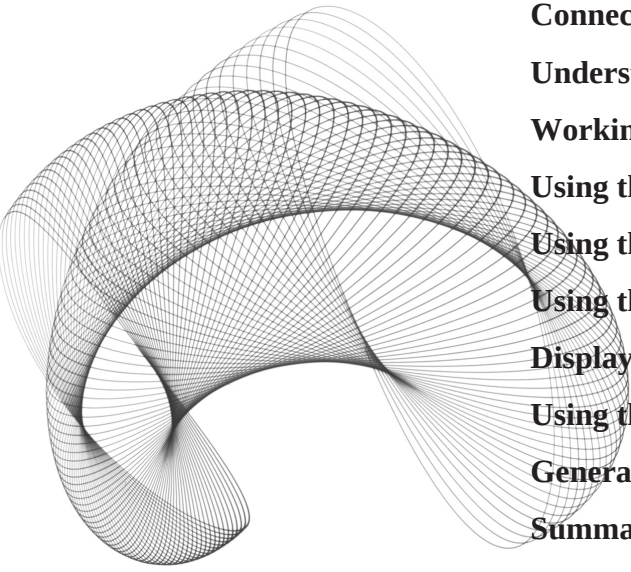
CHAPTER

5

Building Database Applications with ADO.NET

IN THIS CHAPTER:



The current version of ADO incorporates revisions designed to accommodate the world we live in today. In the 21st century, information is big business. Speedy access to mountains of data by thousands or even millions of simultaneous users is the reality. ADO.NET was revised to mirror this reality.

Previous versions of ADO were based largely on a *connected* model. Each user held a connection to a data source while interacting with the data. The result were bottlenecks caused by the large number of possible physical connections. Assuming clients are PCs connected to the Internet and the servers are web servers, then database problems can be exacerbated by distance and bandwidth.

Generally, the number of transactions per interval of time can be used to determine total throughput. At one completed transaction per minute, an application could support 1,440 transactions per day. One transaction per second and an application could support 86,400 transactions per day. Clearly, these numbers are not in the millions. ADO.NET was revised to address the problem of limited physical connections to a data source, to increase reliability, and to work in the world as it exists today—a connected world.

The current ADO.NET is based on a disconnected model and centers around the DataSet and XML. In short, the ADO.NET model follows the pattern of connecting to the data source, performing a short transaction, and disconnecting from the data source. There are some new capabilities and classes that were introduced with ADO.NET. This chapter will demonstrate how to use ADO.NET to write database applications.

Demonstrated Topics

Chapter 2 introduced the subject of Reflection by demonstrating how to explore the CLR. A tools provider could use such a utility to completely document the CLR and provide a reference application that associated specific aspects of the CLR with example code. For this concept to work, we would need to write the information we discovered by Reflection to a database and then add code examples to that database.

Based on a discussion I had with a Microsoft program manager, Microsoft has an internal application that fills the role of a resource tool for developers internally working with .NET. Fortunately, Microsoft has released Rotor, which is the shared source code for the common language infrastructure (CLI), a significant part of the base classes for .NET. Rotor is available for download from Microsoft.

We can create our own reference application as a reasonable means of demonstrating ADO.NET. The demonstrated topics in this chapter are set against the backdrop of a CLR reference application and will show you how to

- Use connections
- Use adapters
- Program with the new DataSet class
- Fill and interact with the DataTable
- Use the DataView class

- Speed up database access with the DataReader
- Execute SQL commands using the Command object
- Automatically generate SQL statements with the CommandBuilder
- Create data bound graphical user interfaces with the DataGrid

The Secondary Topics section will borrow from the demonstrated topics and describe how to use the DataSet as a return type for an XML Web Service, binding data to controls on Web Pages and inheriting the TraceListener to facilitate debugging and testing.

A Quick Review of ADO.NET Namespaces

ADO.NET is comprised of assemblies, namespaces, and classes that are part of the bigger .NET Framework. The main namespace for ADO.NET is the System.Data namespace. System.Data contains classes like the DataSet and DataTable. Within the System.Data namespace is System.Data.Common, System.Data.OleDb, System.Data.SqlClient, and System.Data.SQLTypes. Additionally, System.XML is fundamental to ADO.NET and to .NET in general.

The System.Data.Common namespace contains classes that are shared by ADO.NET providers. For example, both the System.Data.OleDb and System.Data.SQLClient namespaces contain adapters that inherit from a common adapter in the System.Data.Common namespace.

System.Data.OleDb and System.Data.SQLClient are namespaces that contain symmetric capabilities. The SQLClient namespace contains classes for working with MS SQL Server 7.0 or higher databases, and the OleDb namespace contains classes for working with all other OleDb-compatible databases, including MS Access.

System.Data.SQLTypes contains classes that represent native SQL data types.

Last but not least is System.XML. XML is used to describe data. For example, if you specify the DataSet as a return type for a Web Service, then the DataSet will be serialized as XML to facilitate transporting the DataSet. XML is used to define data schemas (XSD schemas).

As we proceed with the examples in this chapter, I will indicate where specific classes come from, and you can use this short section as a resource to explore additional information about ADO.NET.

Let's approach ADO.NET systematically, beginning with the first thing we must do: create a connection to a data source.

Connecting to DataSources

Programmers experienced with prior versions of ADO know that ADO supported a disconnected database model. However, ADO prior to .NET was fundamentally a connected model and was not based on XML. The disconnected ADO.NET will not hold

connections—we say it is disconnected—and uses XML, which makes it easy to move data across networks because XML is just hypertext.

However, you will need to create and use a connection to get at the data. You can create an SqlConnection for MS SQL Server 7.0 data sources or higher or an OleDbConnection for any other data source that supports OLE DB.

To optimize connection usage, you will want to take advantage of connection pooling. Connection pooling is a collection, or pool, of connections that applications can share. An OleDbConnection uses connection pooling automatically. An SqlConnection manages connection pooling implicitly. An SqlConnection that uses the same connection string can be pooled. You must close connections to take advantage of connection pooling.

Connecting to an OLE DB Data Store

ADO.NET creates an environment where the conditions of working with various data sources are similar from the perspective of the code you write. To learn how to program using ADO.NET, you can use any data source that is available. One such data source that supports OLE DB is the Microsoft Access Jet Engine. (MS Access ships with Microsoft Office Professional.) We'll use Access as our OLE DB example.

To connect to an OLE DB provider you will need to provide a specific connection string. There are a couple of good strategies that you can employ to obtain a working connection string. If you are adding the connection in the presentation layer, then you can drag an OleDbConnection right out of Server Explorer onto the Windows Form or Web Form (see Figure 5-1). Another good strategy for creating a second string has to do with creating a Microsoft Data Link file.

If you create a text file with a .UDL extension (for example, in Windows Explorer) and double-click that file, then you can use the Data Link Properties applet (see Figure 5-2) to configure a connection. The Data Link Properties applet provides you with a visual interface to create the connection. Complete the information on each tab and the .UDL file will contain a valid connection string. To create a connection to a Microsoft Access 2000 or 2002 database, follow these steps:

1. Create a blank text file with a .UDL extension. Double-click the file to open the file with the Data Link Properties applet.
2. On the Provider tab, select the Microsoft Jet 4.0 OLE DB Provider. Click the Next button, shown in Figure 5-2.
3. On the Connection tab, use the browse button—a button with an ellipses caption—to browse to your database file. Access databases have an .MDB extension. (You can use Windows Explorer to search for an Access database.) You can ignore item 2, leaving the default user name "Admin" in the user name field unless you know this isn't valid for the database you selected.
4. You can use the Test Connection button to determine if you have a valid connection.
5. If the Test succeeds, then you can click OK to save the configuration changes.

