Lesson 01: Introduction to ADO.NET

Introduction

ADO.NET is an object-oriented set of libraries that allows you to interact with data sources. Commonly, the data source is a database, but it could also be a text file, an Excel spreadsheet, or an XML file. For the purposes of this tutorial, we will look at ADO.NET as a way to interact with a data base.

As you are probably aware, there are many different types of databases available. For example, there is Microsoft SQL Server, Microsoft Access, Oracle, Borland Interbase, and IBM DB2, just to name a few. To further refine the scope of this tutorial, all of the examples will use SQL Server.

You can download the Microsoft SQL Server 2000 Desktop Engine (MSDE 2000) here:

http://www.microsoft.com/sql/msde/downloads/download.asp

MSDE contains documentation on how to perform an installation. However, for your convenience, here are quick instructions on how to install MSDE:

http://www.asp.net/msde/default.aspx?tabindex=0&tabid=1

MSDE 2000 is a scaled down version of SQL Server. Therefore, everything you learn in this tutorial and all code will work with SQL Server. The examples will use the Northwind database. This is a tutorial is specifically for ADO.NET. MSDE is not part of ADO.NET, but it is one of the many data sources you can interact with by using ADO.NET If you need help with MSDE 2000, I refer you to the Microsoft Web site, where you can find pertinent information on licensing and technical assistance:

http://www.microsoft.com/sql/msde/

Data Providers

We know that ADO.NET allows us to interact with different types of data sources and different types of databases. However, there isn't a single set of classes that allow you to accomplish this universally. Since different data sources expose different protocols, we need a way to communicate with the right data source using the right protocol. Some older data sources use the ODBC protocol, many newer data sources use the OleDb protocol, and there are more data sources every day that allow you to communicate with them directly through .NET ADO.NET class libraries.

ADO.NET provides a relatively common way to interact with data sources, but comes in different sets of libraries for each way you can talk to a data source. These libraries are called Data Providers and are usually named for the protocol or data source type they allow you to

interact with. table 1 lists some well known data providers, the API prefix they use, and the type of data source they allow you to interact with.

table 1. ADO.NET Data Providers are class libraries that allow a common way to interact with specific data sources or protocols. The library APIs have prefixes that indicate which provider they support.

Provider Name	API prefix	Data Source Description
ODBC Data Provider	Odbc	Data Sources with an ODBC interface. Normally older data bases.
OleDb Data Provider	OleDb	Data Sources that expose an OleDb interface, i.e. Access or Excel.
Oracle Data Provider	Oracle	For Oracle Databases.
SQL Data Provider	Sql	For interacting with Microsoft SQL Server.
Borland Data Provider	Bdp	Generic access to many databases such as Interbase, SQL Server, IBM DB2, and Oracle.

An example may help you to understand the meaning of the API prefix. One of the first ADO.NET objects you'll learn about is the connection object, which allows you to establish a connection to a data source. If we were using the OleDb Data Provider to connect to a data source that exposes an OleDb interface, we would use a connection object named OleDbConnection. Similarly, the connection object name would be prefixed with Odbc or Sql for an OdbcConnection object on an Odbc data source or a SqlConnection object on a SQL Server database, respectively. Since we are using MSDE in this tutorial (a scaled down version of SQL Server) all the API objects will have the Sql prefix. i.e. SqlConnection.

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 an SqlDataAdapter, which holds a set of commands that work on a group of data as described below.

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.

Lesson 02: The SqlConnection Object

This lesson describes the SqlConnection object and how to connect to a data base. Here are the objectives of this lesson:

- Know what connection objects are used for.
- Learn how to instantiate a SqlConnection object.
- Understand how the SqlConnection object is used in applications.
- Comprehend the importance of effective connection lifetime management.

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 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 C# object. Most of the time, you just declare and instantiate the SqlConnection all at the same time, as shown below:

```
SqlConnection conn = 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	Description

Name	
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:

```
SqlConnection conn = 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 sequence 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

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

```
/// <summary>
/// Demonstrates how to work with SqlConnection objects
/// </summary>
class SqlConnectionDemo
{
   static void Main()
   {
      // 1. Instantiate the connection
      SqlConnection conn = new SqlConnection(
         "Data Source=(local);Initial Catalog=Northwind;Integrated Security=SSPI");

      SqlDataReader rdr = null;

      try
      {
         // 2. Open the connection
         conn.Open();

         // 3. Pass the connection to a command object
         SqlCommand cmd = 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]);
         }
      }
      finally
      {
         // close the reader
         if (rdr != null)
         {
            rdr.Close();
         }

         // 5. Close the connection
         if (conn != null)
         {
            conn.Close();
         }
      }
   }
}
```

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. As described in Lesson 15: Introduction to Exception Handling of the C# Tutorial, *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.

