TRAINING & REFERENCE

murach's ADO.NET 2.0 database programming with VB 2005

(Chapter 2)

MIKE MURACH & ASSOCIATES, INC.
1-800-221-5528 • (559) 440-9071 • Fax: (559) 440-0963
murachbooks@murach.com • www.murach.com

Contents

2

An introduction to ADO.NET 2.0

ADO.NET consists of a set of classes defined by the .NET Framework that you can use to access the data in a database. The current version of ADO.NET is ADO.NET 2.0, and that's the version you'll learn about in this book.

This chapter introduces you the primary ADO.NET classes that you'll use as you develop database applications with Visual Basic. This chapter also introduces you to the two basic ways that you can develop database applications with ADO.NET.

If you've used ADO.NET 1.0, you'll see that the classes that are included in ADO.NET 2.0 have changed very little. But you'll also see that the features that Visual Studio provides for developing database applications with ADO.NET 2.0 have changed a lot.

An overview of ADO.NET

ADO.NET (*ActiveX Data Objects .NET*) is the primary data access API for the .NET Framework. It provides the classes that you use as you develop database applications with Visual Basic as well as the other .NET languages. In the topics that follow, you'll learn the two basic ways that you can use the ADO.NET classes for accessing and updating the data in a database.

How to use ADO.NET with datasets

One way to develop database applications with ADO.NET is to use datasets. With this approach, your application gets data from a database and stores it in a *dataset* that is kept in cache memory on disk. Then, your application can add, update, or delete rows in the dataset, and it can later save those changes from the dataset to the database.

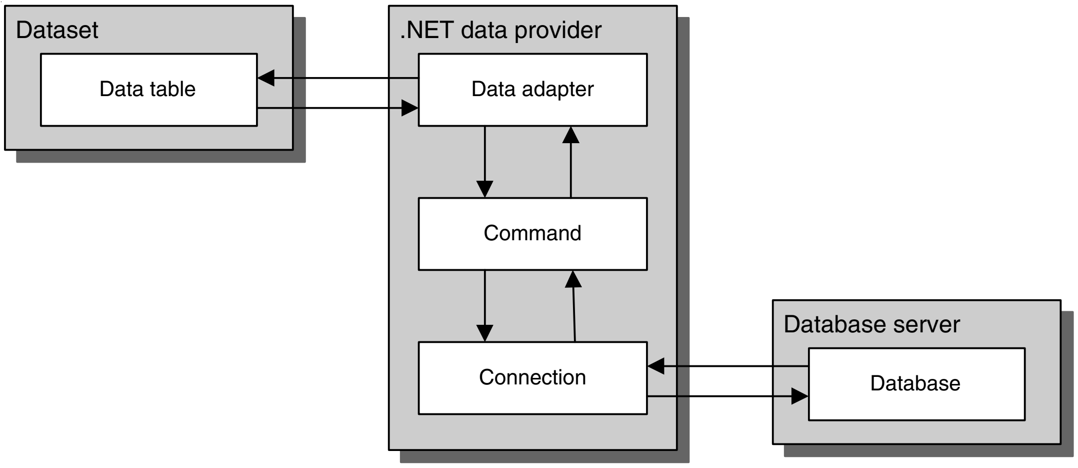
When you use this approach, your application uses the ADO.NET objects shown in figure 2-1. To load data into a *data table* within a dataset, you use a *data adapter*. Its main function is to manage the flow of data between a dataset and a database. To do that, it uses *commands* that define the SQL statements to be issued. The command for retrieving data, for example, typically defines a Select statement. Then, the command connects to the database using a *connection* and passes the Select statement to the database. After the Select statement is executed, the result set it produces is sent back to the data adapter, which stores the results in the data table.

To update the data in a database, the data adapter determines which rows in the data table have been inserted, updated, or deleted. Then, it uses commands that define Insert, Update, and Delete statements for the data table to update the associated rows in the database. Like the command that retrieves data from the database, the commands that update the database use a connection to connect to the database and perform the requested operation.