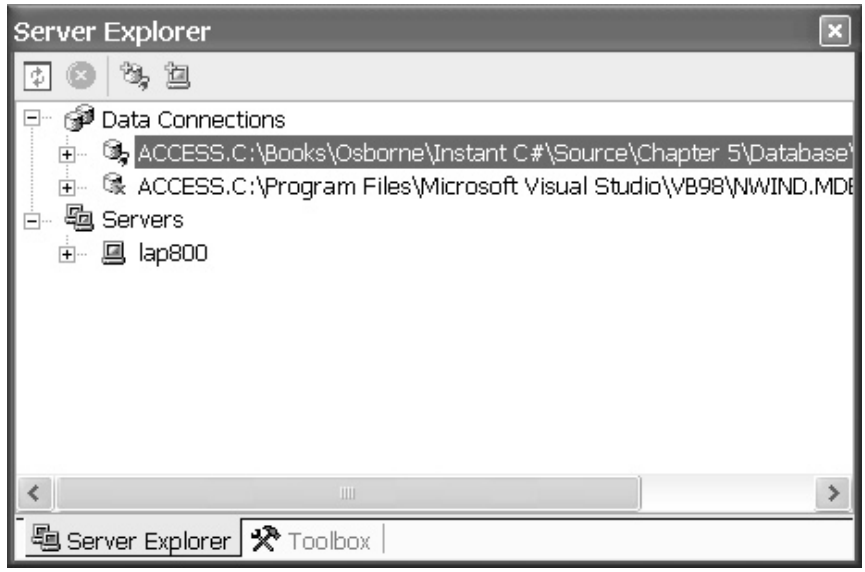


Figure 5-1 *Drag a connection from the Data Connections in Server Explorer to automatically add an OleDbConnection component to your project.*

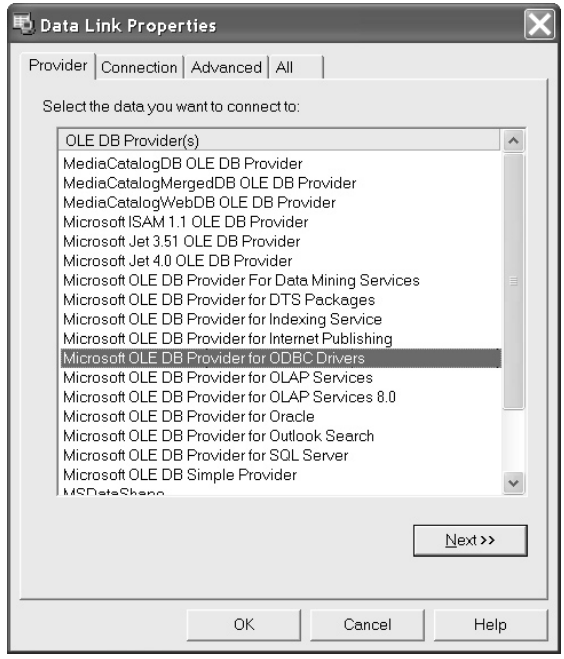


Figure 5-2 *Use the Data Link Properties applet to quickly and accurately configure a connection string.*

The Advanced tab of the Data Link Properties applet allows you to specify access permissions. The default is shared, read-write access to the database through this connection. The All tab contains name and value pairs that allow you to modify every connection value. Generally, the defaults will do, unless you have a specific reason for modifying initialization values, such as Jet OLEDB: Encrypt Database.

After you close the Data Link Properties applet, you open the .UDL file with Notepad and copy the connection string created by the applet. Here is an example of the connection string value created by the UDL applet for the Reference.mdb sample database available with the source code for this book.

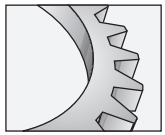
```
Provider=Microsoft.Jet.OLEDB.4.0;DataSource=C:\Temp\Reference.mdb;
Persist Security Info=False
```

Wrap the preceding statement in quotes and you can use it to initialize an OleDbConnection object without having to remember the provider name, Microsoft.Jet.OLEDB.4.0. Listing 5-1 demonstrates how to write code that will open the database described by our example connection string.

Listing 5-1 *The following demonstrates how to open an OleDbConnection and ensure it is closed using a resource protection block.*

```
public static void TestConnection()
{
  OleDbConnection connection = new OleDbConnection();
  connection.ConnectionString =
    @"Provider=Microsoft.Jet.OLEDB.4.0;" +
    @"Data Source=C:\Temp\Reference.mdb;" +
    @"Persist Security Info=False";

  connection.Open();
  try
  {
    Console.WriteLine(connection.State.ToString());
    Console.ReadLine();
  }
  finally
  {
    connection.Close();
  }
}
```



TIP

Strings in C# can be @-quoted or quoted. A quoted string will treat a single backslash as an escape character and a double backslash as a backslash. An @-quoted string will prevent escape characters from being processed.

Add a *using* statement to refer to the System.Data.OleDb namespace to use the OleDb providers in Listing 5-1. You can create the OleDbConnection object passing the connection string to the constructor or assigning the connection string to the ConnectionString property after the object is created. Invoke the Open method to open the connection. The code in the *try* part of the *try finally* handler—also referred to as a *resource protection block*—represents work. In Listing 5-1, we are simply writing the state of the connection object to the console. The *finally* block is always invoked, which is what we want. We always want to close the connection, and the *finally* block will ensure that the connection is closed even in the event of an exception.

Connecting to an MS SQL Server Data Store

From the perspective of the code needed to connect an SQL database, the code is almost identical to Listing 5-1. The biggest difference resides in the connection string. The connection string will need to contain information that is relevant to an MS SQL Server database. You can use the same two techniques described in the previous section—drag a connection from Server Explorer or create a .UDL file and use the Data Link Properties applet—to define a connection string to an SQL Server database.

We can use almost the identical code to that found in Listing 5-1 to open a connection to an SQL Server database. (The Microsoft Desktop Engine [MSDE] is a good SQL Sever database to use in a development environment, because you can perform initial software writing on a disconnected PC. This is especially useful when you are programming at the beach or in a hammock.) When connecting to an SQL Server database, you will need to include the System.Data.SqlClient namespace and create an instance of the SqlConnection object.

```
SqlConnection connection = new SqlConnection();
connection.ConnectionString =
  @"data source=LAP800\VSDOTNET;initial catalog=master;" +
  @"integrated security=SSPI;persist security info=False;" +
  @"workstation id=LAP800;packet size=4096";
```

There are huge benefits to be reaped when using a well-architected framework. Let's take a moment to look at one such benefit that is offered to us in ADO.NET.

Using ADO.NET Interfaces to Declare Types

A reasonable thing you may want to do is to make it easy to switch between data providers. There are several scenarios where this is likely, and I will describe one next.

Working on a project in Portland, Oregon, we were building an enterprise solution in C# to IBM's Universal Database. Residing in Michigan, I found it nice to occasionally go home and telecommute. (It helps me get caught up on yard work.) I want to work from the home office—for the multi-processor workstations with flat screens, a great library, a huge office chair, and, best of all, family—but I don't want to install every database a customer may be using. Instead, I will write my applications to allow me to quickly and simply switch to an alternate data provider. Combine ADO.NET with CASE tools for databases and you can easily replicate a production database in a non-production environment. Of course, if your

employer doesn't buy my arguments, then you could present the alternate database scenario as a flexibility issue.

The basic idea is that you define methods that return ADO.NET interfaces (instead of specific provider classes) and then construct a specific provider class. You can combine CASE tools to replicate a production database with ADO.NET interfaces rather than classes and a Boolean switch managed with an external XML file, and you have a versatile application-development environment and database heterogeneity.

Switching Between Databases Using an IDbConnection

Listing 5-2 demonstrates how we can declare variables as interfaces realized by the ADO.NET providers and create an instance of a specific provider based on dynamic criterion. Listing 5-2 uses a conditional compiler directive to switch between databases. This solution requires a recompile. In a moment, I will demonstrate how to achieve the same result without recompiling.

Listing 5-2 *Use the IDbConnection interface to declare your types and you can assign any ADO.NET connection that realizes the interface.*

```
1:  public class MultiConnection
2:  {
3:    private static string GetConnectionString()
4:    {
5:      #if DEBUG
6:      return @"Provider=Microsoft.Jet.OLEDB.4.0;" +
7:             @"Data Source=C:\Temp\Refernce.mdb;" +
8:             @"Persist Security Info=False";
9:      #else
10:     return @"data source=LAP800\VSDOTNET;initial catalog=master;" +
11:            @"integrated security=SSPI;persist security info=False;" +
12:            @"workstation id=LAP800;packet size=4096";
13:     #endif
14: }
15:   private static IDbConnection GetConnection()
16:   {
17:     #if DEBUG
18:     return new OleDbConnection(GetConnectionString());
19:     #else
20:     return new SqlConnection(GetConnectionString());
21:     #endif
22:   }
23:
24:   public static void Test()
25:   {
26:     IDbConnection connection = GetConnection();
```

```
27:     connection.Open();
28:     try
29:     {
30:       Console.WriteLine(connection.State.ToString());
31:       Console.ReadLine();
32:     }
33:     finally
34:     {
35:       connection.Close();
36:     }
37:   }
38: }
```

The only real change made to the code is to declare the connection as an IDbConnection instead of an SqlConnection or an OleDbConnection. The balance of the code is consistent with the previous example in Listing 5-1. You can toggle the conditional compiler statements on lines 5 and 17 of Listing 5-2 to switch between the SQL and OLE databases.