Lesson 03: The SqlCommand Object

This lesson describes the SqlCommand object and how you use it to interact with a database. Here are the objectives of this lesson:

- Know what a command object is.
- Learn how to use the ExecuteReader method to query data.
- Learn how to use the ExecuteNonQuery method to insert and delete data.
- Learn how to use the ExecuteScalar method to return a single value.

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 C# objects, you instantiate a SqlCommand object via the new instance declaration, as follows:

```
SqlCommand cmd = 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
SqlCommand cmd = new SqlCommand("select CategoryName from Categories", conn);

// 2. Call Execute reader to get query results
SqlDataReader rdr = 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.

This code is part of the ReadData method of Listing 1 in the Putting it All Together section later in this lesson.

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
 string insertString = @"
    insert into Categories
    (CategoryName, Description)
    values ('Miscellaneous', 'Whatever doesn"t fit elsewhere')";

 // 1. Instantiate a new command with a query and connection
 SqlCommand cmd = 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.

This code is part of the Insertdata method of Listing 1 in the Putting it All Together section later in this lesson.

Updating Data

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

```
// prepare command string
 string updateString = @"
    update Categories
    set CategoryName = 'Other'
```

```
    where CategoryName = 'Miscellaneous'";

 // 1. Instantiate a new command with command text only
 SqlCommand cmd = 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.

This code is part of the UpdateData method of Listing 1 in the Putting it All Together section later in this lesson.

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
 string deleteString = @"
    delete from Categories
    where CategoryName = 'Other'";

 // 1. Instantiate a new command
 SqlCommand cmd = 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 explicity 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.

This code is part of the DeleteData method of Listing 1 in the Putting it All Together section later in this lesson.

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
 SqlCommand cmd = new SqlCommand("select count(*) from Categories", conn);

 // 2. Call ExecuteNonQuery to send command
 int count = (int)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 1 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 1. SqlConnection Demo

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

/// <summary>
/// Demonstrates how to work with SqlCommand objects
```

```
/// </summary>
class SqlCommandDemo
{
   SqlConnection conn;

   public SqlCommandDemo()
   {
      // Instantiate the connection
      conn = new SqlConnection(
        "Data Source=(local);Initial Catalog=Northwind;Integrated Security=SSPI");
   }

   // call methods that demo SqlCommand capabilities
   static void Main()
   {
      SqlCommandDemo scd = 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
      int numberOfRecords = scd.GetNumberOfRecords();

      Console.WriteLine();
      Console.WriteLine("Number of Records: {0}", numberOfRecords);
```

```
}

/// <summary>
/// use ExecuteReader method
/// </summary>
public void ReadData()
{
  SqlDataReader rdr = null;

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

      // 1. Instantiate a new command with a query and connection
      SqlCommand cmd = 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]);
      }
   }
   finally
   {
      // close the reader
      if (rdr != null)
      {
         rdr.Close();
      }

      // Close the connection
      if (conn != null)
      {
         conn.Close();
      }
   }
}

/// <summary>
/// use ExecuteNonQuery method for Insert
/// </summary>
public void Insertdata()
{
   try
   {
      // Open the connection
      conn.Open();

      // prepare command string
      string insertString = @"
         insert into Categories
```

```
            (CategoryName, Description)
            values ('Miscellaneous', 'Whatever doesn''t fit elsewhere')";

        // 1. Instantiate a new command with a query and connection
        SqlCommand cmd = new SqlCommand(insertString, conn);

        // 2. Call ExecuteNonQuery to send command
        cmd.ExecuteNonQuery();
    }
    finally
    {
        // Close the connection
        if (conn != null)
        {
            conn.Close();
        }
    }
}

/// <summary>
/// use ExecuteNonQuery method for Update
/// </summary>
public void UpdateData()
{
    try
    {
        // Open the connection
        conn.Open();

        // prepare command string
        string updateString = @"
            update Categories
            set CategoryName = 'Other'
            where CategoryName = 'Miscellaneous'";

        // 1. Instantiate a new command with command text only
        SqlCommand cmd = 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 != null)
        {
            conn.Close();
        }
    }
}

/// <summary>
```

```
/// use ExecuteNonQuery method for Delete
/// </summary>
public void DeleteData()
{
   try
   {
      // Open the connection
      conn.Open();

      // prepare command string
      string deleteString = @"
         delete from Categories
         where CategoryName = 'Other'";

      // 1. Instantiate a new command
      SqlCommand cmd = 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 != null)
      {
         conn.Close();
      }
   }
}