Note that the data in a dataset is independent of the database that the data was retrieved from. In fact, the connection to the database is typically closed after the data is retrieved from the database. Then, the connection is opened again when it's needed. Because of that, the application must work with the copy of the data that's stored in the dataset. The architecture that's used to implement this type of data processing is referred to as a *disconnected data architecture*.

Although this approach is more complicated than a connected architecture, it has several advantages. One advantage is that using a disconnected data architecture can improve system performance due to the use of fewer system resources for maintaining connections. Another advantage is that it makes ADO.NET compatible with ASP.NET web applications, which are inherently disconnected.

Basic ADO.NET objects



Description

- A *.NET data provider* provides the classes that let you create the objects that you use to retrieve data from a database and to store data in a database.
- One way to work with a database when you use ADO.NET is to retrieve data from a database into a dataset and to store data from the dataset to the database.
- A *dataset* contains one or more *data tables* that store the data retrieved from the data-base. Then, the application can work with the data in the data tables, and it can insert, update, and delete rows in the data tables.
- To retrieve data from a database and store it in a data table, a *data adapter* object issues a Select statement that's stored in a *command* object. Next, the command object uses a *connection* object to connect to the database and retrieve the data. Then, the data is passed back to the data adapter, which stores the data in the dataset.
- To update the data in a database based on the data in a data table, the data adapter object issues an Insert, Update, or Delete statement that's stored in a command object. Then, the command object uses a connection to connect to the database and update the data.
- When you use a data adapter to work with the data in a database, the data provider remains connected to the database only long enough to retrieve or update the specified data. Then, it disconnects from the database and the application works with the data via the dataset object. This is referred to as a *disconnected data architecture.*
- The disconnected data architecture offers improved system performance due to the use of fewer system resources for maintaining connections.

Figure 2-1 How to use ADO.NET with datasets

Two ways to create the ADO.NET objects for working with datasets

When you use datasets in your database applications, there are two basic techniques you can use to create the ADO.NET objects that you need. Both are illustrated in figure 2-2.

First, you can create the ADO.NET objects from a *data source* that's shown in the *Data Sources window*. Data sources are a new feature of .NET 2.0 that makes it easy to create forms that work with the data in a data source such as a database. In this example, the data source corresponds with the data in the Terms table in the Payables database.

In the next chapter, you'll learn how to create a data source. For now, you should know that once you create a data source, you can drag it onto a form to automatically add controls to the form and to create the ADO.NET objects for working with the data in the data source.

In this figure, for example, you can see the controls and objects that are generated when you drag the Terms table onto the form. Here, a DataGridView control has been added to the form to display the terms in the Terms table, and a toolbar has been added that lets you work with this data.

In addition, four objects have been added to the *Component Designer tray* below the form. Two of these are ADO.NET objects. The first one, named PayablesDataSet, defines the dataset for the form. Then, an object named TermsTableAdapter defines the table adapter for the Terms table. Although a *table adapter* is similar to a data adapter, it provides improved functionality, including a built-in connection object and the ability to contain multiple queries. You'll learn more about table adapters in the next chapter.