A common framework makes it possible to write code once, and switch between something as significant as a database without writing multiple versions of the code for each database. A great framework puts this kind of flexibility at your fingertips. The .NET framework is a superlative framework. We can use a BooleanSwitch to externalize switching databases without recompiling.

Switching Between Data Providers with a BooleanSwitch

A second scenario where it may be beneficial to allow a user to switch between databases is when you are writing a consumer product, and you don't want to force your customer to use a specific database provider. Suppose you write a consumer application and want to allow the customer to choose between databases. By the time the customer has the application, it is too late to recompile. You can use a BooleanSwitch internally and an XML file externally to support switching between data providers (as well as supporting any kind of externalized dynamic switching).

Listing 5-3 is a revision of Listing 5-2. The revision demonstrates the addition of a BooleanSwitch, and Listing 5-4 provides the listing for the XML file that defines the switch.

Listing 5-3 *This code (a revision of Listing 5-2) demonstrates using a BooleanSwitch.*

```
1:  public class SwitchedMultiConnection
2:  {
3:    private static BooleanSwitch Switch =
4:      new BooleanSwitch("Provider",
5:      "Supports switching between data providers");
6:
7:    private static string GetConnectionString()
8:    {
9:      if( Switch.Enabled )
```

```
10:     {
11:       return @"Provider=Microsoft.Jet.OLEDB.4.0;" +
12:         @"Data Source=C:\Books\Osborne\Instant C#" +
13:         @"\Source\Chapter 5\Database\Reference.mdb;" +
14:         @"Persist Security Info=False";
15:     }
16:     else
17:     {
18:       return @"data source=LAP800\VSDOTNET;initial catalog=master;" +
19:         @"integrated security=SSPI;persist security info=False;" +
20:         @"workstation id=LAP800;packet size=4096";
21:     }
22:   }
23:
24:   private static IDbConnection GetConnection()
25:   {
26:     if (Switch.Enabled)
27:     {
28:       return new OleDbConnection(GetConnectionString());
29:     }
30:     else
31:     {
32:       return new SqlConnection(GetConnectionString());
33:     }
34:   }
35:
36:   public static void Test()
37:   {
39:     IDbConnection connection = GetConnection();
40:     connection.Open();
41:     try
42:     {
43:       Console.WriteLine(connection.State.ToString());
44:       Console.ReadLine();
45:     }
46:     finally
47:     {
48:       connection.Close();
49:     }
48:   }
49: }
```

Lines 3 through 5 of Listing 5-3 declare and initialize a BooleanSwitch as a static member. Making the switch static means that we will only instantiate one BooleanSwitch and share it

between all instances of the class using the switch. Lines 9 and 26 use the BooleanSwitch in place of the conditional compiler directive to determine whether we should use the OleDb or SqlClient connection.

There are a few differences between a BooleanSwitch and conditional compiler directives. The conditional compiler directive must be changed and recompiled. The code that fails evaluation is excluded from the compiled assembly. Code in both parts of the BooleanSwitch code block is written to the compiled assembly, and to switch between different versions of the code we change the external definition of the switch in an XML file without recompiling. The .config file containing the switch definition is provided in Listing 5-4.

Listing 5-4 *This code defines the .config file containing the BooleanSwitch.*

```
<configuration>
  <system.diagnostics>
    <switches>
      <add name="Provider" value="1" />
    </switches>
  </system.diagnostics>
</configuration>
```

Name the configuration file the same name as the assembly that will use it, adding a .config extension. For example, if your assembly is named myapp.exe, then your configuration file will be myapp.exe.config. Create the configuration file in the same directory as the one containing the assembly. A complete description of the contents of a BooleanSwitch defined in a configuration file is provided in Chapter 7, in the section entitled "Using Switches."

Combining provider interfaces with BooleanSwitch objects provides us with a convenient way to switch between providers without recompiling our code. The next piece of the ADO.NET puzzle is the adapter.

Understanding the Role of the Adapter

ADO.NET separates data access from data manipulation. Adapters are part of the data access layer of ADO.NET. Adapters are used to move data between a connection and classes that store cached data for manipulation, including the DataSet and DataTable. To read data from a connection to a data source, you can use the Fill method. To read the schema only, you can invoke the FillSchema method, and Update will accept the modified revisions to data in a DataTable or DataSet and update the data source with the revisions.

There is an OleDbDataAdapter for OLE DB providers, an SqlDataAdapter for MS SQL Server providers, and an IDbAdapter interface that allows you to write provider-independent code, as we did with IDbConnection in the preceding section.

Adapters are initialized with SQL and connection objects. The adapter provides the bridge from the connection to the data manipulation objects. In this section, I will demonstrate the Fill, FillSchema, and Update methods of an adapter.

Initializing an Adapter

Adapters are initialized with an SQL SELECT command and a connection object of the same provider type. Use an OleDbDataAdapter with an OleDbConnection and a SqlDataAdapter with a SqlConnection. (I won't repeat this information. Assume that, if one or the other of the OleDb provider or SqlClient provider has a specific class, there is a symmetric class for the other provider type.)

To initialize an adapter, create an instance of the adapter with an SQL SELECT command and a like connection. Here is an example of an adapter object being created with an OleDbConnection referring to the Reference.mdb MS Access database available with this book.

```
OleDbConnection connection = new OleDbConnection();
connection.ConnectionString =
  @"Provider=Microsoft.Jet.OLEDB.4.0;" +
  @"Data Source=C:\Temp\Reference.mdb;" +
  @"Persist Security Info=False";

OleDbDataAdapter adapter = new OleDbDataAdapter(
  "SELECT * FROM METHOD", connection);
```

The last statement demonstrates one of the four possible ways to initialize an adapter object. The other three overloaded constructors for the adapter are variations of the one shown; you can use the Visual Studio help documentation for examples of the other variations.

Connection Pooling Strategy

If you recall, we spoke about connection pooling earlier. Connection pooling is implicitly based on the connection string. To take advantage of the more optimal use of connections via connection pooling, you can use a simple technique for ensuring that your code isn't littered with literal connection strings. Listing 5-5 provides an example of a factored class that supports maintaining only one instance of a connection string.

Listing 5-5 *This code demonstrates how to use a class to ensure that you have only one instance of a connection string.*

```
1:  public class FactoredConnection
2:  {
3:    private static BooleanSwitch booleanSwitch =
4:      new BooleanSwitch("PKimmel",
5:      "Used to switch between connection strings");
6:
7:    public static string GetOleDbConnectionString()
8:    {
9:      if(booleanSwitch.Enabled)
10:     {
11:       return @"Provider=Microsoft.Jet.OLEDB.4.0;" +
```

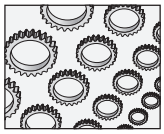
```
12:              @"Data Source=C:\Books\Osborne\Instant C#" +
13:              @"\Source\Chapter 5\Database\Reference.mdb;" +
14:              @"Persist Security Info=False";
15:     }
16:     else
17:     {
18:       return @"Provider=Microsoft.Jet.OLEDB.4.0;" +
19:              @"Data Source=C:\Temp\Reference.mdb;" +
20:              @"Persist Security Info=False";
21:     }
22: }
23:
24:   public static string GetSqlConnectionString()
25:   {
26:     if(booleanSwitch.Enabled)
27:     {
28:       return @"data source=LAP800\VSDOTNET;initial catalog=master;" +
29:              @"integrated security=SSPI;persist security info=False;" +
30:              @"workstation id=LAP800;packet size=4096";
31:     }
32:     else
33:     {
34:       throw new Exception(
35:         "Make sure you have a MS SQL Server instance available.");
36:     }
37:   }
39:   public static OleDbConnection GetOleDbConnection()
40:   {
41:     return new OleDbConnection(GetOleDbConnectionString());
42:   }
43:
44:   public static SqlConnection GetSqlConnection()
45:   {
46:     return new SqlConnection(GetSqlConnectionString());
47:   }
48: }
```

Again, in Listing 5-5 we use a BooleanSwitch—on lines 9 and 26—to support easy switching between versions of the connection string. This is consistent with an approach we might take for a connected versus a disconnected development environment. Notice that the *else* condition for GetSqlConnectionString throws an exception. You will need to ensure that you have access to MS SQL Server or MSDE installed on your desktop and have an instance accessible to your application.

Using the approach demonstrated in Listing 5-5, we can be sure that we are facilitating connection pooling by using the same connection string. This approach does not prohibit you from creating a new instance of a connection without the FactoredConnection class.

General Programming Strategy

The preceding section demonstrated a FactoredConnection class that facilitates connection pooling by creating a separate class containing the connection string. There is another strategic reason to write code like that demonstrated. I refer to it as *Kimmel's Theory of Convergent Code*.



