandType = CommandType.StoredProcedure
```

While declaring the SqlCommand object above, the first parameter is set to "Ten Most Expensive Products". This is the name of a stored procedure in the Northwind database. The second parameter is the connection object, which is the same as the SqlCommand constructor used for executing query strings.

The second command tells the SqlCommand object what type of command it will execute by setting its *CommandType* property to the *StoredProcedure* value of the CommandType enum. The default interpretation of the first parameter to the SqlCommand constructor is to treat it as a query string. By setting the *CommandType* to *StoredProcedure*, the first parameter to the SqlCommand constructor will be interpreted as the name of a stored procedure (instead of interpreting it as a command string). The rest of the code can use the SqlCommand object the same as it is used in previous lessons.

Sending Parameters to Stored Procedures

Using parameters for stored procedures is the same as using parameters for query string commands. The following code shows this:

```
' 1. create a command object identifying the stored procedure
Dim cmd As New SqlCommand("CustOrderHist", conn)
' 2. set the command object so it knows to execute a stored procedure
cmd.CommandType = CommandType.StoredProcedure
' 3. add parameter to command, which will be passed to the stored procedure
cmd.Parameters.Add(New SqlParameter("@CustomerID", custId))
```

The SqlCommand constructor above specifies the name of a stored procedure, *CustOrderHist*, as its first parameter. This particular stored procedure takes a single parameter,

named *@CustomerID*. Therefore, we must populate this parameter using a SqlParameter object. The name of the parameter passed as the first parameter to the SqlParameter constructor must be spelled exactly the same as the stored procedure parameter. Then execute the command the same as you would with any other SqlCommand object.

A Full Example

The code in Listing 6 contains a full working example of how to use stored procedures. There are separate methods for a stored procedure without parameters and a stored procedure with parameters.

Listing 6: Executing Stored Procedures

```
Imports System.Data
Imports System.Data.SqlClient
Class StoredProcDemo
   Private Shared Sub Main()
      Dim spd As New StoredProcDemo()
      ' run a simple stored procedure
      spd.RunStoredProc()
      ' run a stored procedure that takes a parameter
      spd.RunStoredProcParams()
   End Sub
   ' run a simple stored procedure
   Public Sub RunStoredProc()
      Dim conn As SqlConnection = Nothing
      Dim rdr As SqlDataReader = Nothing
      Console.WriteLine(vbLf & "Top 10 Most Expensive Products:" & vbLf)
      Try
         ' create and open a connection object
         conn = New SqlConnection("Server=(local);DataBase=Northwind;Integrated Security=SSPI")
         conn.Open()
         ' 1. create a command object identifying the stored procedure
         Dim cmd As New SqlCommand("Ten Most Expensive Products", conn)
         ' 2. set the command object so it knows to execute a stored procedure
         cmd.CommandType = CommandType.StoredProcedure
         ' execute the command
         rdr = cmd.ExecuteReader()
         ' iterate through results, printing each to console
         While rdr.Read()
            Console.WriteLine("Product: {0,-25} Price: ${1,6:####.00}", rdr("TenMostExpensiveProducts"), rdr("UnitPrice"))
         End While
      Finally
```

```
        If conn IsNot Nothing Then
            conn.Close()
        End If
        If rdr IsNot Nothing Then
            rdr.Close()
        End If
    End Try
End Sub
' run a stored procedure that takes a parameter
Public Sub RunStoredProcParams()
    Dim conn As SqlConnection = Nothing
    Dim rdr As SqlDataReader = Nothing
    ' typically obtained from user input, but we take a short cut
    Dim custId As String = "FURIB"
    Console.WriteLine(vbLf & "Customer Order History:" & vbLf)
    Try
        ' create and open a connection object
        conn = New SqlConnection("Server=(local);DataBase=Northwind;Integrated Security=SSPI")
        conn.Open()
        ' 1. create a command object identifying the stored procedure
        Dim cmd As New SqlCommand("CustOrderHist", conn)
        ' 2. set the command object so it knows to execute a stored procedure
        cmd.CommandType = CommandType.StoredProcedure
        ' 3. add parameter to command, which will be passed to the stored procedure
        cmd.Parameters.Add(New SqlParameter("@CustomerID", custId))
        ' execute the command
        rdr = cmd.ExecuteReader()
        ' iterate through results, printing each to console
        While rdr.Read()
            Console.WriteLine("Product: {0,-35} Total: {1,2}", rdr("ProductName"), rdr("Total"))
        End While
    Finally
        If conn IsNot Nothing Then
            conn.Close()
        End If
        If rdr IsNot Nothing Then
            rdr.Close()
        End If
    End Try
End Sub
```

```
End Class
```

The *RunStoredProc* method in Listing 6 simply runs a stored procedure and prints the results to the console. In the *RunStoredProcParams* method, the stored procedure used takes a single parameter. This demonstrates that there is no difference between using parameters with query strings and stored procedures. The rest of the code should be familiar to those who have read previous lessons in this tutorial.

Summary

To execute stored procedures, you specify the name of the stored procedure in the first parameter of a SqlCommand constructor and then set the *CommandType* of the SqlCommand to *StoredProcedure*. You can also send parameters to a stored procedure by using SqlParameter objects, the same way it is done with SqlCommand objects that execute query strings. Once the SqlCommand object is constructed, you can use it just like any other SqlCommand object as described in previous lessons.