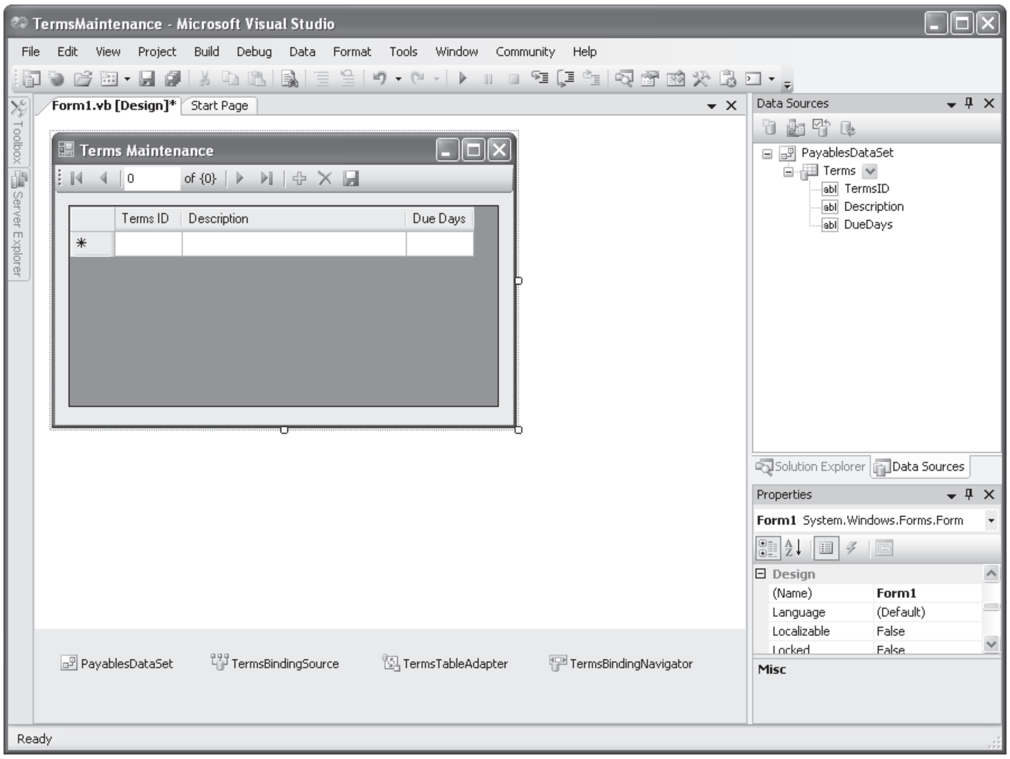
The other two objects in the Component Designer tray are used to bind the controls on the form to the data source. The first object, named TermsBindingSource, identifies the Terms table as the data source for the controls. The second object, named TermsBindingNavigator, defines the toolbar that's displayed across the top of the form.

Although you don't usually need to change the properties of the objects in the Component Designer tray, you should know that you can do that using the same technique you use to change the properties of a control on a form. That is, you just select an object to display its properties in the Properties window and then work with them from there.

The second technique for creating the ADO.NET objects that you need for using datasets is to write the code yourself. In this figure, for example, you can see the code that creates four objects: a connection, a command named selectCommand that contains a Select statement, a data adapter named termsDataAdapter, and a dataset named termsDataSet.

Although creating ADO.NET objects through code is more time-consuming than using data sources, it can help you do tasks that are difficult or impossible to do with data sources. It can result in more compact and efficient code. And it lets you encapsulate an application's database processing in database classes. You'll learn more about this in section 4 of this book.

Using the Data Sources window to create ADO.NET objects


TermsMaintenance - Microsoft Visual Studio
File Edit View Project Build Debug Data Format Tools Window Community Help
Form1.vb [Design]* Start Page
Terms Maintenance
0 of {0}
Terms ID Description Due Days
Data Sources
PayablesDataSet
Terms
TermsID
Description
DueDays
Solution Explorer Data Sources
Properties
Form1 System.Windows.Forms.Form
Design
(Name) Form1
Language (Default)
Localizable False
Locked False
Misc
PayablesDataSet TermsBindingSource TermsTableAdapter TermsBindingNavigator
Ready

Visual Basic code that creates ADO.NET objects

```
Dim connectionString As String _
    = "Data Source=localhost\SqlExpress;Initial Catalog=Payables;" _
    & "Integrated Security=True"
Dim connection As New SqlConnection(connectionString)

Dim selectStatement As String = "SELECT * FROM Terms"
Dim selectCommand As New SqlCommand(selectStatement, connection)

Dim termsDataAdapter As New SqlDataAdapter(selectCommand)

Dim termsDataSet As New DataSet
```

Description

- You can use the *Data Sources window* in Visual Studio to create a *data source*. Then, you can drag the data source onto the form to automatically generate a table adapter object and a dataset object.
- A *table adapter* is like a data adapter, but it has a built-in connection object, and it can contain more than one query. You'll learn how to work with table adapters in chapter 3.
- To create ADO.NET objects in code, you write declarations that identify the class each object is created from. You'll learn how to write code like this in section 4.