NOTE

Kimmel's Theory of Convergent Code: *Code that converges on a single instance of an algorithm or class is good because it promotes reuse, consistency, and a high degree of re-orchestrated dynamic behavior.*

The basic idea behind convergent versus divergent code is that the smallest piece of code that can be reused without replicating the code is the *method*. Anything below a method must be duplicated to be reused. That is, lines of code must be copied and pasted to be reused. Copied and pasted code is divergent code for the simple reason that there is more than one instance of an algorithm. The negative result of divergent code is that for each instance of the divergent code, a programmer must replicate, individually test, and separately maintain the individuated lines of code. The result is that divergent code tends to decay over time according to the number of times the code is copied and pasted. The result is that divergent code tends toward semantically similar operations diverging in behavior, yielding the perception of unreliable performance.

To summarize: relative to our FactoredConnection class, we have only one instance of the connection string to maintain. If the connection changes, we propagate the change in one place. If we want the behavior of the FactoredConnection to change then, we again only need to change the code in place. Consider the case where we want to read the connection string from an external resource like the registry. Again, we only need change the code in one place.

Convergent code in the form of methods yields the following good rule of thumb: prefer singular, short, well-named, highly reusable methods to lines of code, which represent plural, monolithic, non-reusable methods. William Opdikes' doctoral dissertation from 1990 was the impetus for the subject of *refactoring*, which supports my theory of convergence. You can read about refactoring in Martin Fowler's superlative *Refactoring: Improving the Design of Existing Code* (Addison-Wesley).

Invoking the Adapter Fill Method

The Fill method is used to move data between a connection and a DataSet. To fill a DataSet, you will need to create a connection and an adapter and invoke the adapter's Fill method. A DataSet object is passed as an argument to the Fill method. You do not need to specifically create an instance of a connection to use an adapter. You may pass a connection string instead of a connection object to the adapter, and the adapter will internally create an instance

of a connection and then open and close the connection automatically. Here are a couple of examples that demonstrate using the Fill command.

```
public static void TestFill()
{
  OleDbConnection connection = FactoredConnection.GetOleDbConnection();
  OleDbDataAdapter adapter = new OleDbDataAdapter(
    "SELECT * FROM METHOD", connection);

  DataSet dataSet = new DataSet();
  connection.Open();
  try
  {
    adapter.Fill( dataSet );
  }
  finally
  {
    connection.Close();
  }

  WalkDataSet( dataSet );
}
```

The preceding example verbosely creates an OlDbConnection, OleDbDataAdapter, and a DataSet, opens the connection and then fills the DataSet. WalkDataSet represents a method that performs useful work. (You can find the code for WalkDataSet in the ADOSampleCode.sln available online at **www.osborne.com**.)

The second example, which follows, demonstrates a concise version that passes a connection string to the OleDbDataAdapter constructor, and the adapter will be responsible for opening and closing the connection.

```
private static void TestFill2()
{
  OleDbDataAdapter adapter = new OleDbDataAdapter(
    "SELECT * FROM METHOD",
    FactoredConnection.GetOleDbConnectionString());

    DataSet dataSet = new DataSet();
    adapter.Fill( dataSet );
    WalkDataSet( dataSet );
}
```

(The FactoredConnection class was introduced in the earlier section "Connection Pooling Strategy.") The preceding example demonstrates a concise way to fill a DataSet. More important, notice that in both instances we are operating on the DataSet after the connection is closed. (The first example closes the DataSet explicitly, and the second example closes the DataSet after the Fill operation.) This demonstrates the connectionless mode of operation in ADO.NET. The data has been cached in the DataSet and is available completely independent of the connection and adapter.

Invoking the Adapter FillSchema Method

If you are performing a large number of insert statements and aren't interested in existing rows of data, then you can request the schema only. To read schema—a description of a table—into a DataSet only, employ the FillSchema method.

As is true with the Fill method, you can actually fill a single DataTable or add a table to a DataSet. The DataSet is a collection of tables and optional relationships between those tables. (Refer to the later sections "Working with the DataSet" and "Using the DataTable" for more information.) FillSchema takes a DataSet or DataTable argument and a SchemaType enumerated value, either Mapped or Source. Here is an example of a method that reads the description of a table only into a DataSet.

```
public static void TestFillSchema()
{
  OleDbDataAdapter adapter = new OleDbDataAdapter(
    "SELECT * FROM METHOD",
    FactoredConnection.GetOleDbConnectionString());

  DataSet dataSet = new DataSet();
  adapter.FillSchema(dataSet, SchemaType.Source);
}
```

The DataSet in the preceding fragment will contain a single table with columns only; the columns represent the schema information. The enumerated value SchemaType.Source—of two possible values, the other being SchemaType.Mapped—instructs the adapter to use the schema described by the source table. The alternative is to use the schema transformed by column mappings. (Refer to the section "Working with the DataSet" later in the chapter for more information on column mappings.)

Updating Changes to Data

An adapter plays the role of bridge between a connection and a DataSet. Just as we used an adapter to read data from a data source via a connection into a DataSet, we use the adapter to update changes made to the data. Updating includes SQL UPDATE, SQL INSERT, and SQL DELETE operations.

The means by which we instruct the adapter to update data is to provide SQL commands for the operations we want to perform. For example, if we have added new rows, then we need to provide an INSERT command to the adapter before we call the Update method. The basic steps for updating data are to open a connection, create an adapter providing an SQL SELECT statement and the connection as initial values, fill a DataSet with data, modify the data, create SQL commands that describe how to perform updates, and invoke the adapter Update method. One version of these steps is demonstrated in Listing 5-6.

Listing 5-6 *This code demonstrates how to update a DataSet using an adapter.*

```
1:  public static void TestUpdate()
2:  {
3:    OleDbConnection connection =
4:      FactoredConnection.GetOleDbConnection();
5:
6:    OleDbDataAdapter adapter = new OleDbDataAdapter(
7:      "SELECT * FROM METHOD", connection);
8:
9:    DataSet dataSet = new DataSet();
10:   connection.Open();
11:   adapter.Fill(dataSet);
12:   connection.Close();
13:
14:   DataRow row = dataSet.Tables[0].NewRow();
15:   row["Name"] = "TestUpdate";
16:   dataSet.Tables[0].Rows.Add(row);
17:
18:   OleDbCommandBuilder commandBuilder =
19:     new OleDbCommandBuilder(adapter);
20:
21:   connection.Open();
22:   adapter.Update(dataSet);
23:   connection.Close();
24: }
```

Lines 3 and 4 of Listing 5-6 create a connection object. Lines 6 and 7 use the connection object to create the adapter. Line 9 creates an instance of a DataSet. Line 10 opens the connection, line 11 fills the DataSet, and line 12 closes the connection. We don't need an open connection to manage the DataSet, and it is better if we close the connection immediately. In a real application, it is unlikely that the code actually adding data will be this simple.

Lines 14 through 16 of Listing 5-6 demonstrate how to add a new row to the tables in a DataSet. Line 14 adds the row. Line 15 updates one field in the row, and line 16 adds the row to the table's collection of Rows using the Rows.Add command.

Line 18 of Listing 5-6 uses the adapter to construct an OleDbCommandBuilder. The command builder automatically uses the schema information for a table to generate INSERT, DELETE, and UPDATE SQL commands. Refer to the section "Generating SQL with the CommandBuilder" later in this chapter for more information on the OleDbCommandBuilder.

Lines 21 through 22 of Listing 5-6 opens the connection, updates the database, and closes the connection. The DataSet knows that we inserted a new row; hence, the adapter knows to use the adapter's InsertCommand property to write the row of data we added on lines 14 through 16.

Working with the DataSet

The DataSet is an evolution of the ADO RecordSet. The DataSet is a connectionless repository for ADO.NET, and you can use DataSet objects without ever connecting to a data source. We have only briefly introduced the DataColumn, DataRow, and DataTable, but these objects can exist and you can programmatically interact with them without ever connecting to a data source.

In general, the DataSet model is consistent with how we think about a relational database. The DataSet contains a DataRelationCollection and a DataTableCollection. The DataTableCollection contains a DataRowCollection, DataColumnCollection, ChildRelations, ParentRelations, and ExtendedProperties. The DataRowCollection contains instances of DataRow objects, and the DataColumnCollection contains DataColumn objects. There is also a view of data represented by the DataView class.

You can use the DataSet and contained objects in the traditional way by connecting to data providers as demonstrated earlier in this chapter, or you can use the DataSet as an in-memory repository to store data in an organized way. This next section demonstrates how to employ aspects of the DataSet.