/// <summary>
/// use ExecuteScalar method
/// </summary>
/// <returns>number of records</returns>
public int GetNumberOfRecords()
{
   int count = -1;

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

      // 1. Instantiate a new command
      SqlCommand cmd = new SqlCommand("select count(*) from Categories", conn);

      // 2. Call ExecuteScalar to send command
      count = (int)cmd.ExecuteScalar();
   }
```

```
        finally
        {
          // Close the connection
          if (conn != null)
          {
             conn.Close();
          }
        }
        return count;
    }
}
```

In Listing 1, 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.

Lesson 04: Reading Data with the SqlDataReader

This lesson explains how to read data with a SqlDataReader object. Here are the objectives of this lesson:

- Learn what a SqlDataReader is used for.
- Know how to read data using a SqlDataReader.
- Understand the need to close a 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 firehose-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:

```
SqlDataReader rdr = 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

previous lessons contained code that used a SqlDataReader, but the discussion was delayed so we could focus on the specific subject of that particular lesson. This lesson builds from what you've seen and explains how to use the SqlDataReader.

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
        string contact = (string)rdr["ContactName"];
        string company = (string)rdr["CompanyName"];
        string city    = (string)rdr["City"];

        // print out the results
        Console.Write("{0,-25}", contact);
        Console.Write("{0,-20}", city);
        Console.Write("{0,-25}", company);
        Console.WriteLine();
}
```

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 lessons, 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:

```
try
{
        // data access code
}
finally
{
        // 3. close the reader
        if (rdr != null)
```

```
                {
                        rdr.Close();
                }

                // close the connection too
        }
```

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 1 shows the code for the previous sections in its entirety.

Listing 1: Using the SqlDataReader

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

namespace Lesson04
{
        class ReaderDemo
        {
                static void Main()
                {
                        ReaderDemo rd = new ReaderDemo();
                        rd.SimpleRead();
                }

                public void SimpleRead()
                {
                        // declare the SqlDataReader, which is used in
                        // both the try block and the finally block
                        SqlDataReader rdr = null;

                        // create a connection object
                        SqlConnection conn = new SqlConnection(
"Data Source=(local);Initial Catalog=Northwind;Integrated Security=SSPI");

                        // create a command object
                        SqlCommand cmd  = 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
                                        string contact =
(string)rdr["ContactName"];
                                        string company =
(string)rdr["CompanyName"];
                                        string city    = (string)rdr["City"];

                                        // print out the results
                                        Console.Write("{0,-25}", contact);
                                        Console.Write("{0,-20}", city);
                                        Console.Write("{0,-25}", company);
                                        Console.WriteLine();
                                }
                        }
                        finally
                        {
                                // 3. close the reader
                                if (rdr != null)
                                {
                                        rdr.Close();
                                }

                                // close the connection
                                if (conn != null)
                                {
                                        conn.Close();
                                }
                        }
                }
        }
}
```

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.

Lesson 05: Working with Disconnected Data - The DataSet and SqlDataAdapter

Introduction

In Lesson 3, we discussed a fully connected mode of operation for interacting with a data source by using the SqlCommand object. In Lesson 4, we learned about how to read data quickly an 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. DataSets 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 applications 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:

```
DataSet dsCustomers = 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:

```
SqlDataAdapter daCustomers = 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 statment, 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. In a later lesson, I'll show you how to use the SqlDataAdapter properties, which takes more work but will give you more capabilities than what the

SqlCommandBuilder does. Here's how to add commands to the SqlDataAdapter with the SqlCommandBuilder:

```
SqlCommandBuilder cmdBldr = 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 mor tables or must do a stored procedure, it won't work. I'll describe a work-around for these scenarios in future lessons.

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 managment. What you really need is to see all this implemented in an application. Listing 1 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 1: Implementing a Disconnected Data Management Strategy

```
using System;
using System.Data;
using System.Data.SqlClient;
using System.Drawing;
using System.Windows.Forms;

class DisconnectedDataform : Form
{
        private SqlConnection  conn;
        private SqlDataAdapter daCustomers;

        private DataSet  dsCustomers;
        private DataGrid dgCustomers;

        private const string tableName = "Customers";

        // initialize form with DataGrid and Button
        public DisconnectedDataform()
        {
                // fill dataset
                Initdata();
```

```
                // set up datagrid
                dgCustomers = new DataGrid();
                dgCustomers.Location = new Point(5, 5);
                dgCustomers.Size = new Size(
                        this.ClientRectangle.Size.Width - 10,
                        this.ClientRectangle.Height - 50);
                dgCustomers.DataSource = dsCustomers;
                dgCustomers.DataMember = tableName;