Figure 2-2 Two ways to create the ADO.NET objects for working with datasets

How to use ADO.NET without using datasets

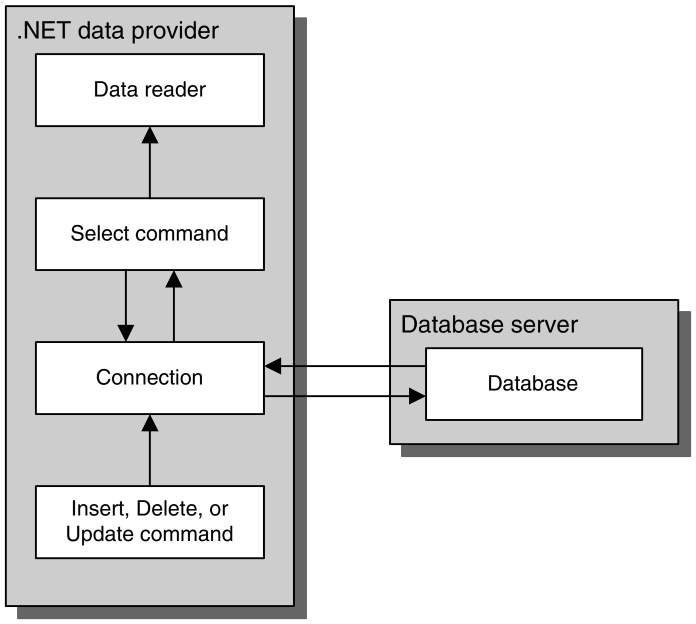
The second way to develop database applications is to work with the database directly, without using datasets. This approach is illustrated in figure 2-3. As you can see, you still use command and connection objects to access the database. But instead of using a data adapter to execute the commands, you execute the commands directly.

When you work this way, you have to provide the code that handles the result of each command. If you issue a command that contains an Insert, Update, or Delete statement, for example, the result is an integer that indicates the number of rows that were affected by the operation. You can use that information to determine if the operation was successful.

The code example in this figure illustrates how this works. Here, a command that inserts a row into the Terms table is created. In this case, the Insert statement uses parameters to identify the column values that must be supplied. Then, values are assigned to these parameters before the command is executed. Finally, the connection is opened, the command is executed, and the connection is closed.

If you execute a command that contains a Select statement, the result is a result set that contains the rows you requested. To read through the rows in the result set, you use a *data reader* object. Although a data reader provides an efficient way of reading the rows in a result set, you can't use it to modify those rows. In addition, it only lets you read rows in a forward direction, so once you read the next row, the previous row is unavailable. Because of that, you typically use a data reader either to retrieve rows that are displayed in a control such as a combo box, or to retrieve and work with a single database row at a time.

ADO.NET components for accessing a database directly



Code that creates and executes a command that inserts a row

```
Dim insertStatement As String _
    = "INSERT Terms (Description, DueDays) " _
    & "VALUES (@Description, @DueDays)"
Dim insertCommand As New SqlCommand(insertStatement, connection)
insertCommand.Parameters.AddWithValue("@Description", terms.Description)
insertCommand.Parameters.AddWithValue("@UnitPrice", terms.DueDays)
connection.Open()
Dim count As Integer = insertCommand.ExecuteNonQuery
connection.Close()
```

Description

- Instead of executing the commands associated with a data adapter to manage the flow of data between a dataset and a database, you can execute those commands directly. When you do that, you create and work with the ADO.NET objects through code.
- To retrieve data from a database, you execute a command object that contains a Select statement. Then, the command object uses a connection to connect to the database and retrieve the data. You can then read the results one row at a time using a *data reader* object.
- To insert, update, or delete data in a database, you execute a command object that contains an Insert, Update, or Delete statement. Then, the command object uses a connection to connect to the database and update the data. You can then check the value that's returned to determine if the operation was successful.

Figure 2-3 How to use ADO.NET without using datasets

Concurrency and the disconnected data architecture

Although the disconnected data architecture has advantages, it also has some disadvantages. One of those is the conflict that can occur when two or more users retrieve and then try to update data in the same row of a table. This is called a *concurrency* problem. This is possible because once a program retrieves data from a database, the connection to that database is dropped. As a result, the database management system can't manage the update process.