Adding DataTable Objects to a DataSet

When you create a DataSet object and fill it from a DataAdapter based on a single SQL SELECT statement, what really happens is that a DataTable is created and is added to the DataSet's DataTableCollection. The premise is that for ADO.NET to support a connectionless model for database management, it must support storing and managing data in relations that are likely to exist.

The most common relationship is the simple master-detail relationship. One table plays the role of the master table and the other table plays the role of detail table. In a typical master-detail relationship, you will need a master table, a detail table, and a DataRelation. The AssemblyViewer.sln for this chapter available for download demonstrates one such relationship between a table containing CLR TypeInfo objects and ConstructorInfo objects. Listing 5-7 demonstrates how to fill DataTable objects and add those objects to a DataSet.

Listing 5-7 *This code demonstrates how to add DataTable objects to a DataSet.*

```
1:  OleDbConnection connection = new OleDbConnection(connectionString);
2:  OleDbDataAdapter adapter = new OleDbDataAdapter(
3:    "SELECT * FROM TYPE", connection);
4:  DataSet dataSet = new DataSet();
5:
6:  DataTable typeTable = new DataTable("TYPE");
7:  adapter.Fill( dataTable );
8:  dataSet.Tables.Add(typeTable);
9:
10: DataTable constructorTable = new DataTable("CONSTRUCTOR");
```

```
11: adapter = new OleDbDataAdapter(
12:   "SELECT * FROM CONSTRUCTOR", connection);
13: adapter.Fill( constructorTable );
14:
15: dataSet.Tables.Add(constructorTable);
```

Line 1 of Listing 5-7 creates a connection object. Because we don't explicitly open the connection, the adapter will take care of opening and closing the connection for us. (You have seen several examples of a connection string to the Reference.mdb database. The variable connectionString was used to represent the actual connection string.) The second statement creates an OleDbDataAdapter to bridge between the connection and a DataTable.

Line 4 of Listing 5-7 creates a DataSet. When we want to represent a collection of tables, we use the DataSet, although it is acceptable and possible to use a single DataTable if we only need one table. Line 6 creates a new DataTable object, and line 7 demonstrates how easy it is to fill a DataTable directly without using a DataSet. Line 8 adds the DataTable to our DataSet object. Lines 10 through 15 repeat the process for the CONSTRUCTOR table.

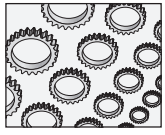
When the last statement in line 15 of Listing 5-7 finishes executing, the DataSet contains two tables. There is no relationship between the tables yet. We need to define the relationship as a separate object and add it to the DataSet.

Creating Master-Detail Relationships

In the preceding section, we added two tables to a DataSet. To logically join those tables, we need to create a DataRelation object and add that to the table. Listing 5-8 demonstrates the code that, when combined with the code in Listing 5-7, defines a master-detail relationship within the DataSet.

Listing 5-8 *Add a DataRelation to a DataSet.*

```
DataColumn parent =
  dataSet.Tables["TYPE"].Columns["UnderlyingSystemType"];
DataColumn child =
  dataSet.Tables["CONSTRUCTOR"].Columns["DeclaringType"];
DataRelation relation = new DataRelation(
  "TypesAndConstructors", parent, child);
dataSet.Relations.Add(relation);
```



NOTE

Generally, I prefer to name my related columns identically to facilitate identifying relationships between tables in the same database. However, the Reference.mdb database represents the types defined in the CLR, and the columns are created from the names of actual properties in the CLR. For example, UnderlyingSystemType is a property in the TypeInfo type defined in the CLR.

Just as with JOIN statements in SQL, a DataRelation relies on columns. In Listing 5-8 we initialize two DataColumn objects by selecting a column from each DataTable that represents a logical relationship between two disparate tables. In the Common Language Runtime, a TypeInfo object has an UnderlyingSystemType property that is logically related to the ConstructorInfo's DeclaringType property. We use the TYPE table's UnderlyingSystem Type column and the CONSTRUCTOR table's DeclaringType column to initialize a new DataRelation, providing a name and the two logically related columns as arguments to the DataRelation constructor.

When the last statement—dataSet.Relations.Add(relation)—runs in Listing 5-8, you have a logical master-detail relationship in the DataSet. This relationship is depicted Figure 5-3.

Figure 5-3 was created from the Relationships dialog in MS Access 2002 but is an accurate visualization of the DataSet after the code in Listing 5-7 and Listing 5-8 runs. Each table shown represents a DataTable, and the line between the tables represents the DataRelation object.

You can bind the DataSet to a Windows Forms or Web Forms DataGrid, and the DataGrid will accurately manage the master-detail aspects by presenting the data in a hierarchical relationship. You can explore the AssemblyViewer.sln for an example. The code to bind the DataSet to a DataGrid requires a single statement:

```
dataGrid1.DataSource = dataSet;
```

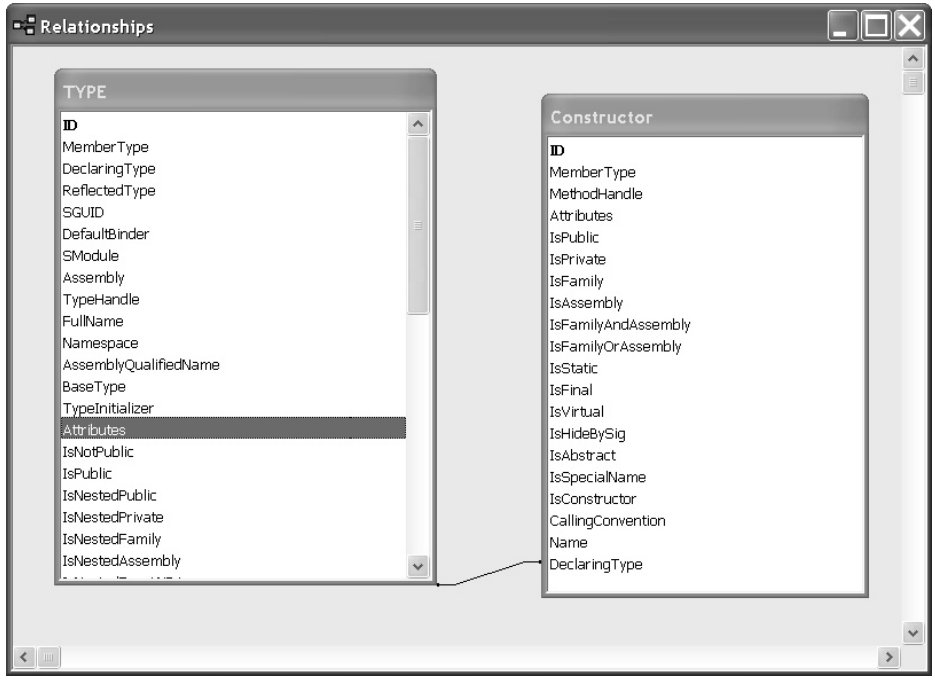
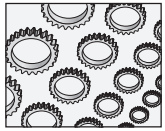


Figure 5-3 *The Relationships dialog in MS Access is an accurate depiction of the two DataTables related by a single DataRelation.*

Creating Data Column Mappings

Another common thing that you might want to do is clean up unsightly column names. I have encountered a large number of databases with obscure column names, like the NAME_SEQ_TS. It is often very difficult to figure out the role of such columns after the original DBA has left the project. Unfortunately, you may not always be in a position to re-design a database.



NOTE

Documenting and managing databases are good reasons for using CASE tools. ERwin, DataArchitect, and Select are all good CASE tools that make it easy to design, document, and maintain databases.

One such instance where we elected not to re-design the database was where the customer's motivation for re-implementing the system was not dissatisfaction with the existing implementation. The customer simply no longer wanted to be held hostage by a hardware vendor that charged exorbitant rent. Our mission statement was to provide a new system with behavior that duplicated the behavior of the existing system. There were supposed to be no new features. The existing system was implemented in Natural and Adabas, and we were porting it to C#, ASP.NET, and UDB. In essence, we were porting a non-object–oriented system implemented as a terminal application to a thin client Web application. Our charter did not really permit re-engineering the database, so we did the next best thing. We used column mappings to more clearly name older legacy columns that had obscure names. Listing 5-9 demonstrates how we can use DataTableMapping and DataColumnMapping objects in concert to support providing alternate names for columns.