                // create update button
                Button btnUpdate = new Button();
                btnUpdate.Text = "Update";
                btnUpdate.Location = new Point(
                        this.ClientRectangle.Width/2 - btnUpdate.Width/2,
                        this.ClientRectangle.Height - (btnUpdate.Height + 10));
                btnUpdate.Click += new EventHandler(btnUpdateClicked);

                // make sure controls appear on form
                Controls.AddRange(new Control[] { dgCustomers, btnUpdate });
        }

        // set up ADO.NET objects
        public void 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
                SqlCommandBuilder cmdBldr = new
SqlCommandBuilder(daCustomers);

                // 4. fill the dataset
                daCustomers.Fill(dsCustomers, tableName);
        }

        // Update button was clicked
        public void btnUpdateClicked(object sender, EventArgs e)
        {
                // write changes back to DataBase
                daCustomers.Update(dsCustomers, tableName);
        }

        // start the Windows form
        static void Main()
        {
                Application.Run(new DisconnectedDataForm());
        }
 }
```

The *Initdata* method in Listing 1 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.

Lesson 06: Adding Parameters to Commands

This lesson shows you how to use parameters in your commands. Here are the objectives of this lesson:

- Understand what a parameter is.
- Be informed about the benefits of using parameters.
- Learn how to create a parameter.
- Learn how to assign 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!
SqlCommand cmd = new SqlCommand(
        "select * from Customers where city = '" + inputCity + "'";
```

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 an '@' symbol prefix on the parameter name as shown below:

```
// 1. declare command object with parameter
SqlCommand cmd = 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
SqlParameter param  = 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 1: Adding Parameters to Queries

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

class ParamDemo
{
        static void Main()
        {
                // conn and reader declared outside try
                // block for visibility in finally block
                SqlConnection conn   = null;
                SqlDataReader reader = null;

                string inputCity = "London";

                try
                {
                        // instantiate and open connection
                        conn =  new

        SqlConnection("Server=(local);DataBase=Northwind;Integrated
Security=SSPI");
                        conn.Open();

                        // don't ever do this!
//                      SqlCommand cmd = new SqlCommand(
//                              "select * from Customers where city = '" +
inputCity + "'";

                        // 1. declare command object with parameter
                        SqlCommand cmd = new SqlCommand(
                                "select * from Customers where city = @City",
conn);

                        // 2. define parameters used in command object
                        SqlParameter param  = 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"]);
                }
            }
            finally
            {
                // close reader
                if (reader != null)
                {
                        reader.Close();
                }

                // close connection
                if (conn != null)
                {
                        conn.Close();
                }
            }
        }
}
```

The code in Listing 1 simply 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.

Lesson 07: Using Stored Procedures

This lesson shows how to use stored procedures in your data access code. Here are the objectives of this lesson:

- Learn how to modify the SqlCommand object to use a stored procedure.
- Understand how to use parameters with stored procedures.

Introduction
A stored procedures 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
SqlCommand cmd  = 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
SqlCommand cmd  = 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 1 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 1: Executing Stored Procedures

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

class StoredProcDemo
{
        static void Main()
        {
                StoredProcDemo spd = new StoredProcDemo();

                // run a simple stored procedure
                spd.RunStoredProc();

                // run a stored procedure that takes a parameter
                spd.RunStoredProcParams();
        }

        // run a simple stored procedure
        public void RunStoredProc()
        {
                SqlConnection conn = null;
```

```
                SqlDataReader rdr  = null;

                Console.WriteLine("\nTop 10 Most Expensive Products:\n");

                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
                        SqlCommand cmd  = 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"]);
                        }
                }
                finally
                {
                        if (conn != null)
                        {
                                conn.Close();
                        }
                        if (rdr != null)
                        {
                                rdr.Close();
                        }
                }
        }

        // run a stored procedure that takes a parameter
        public void RunStoredProcParams()
        {
                SqlConnection conn = null;
                SqlDataReader rdr  = null;

                // typically obtained from user
                // input, but we take a short cut
                string custId = "FURIB";
```

```
                Console.WriteLine("\nCustomer Order History:\n");

                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
                        SqlCommand cmd  = 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"]);
                        }
                }
                finally
                {
                        if (conn != null)
                        {
                                conn.Close();
                        }
                        if (rdr != null)
                        {
                                rdr.Close();
                        }
                }
        }
}
```

The *RunStoredProc* method in Listing 1 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.

Tutorial Source: http://www.csharp-station.com/Tutorials/AdoDotNet/Lesson01.aspx