Listing 5-9 *Using DataTableMapping and DataColumnMapping objects*

```
OleDbConnection connection = new OleDbConnection(connectionString);
OleDbDataAdapter adapter = new OleDbDataAdapter(
  "SELECT * FROM PARAMETER_TABLE", connection);
DataSet dataSet = new DataSet();

DataTableMapping tableMapping =
  adapter.TableMappings.Add("Parameter", "PARAMETER_TABLE");

tableMapping.ColumnMappings.Add("ParameterType", "Parameter Type");
adapter.Fill(dataSet, "Parameter");

dataGrid1.DataSource = dataSet.Tables[0];
```

Listing 5-9 creates a table mapping named Parameter that maps to the Reference.mdb PARAMETER_TABLE table. The DataColumnMapping is implicitly created when we invoke the DataTableMapping.ColumnMappings.Add method. The source table column ParameterType is mapped to the reader-friendly "Parameter Type" column. Invoking the

Fill method on adapter and using the DataTableMapping name will fill the data table with the columns from the source table, PARAMETER_TABLE, using the mapped column names.

Of course, you can accomplish this much with SQL statements that describe column aliases—usually with an *As* clause in the SELECT statement. The DataTableMapping and DataColumnMappings can be used for updates too. Listing 5-10 demonstrates how to use the mapped table and column to perform an update using the mapped values.

Listing 5-10 *A revised version of Listing 5-9 demonstrates how to perform an update against the mapped table and column.*

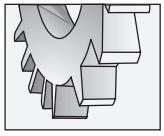
```
OleDbConnection connection = new OleDbConnection(connectionString);
OleDbDataAdapter adapter = new OleDbDataAdapter(
  "SELECT * FROM PARAMETER_TABLE", connection);
  DataSet dataSet = new DataSet();

DataTableMapping tableMapping =
  adapter.TableMappings.Add("Parameter", "PARAMETER_TABLE");

tableMapping.ColumnMappings.Add("ParameterType", "Parameter Type");
  adapter.Fill(dataSet, "Parameter");

// Demonstrates updating using the mapped objects
DataRow dataRow = dataSet.Tables[0].NewRow();
dataRow["Parameter Type"] = "Test";
dataSet.Tables[0].Rows.Add(dataRow);
adapter.InsertCommand = (new
  OleDbCommandBuilder(adapter)).GetInsertCommand();
adapter.Update(dataSet, "Parameter");

dataGrid1.DataSource = dataSet.Tables[0];
```



TIP

It is perfectly acceptable to create an inline object to avoid littering your code with temporary variables, as demonstrated by the OleDbCommandBuilder created inline in Listing 5-10. Use an inline object if you don't need a reference to that object later in the same code.

The comment beginning with the word "Demonstrates" is where the new code has been inserted to revise Listing 5-9. This code is very similar to performing an update without mappings. We request a new DataRow. We use the indexer for the DataRow to modify a single field by referring to the mapped column name. We add the modified row to the Rows collection and wrap it up by creating an OleDbCommandBuilder to write our SQL for us and invoking the OleDbDataAdapter.Update command. Remember to use the mapped table name for the update just as we did with the Fill operation.

Using the DataTable

The DataTable maps to a table in a database. You can also use a DataTable as an in-memory data store for data that will never see the inside of a database. As my good and smart friend Eric Cotter says, "Think of a DataTable as just a convenient way to store data."

Although you are not required to limit the use of a DataTable to database applications, the DataTable was primarily defined for this purpose. DataTable objects are primarily composed of DataRowCollection and DataColumnCollection objects. Logically, a DataTable looks like a single spreadsheet where the intersection of a column and row represents a field. Operations you are likely to want to perform on a DataTable include filling a table from a data source and updating that data source after you modify the data in the table. We have already looked at several examples demonstrating this use of the DataTable, and the AssemblyViewer.sln for this chapter has several more. Another task you may want to perform is to create and define a DataTable programmatically, including creating keyed and incremented columns.

The AssemblyViewer.sln defines several types that inherit from the DataTable. Each of these types is capable of creating an in-memory DataTable or a table in an Access database that represents a type object. For example, the PropertyTable defined in Database.cs is capable of reading all of the properties from a PropertyInfo object, defining a DataTable, and adding all of the PropertyInfo details to the DataTable. (For discussions on Reflection, see Chapter 2 and Chapter 11.)

The basic idea is that a type is passed to the PropertyTable's CreateNew method. Reflection is used to get all of the PropertyInfo—the property descriptions—objects for that type. A table is dynamically created using the property names and types of the PropertyInfo object, then each of the PropertyInfo objects representing the properties of the type passed to CreateNew is iterated, and the values of each of the PropertyInfo objects is added to the dynamic table. (I'd like to add the complete listing here for the AssemblyViewer.sln, but the Refactored Database module alone is 728 lines of code. You will have to download the example solution and step through the application to explore all of the code.) To facilitate your understanding, the following bulleted list describes the Reflection algorithm used with all of the members of a type.

- Determine the elements of the type you want to explore. In this instance, we will explore the Properties of a type only.
- Properties are described by the System.Reflection.PropertyInfo class.
- Iterate all of the properties of a PropertyInfo class, adding a column to a DataTable for each property defined by the PropertyInfo class. Every property will itself be described by the same properties.
- Iterate all of the properties of the type you want to explore, writing the value associated with each property's PropertyInfo.

A property is described by the PropertyInfo object. A PropertyInfo object has several properties that describe a property. (A bit of a tongue twister.) A property is described by MemberType, PropertyType, Attributes, IsSpecialName, CanRead, CanWrite, Name, DeclaringType, and ReflectedType. Every property in a type is initialized with a value for each of these elements of its type record, if you will. To create a table that describes a class relative to its properties, we can store the value for each property of the PropertyInfo object. When adding this data to a table it is helpful to add a keyed column and columns for each of the PropertyInfo properties. This is what the AssemblyViewer.sln does. We'll examine each part of this process in the sections that follow by walking through the code that creates the Property table in the Reference.mdb database.

Creating a DataTable Object

Creating the DataTable is easy. Ensure that you have the System.Data.dll assembly referenced, which it is by default in new projects. Add a *using* statement that refers to the System.Data namespace to the module that you will be declaring a DataTable in. Construct a new instance of a DataTable providing a name for the DataTable.

```
DataTable table = new DataTable("PROPERTY");
```

Recall that I said that you can use a DataTable independent of providers and DataSets. The preceding code is all you need to create a new DataTable.

Creating a Primary Key, Auto-Incremented Column

For our purposes, we can use a simple auto-increment column as the primary key for our property table. We add a DataColumn to the table created in the preceding section and provide some specific information for that data column. The updated code follows in Listing 5-11.

Listing 5-11 *This code demonstrates how to create an auto-incremented primary key column in a DataTable.*

```
DataTable table = new DataTable("PROPERTY");
DataColumn column = table.Columns.Add();
column.ColumnName = columnName;
column.AutoIncrement = true;
column.AutoIncrementSeed = 1;
column.AutoIncrementStep = 1;
column.Unique = true;
table.PrimaryKey = new DataColumn[]{column};
```

After we create the DataTable, we create a new column by invoking the DataTable.Columns.Add method. Provide a ColumnName as demonstrated, indicate that this column is auto- incremented by setting the DataColumn.AutoIncrement property to True. You can optionally set the DataColumn.AutoIncrementSeed and DataColumn.AutoIncrementStep

properties or use the default value 1 for each. The AutoIncrementSeed value is the starting number for the automatically incremented column, and the AutoIncrementStep value defines the value added to each increment. Indicate that the column must be unique, as demonstrated in the preceding fragment.

Finally, a DataTable's primary key can be composed of multiple columns. For this reason we must initialize the DataTable.PrimaryKey property with an array of DataColumn objects. In our example—on the last line of code in Listing 5-11—the array of columns that make up the key is composed of our single column.

Walking the Properties of the PropertyInfo Class

Adding your basic vanilla column is much easier than adding keyed and incremented columns. All we need to do is create a DataColumn object by requesting it from the DataTable and provide a name and data type for the column. Following this logic, we can create a DataTable that maps to a PropertyInfo record with the code in Listing 5-12.

Listing 5-12 *This code demonstrates how to create a table that mirrors the PropertyInfo class.*

```
DataTable table = new DataTable("PROPERTY");
DataColumn column = table.Columns.Add();
column.ColumnName = columnName;
column.AutoIncrement = true;
column.AutoIncrementSeed = 1;
column.AutoIncrementStep = 1;
column.Unique = true;
table.PrimaryKey = new DataColumn[]{column};

PropertyInfo[] infos = typeof(PropertyInfo).GetProperties(
  BindingFlags.Public | BindingFlags.NonPublic |
  BindingFlags.Instance | BindingFlags.Static);

foreach(PropertyInfo info in infos)
{
  column = table.Columns.Add();
  column.ColumnName = info.Name;
  column.DataType = typeof(string);
  column.ReadOnly = true;
}
```

The new code retrieves the array of PropertyInfo objects for a PropertyInfo class itself. The BindingFlags indicate that we should get both public and non-public as well as instance and static members of the type. The *foreach* statement works under the covers by requesting an Enumerator from a type. The implication is that the type must implement the IEnumerable interface. System.Array types—anything declared with the [ ] array operator—implicitly

implement the IEnumerable interface. For each PropertyInfo, a DataColumn is requested from the DataTable; the ColumnName, DataType, and ReadOnly properties are set. (The ReadOnly property is set because we can't really change the implementation of a type by modifying our database; hence, it makes no sense to allow users to modify PropertyInfo values.)

When we are all finished, we have a table whose schema mirrors the properties of a PropertyInfo class. A similar approach can be used to add rows of data to the DataTable. Refer to Listings 5-6 and 5-10 for examples of adding rows of data to a DataTable. You can also refer to the AssemblyViewer.sln for several examples of adding data to a DataTable.

Using the DataView

A DataView is by default a read-only view of data in a DataTable. You can enable modifying a DataView by changing the AllowNew, AllowEdit, and AllowDelete properties of the DataView to True. A DataView is initialized with a DataTable and can be used to sort, filter, and modify data.

We can create a DataView that shows read-only properties by filtering the CanWrite field of our Property table. Listing 5-13 demonstrates how to create a DataView and apply a row filter using syntax roughly equivalent to a predicate in a *Where* clause.

Listing 5-13 *This code demonstrates how to create a DataView and apply a row filter.*

```
OleDbConnection connection = new OleDbConnection(connectionString);
OleDbDataAdapter adapter =
  new OleDbDataAdapter("SELECT * FROM PROPERTY", connection);
DataTable table = new DataTable();
adapter.Fill(table);
DataView dataView = new DataView(table);
dataView.RowFilter = "CanWrite = 'False'";

dataGrid1.DataSource = dataView;
```

The big difference between Listing 5-13 and earlier listings is the creation of the DataView. As mentioned, we initialize a DataView with a DataTable. Optionally, we can filter and sort the DataView without requerying the data source. The example demonstrates a DataView.RowFilter that filters rows in the PROPERTY table by the string value of 'False' for CanWrite.

The last step demonstrates how to bind the DataView to a Windows Forms DataGrid. Refer to the AssemblyViewer.sln for a complete code listing. Check out the section "Binding a DataSet to a DataGrid" later in this chapter for more on using ADO.NET with controls.

Using the DataReader for Read-Only Data

A DataReader can be used to obtain a forward-only and read-only view of data without the overhead of a DataSet. A DataReader requires a connection and a command and is initialized by invoking a command object's ExecuteReader method. Listing 5-14 provides a brief example of using an OleDbDataReader.

Listing 5-14 *Using an OleDbDataReader to read all of the rows returned by an SQL command*

```
OleDbConnection connection = new OleDbConnection(connectionString);
OleDbCommand command =
  new OleDbCommand("SELECT * FROM FIELD", connection);
connection.Open();
OleDbDataReader reader = command.ExecuteReader();
while( reader.Read())
{
  Debug.WriteLine(reader.GetValue(0));
}
reader.Close();
connection.Close();
```

We can't bind the DataReader to the grid because a DataReader does not implement the IList or IListSource interfaces. That is the technical reason. The logical reason is that a grid can be scrolled forward and backward, and a DataReader is forward-only.

You must maintain an open connection while a DataReader is active, as demonstrated by the code. Ensure that you call Close on the reader when you have finished with it, because the connection cannot be reused until the reader has been closed. The benefit of using a DataReader is that it performs very fast read operations.

Displaying Information in the DataGrid

Object-oriented frameworks tend to be very flexible because there is a certain amount of commonality throughout the framework. One example is the Windows Forms DataGrid control. The DataGrid control has a DataSource property. You can assign a variety of objects

to a DataSource because the DataSource is implemented to work with any object that implements the IList or IListSource interfaces. (IListSource has two members, one of which returns an object that implements IList.) As a result, there is a tremendous flexibility when it comes to binding to the DataSource property.

In general, the DataSource property can be assigned to a DataTable, DataSet, DataView, DataViewManager, and any object that implements IList or IListSource. The net effect of the generic implementation of the DataSource is that you can obviously bind to objects defined in ADO.NET, but you can also bind to a list of any kind of object.

Recall that in Listing 5-12 we invoked the GetProperties method. GetProperties returned an array of PropertyInfo objects, represented in code as PropertyInfo[ ]. An array's underlying type is System.Array. Interestingly enough, System.Array implements the IList interface. The implication, then, is that we should be able to bind the DataGrid.DataSource property directly to the PropertyInfo[ ] returned by GetProperties. In fact, if you write the following code, you will get almost an identical result to the one you would get if you assigned the DataTable in Listing 5-12 to a DataGrid's DataSource property.

```
dataGrid1.DataSource = this.GetType().GetProperties();
```

The preceding single statement produces the view shown in the DataGrid in Figure 5-4. Try binding to the arrays returned by invoking GetMembers, GetFields, and GetMethods.

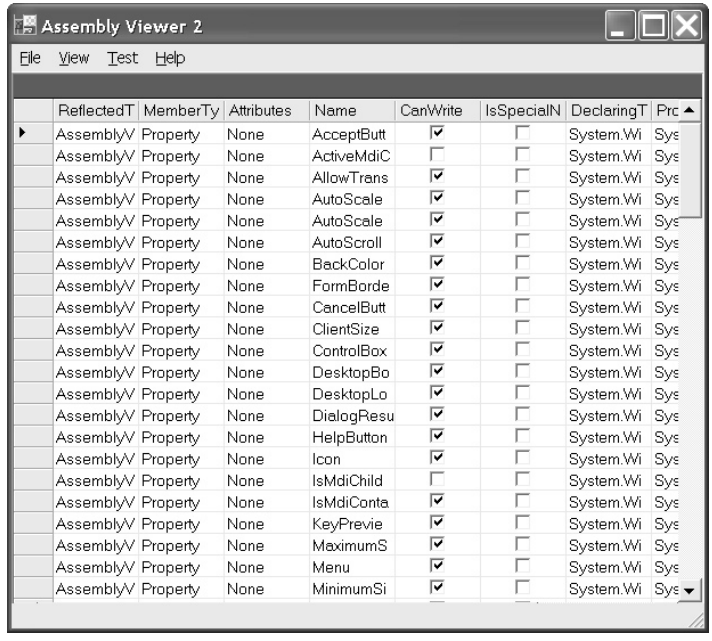


Figure 5-4 *A DataGrid that has been bound directly to an array of PropertyInfo objects*

Using the Command Object

Command objects are used to encapsulate SQL commands in an object. Use the SqlCommand for MS SQL Server providers and the OleDbCommand for OLE DB providers. Command objects store the SQL text that represents SELECT, UPDATE, INSERT, and DELETE statements, as well as provide you with an object for other SQL commands for particular providers. Examples of other commands include CREATE TABLE, ALTER TABLE, and DROP TABLE.

To create an OleDbCommand object, we can construct an instance of the object by passing the SQL text and an instance of an OleDbConnection object. The OleDbCommand object can be used as an instruction to an OleDbDataAdapter or directly to execute SQL commands that don't return a result set. Listing 15-4 provides an example of creating an OleDbCommand used to request a DataReader. You can look at that example and OleDbCommand objects created in the AssemblyViewer.sln for more examples.

One kind of command that we haven't looked at yet are commands used to create entities in a database. We can use an OleDbCommand and an SQL CREATE TABLE statement to add a new table to a database. The code in Listing 5-15 demonstrates how to create a table in an Access database. (You will need to vary the SQL slightly to match the syntax of the provider that you are using; the provider's user's guide is the best resource for specific SQL syntax.)

Listing 5-15 *This code demonstrates how to create a table in an OLE DB provider.*

```
const string connectionString =
  @"Provider=Microsoft.Jet.OLEDB.4.0;Password=;" +
  @"User ID=;Data Source=C:Temp\Reference.mdb;";
OleDbConnection connection = new OleDbConnection();
const string SQL =
  @"CREATE TABLE Constructor ([ID] COUNTER (1,1), " +
  @"[MemberType] string,[MethodHandle] string, " +
  @"[Attributes] string,[IsPublic] string, " +
  @"[IsPrivate] string,[IsFamily] string, " +
  @"[IsAssembly] string,[IsFamilyAndAssembly] string, " +
  @"[IsFamilyOrAssembly] string,[IsStatic] string, " +
  @[IsFinal] string,[IsVirtual] string,[IsHideBySig] string, " +
  @"[IsAbstract] string,[IsSpecialName] string, " +
  @"[IsConstructor] string,[CallingConvention] string, " +
  @"[Name] string,[DeclaringType] string, " +
  @"CONSTRAINT [Index1] PRIMARY KEY ([ID]))";
OleDbCommand command = new OleDbCommand(SQL, connection);
command.ExecuteNonQuery();
```

Most of the code in Listing 5-15 is the SQL text. (We'll see a better way to manage SQL statements in a moment.) The first statement is a literal connection string for the Jet OLE DB provider, including the Provider, Password, User ID, and Data Source clauses. The second statement creates an instance of an OleDbConnection. The third statement is a literal CREATE TABLE SQL statement that is appropriate for a Jet OLE DB provider. (Jet is the name of MS Access' database engine.) The OleDbCommand object is created by initializing an instance with the SQL text and the connection object. Finally, we send the query to the database engine with the command object.

The CREATE TABLE statement is specific to the database we are sending the SQL text to. The basic syntax is CREATE TABLE (*tablename fielddefinitions [fielddefinition1, ... fielddefinition(n)] constraints),* where the table name is a valid name, and the field definitions indicate the field name and data type and any constraints such as a primary key.

The CREATE TABLE statement demonstrates how to create an auto-increment column. The ID column described as [ID] COUNTER (1,1) defines an auto-increment column that uses a seed value of 1 and an increment value of 1. The CONSTRAINT [Index1] PRIMARY KEY ([ID]) defines an index named Index1 as the primary key based on the ID column.

Generating SQL with the CommandBuilder

Writing SQL statements can be a little tedious. There is an easier way to generate SQL with ADO.NET. Construct an instance of an OleDbCommandBuilder and initialize the command builder with an adapter that was initialized with a SELECT statement. The schema returned by the SELECT statement can be used to generate SQL INSERT, DELETE, and UPDATE statements.

Lines 18 and 19 of Listing 5-6 constructs an OleDbCommandBuilder with an OleDbDataAdapter instance. The command text used to initialize the OleDbDataAdapter was "SELECT * FROM METHOD". From the schema that can be read from the METHOD table, verbose instances of SQL commands can be generated. When you invoke the OleDbDataAdapter.Update command, the SQL created by the OleDbCommandBuilder is necessary to update the data source based on changes made to the tables in the DataSet.

Secondary Topics

You know how to create connections and use adapters to bridge data read via a connection into a DataTable or DataSet. These objects are seldom used in isolation. In your average application, you will be using ADO.NET objects in conjunction with visual controls, applications, and perhaps Web Services.

This part of the chapter introduces contextual ways in which you will use ADO.NET, including returning a DataSet from a Web Service. You can use the examples in this part of the chapter as a brief introduction to these subjects and explore the rest of the book for additional examples. Explore the sample applications that ship with Visual Studio .NET and the **www.gotdotnet.com** and **www.codeguru.com** websites as additional resources for more code examples.

Binding a DataSet to a DataGrid

You can bind data directly to a System.Web.UI.WebControls.DataGrid. The requirement is that the data must implement the IEnumerable interface. Data can be bound to a System.Windows.Forms.DataGrid control if the data object implements the IList or IListSource interfaces. The DataSet class implements IEnumerable, IList, and IListSource, which means that you can quickly create a user interface to bind a DataSet to DataGrids in Windows Forms or Web Forms.

These two fragments demonstrate how to bind a DataSet to a Windows Forms and Web Forms DataGrid.

```
DataGrid.DataSource = DataSet;
```

The preceding statement is all you need to bind a DataSet to a Windows Forms DataGrid.

```
DataGrid.DataSource = DataSet;
DataGrid.DataBind();
```

The preceding are the basic two statements necessary to bind a DataSet to a Web Forms DataGrid.

The DataGrid represents a specific instance of a DataGrid, and the DataSet represents a specific instance of a DataSet object. Additionally, you can bind a DataTable or any other collection that implements one of the necessary interfaces. For example, System.Array—which is the underlying type when you declare an array of any type—implements the IList interface. Hence, as demonstrated earlier in "Displaying Information in the DataGrid," any array of types can be displayed in a DataGrid. The public properties of the type of the object in the array will be used to create columns in the DataGrid.

Returning a DataSet from a Web Service

XML is an integral part of the .NET Framework. ADO.NET employs XML to manage data. The DataSet was defined with the SerializableAttribute. It is the SerializableAttribute and XML that support returning a DataSet from a Web Service. Listing 5-16 contains a monolithic Web Method (defined in the Reference.sln available for download from **www.osborne.com**) that dynamically explores the methods of the type passed to the Web Method.

Listing 5-16 *This code demonstrates a Web Method that returns a DataSet.*

```
[WebMethod, ReflectionPermission(SecurityAction.Demand)]
public DataSet GetMethods(string type)
{
  DataTable table = new DataTable("METHOD");
  MethodInfo[] methods = Type.GetType(type).GetMethods();
```

```
  DataColumn column;
  foreach(PropertyInfo propertyInfo
    in typeof(MethodInfo).GetProperties())
  {
    column = table.Columns.Add();
    column.ColumnName = propertyInfo.Name;
    column.DataType = typeof(string);
  }

  DataRow dataRow;
  foreach(MethodInfo methodInfo in methods)
  {
    dataRow = table.NewRow();
    foreach(PropertyInfo propertyInfo
      in methodInfo.GetType().GetProperties())
    {
      dataRow[propertyInfo.Name] =
        propertyInfo.GetValue(methodInfo, null);
    }

    table.Rows.Add(dataRow);
  }
  DataSet dataSet = new DataSet();
  dataSet.Tables.Add(table);
  return dataSet;
}
```

Pass in the full namespace of a type, and the GetMethods Web Method will create a table based on the properties of the MethodInfo type. The properties become the columns of the DataTable. Each method's properties become the rows for the DataTable. Add the DataTable to a DataSet, and we can return the dynamically reflected type's methods.

It is necessary to Demand ReflectionPermission if we are going to use Reflection from a Web Service. Any code that is downloaded from a network is not guaranteed to be granted Reflection permission.

Implementing a TraceListener

The AssemblyViewer.sln defines a nested class that inherits from TraceListener. By inheriting from TraceListener, we can register an instance of our custom TraceListener

listener with the System.Diagnostics.Trace class's Listeners collection. The result is that our custom listener can catch Trace messages and display them as part of our presentation layer. (This is consistent with the Trace window provided with Terrarium. Refer to Chapter 4 for information on Terrarium.)

Listing 15-17 demonstrates a private nested TraceListener class added to the main form of the AssemblyViewer.sln, and we can use the main form as a TraceListener window to display Trace information while we are testing our application.

Listing 15-17 *Implementation of a custom TraceListener*

```
private class Listener : TraceListener
{
  private StatusBar statusBar;
  public Listener(StatusBar statusBar) : base()
  {
    this.statusBar = statusBar;
  }

  public override void Write(string text)
  {
    statusBar.Text = text;
    Application.DoEvents();
  }

  public override void WriteLine(string text)
  {
    statusBar.Text = text;
    Application.DoEvents();
  }
}
```

To implement a custom TraceListener, you need to override the Write and WriteLine methods. When we add an instance of our TraceListener, Listener, to the System.Diagnostics .Trace.Listeners collection, messages written with the Trace object are sent to our listener too. Based on the implementation of our listener, we store a reference to a StatusBar and display trace information on the StatusBar control. In effect, the owning form's StatusBar control becomes a visual Trace Window.

Rather than inventing new classes, we can find new ways to use existing classes. The .NET Framework is well organized but extensive. Before you write a significant amount of new code, leverage as much of the existing framework as you can.

Summary

Consistent with the general approach of this book, this chapter demonstrated several topics that you are likely to find in a single application. A large number of applications are database applications. ADO.NET provides advancements in database programming that are intended to promote scalability. You can use an older version of ADO with COM Interop—but once you get used to ADO.NET you will not want to go back in time.

This chapter provided an AssemblyViewer.sln that builds on your knowledge of Reflection to demonstrate how to create dynamic databases. Reflected properties were a convenient mechanism for accomplishing this.

After completing this chapter, you should have a good understanding of ADO.NET and an improved understanding of Reflection; you should also know how to bind data to the DataGrid control for Windows Forms and Web Forms and how to implement a custom TraceListener. This chapter also further introduced Web Services. You will find these skills beneficial in simple Windows applications, as well as in complex enterprise solutions.