To illustrate, consider the situation shown in figure 2-4. Here, two users have retrieved the Vendors table from a database, so a copy of the Vendors table is stored on each user's PC. These users could be using the same program or two different programs. Now, suppose that user 1 modifies the address in the row for vendor 123 and updates the Vendors table in the database. And suppose that user 2 modifies the phone number in the row for vendor 123 and then tries to update the Vendors table in the database. What will happen? That will depend on the *concurrency control* that's used by the programs.

When you use ADO.NET, you have two choices for concurrency control. First, you can use *optimistic concurrency*, which checks whether a row has been changed since it was retrieved. If it has, the update or deletion will be refused and a *concurrency exception* will be thrown. Then, the program should handle the error. For example, it could display an error message that tells the user that the row could not be updated and then retrieve the updated row so the user can make the change again.

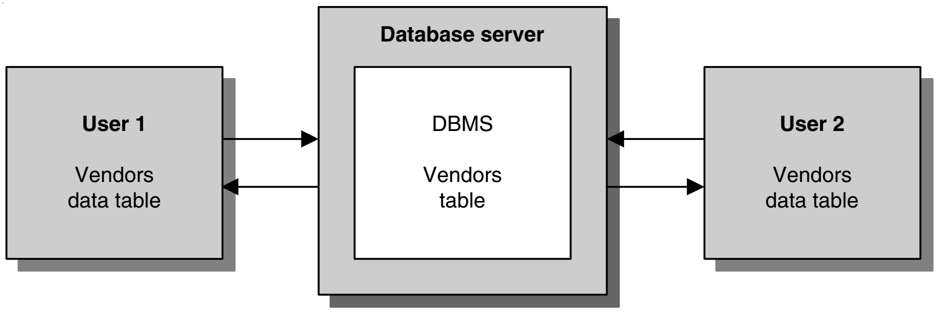
In contrast, the *"last in wins"* technique works the way its name implies. Since no checking is done with this technique, the row that's updated by the last user overwrites any changes made to the row by a previous user. For the example above, the row updated by user 2 will overwrite changes made by user 1, which means that the phone number will be right but the address will be wrong. Since errors like this corrupt the data in a database, optimistic concurrency is used by most programs, which means that your programs have to handle the concurrency exceptions that are thrown.

If you know that concurrency will be a problem, you can use a couple of programming techniques to limit concurrency exceptions. If a program uses a dataset, one technique is to update the database frequently so other programs can retrieve the current data. The program should also refresh its dataset frequently so it contains the recent changes made by other programs.

Another way to avoid concurrency exceptions is to retrieve and work with just one row at a time. That way, it's less likely that two programs will update the same row at the same time. In contrast, if two programs retrieve the same table, they will of course retrieve the same rows. Then, if they both update the same row in the table, even though it may not be at the same time, a concurrency exception will occur when they try to update the database.

Of course, you will understand and appreciate this more as you learn how to develop your own database applications. As you develop them, though, keep in mind that most applications are multi-user applications. That's why you have to be aware of concurrency problems.

Two users who are working with copies of the same data



What happens when two users try to update the same row

- When two or more users retrieve the data in the same row of a database table at the same time, it is called *concurrency*. Because ADO.NET uses a disconnected data architecture, the database management system can't prevent this from happening.
- If two users try to update the same row in a database table at the same time, the second user's changes could overwrite the changes made by the first user. Whether or not that happens, though, depends on the *concurrency control* that the programs use.
- By default, ADO.NET uses *optimistic concurrency*. This means that the program checks to see whether the database row that's going to be updated or deleted has been changed since it was retrieved. If it has, a *concurrency exception* occurs and the update or deletion is refused. Then, the program should handle the exception.
- If optimistic concurrency isn't in effect, the program doesn't check to see whether a row has been changed before an update or deletion takes place. Instead, the operation proceeds without throwing an exception. This is referred to as "*last in wins*" because the last update overwrites any previous update. And this leads to errors in the database.

How to avoid concurrency errors

- For many applications, concurrency errors rarely occur. As a result, optimistic concurrency is adequate because the users will rarely have to resubmit an update or deletion that is refused.
- If concurrency is likely to be a problem, a program that uses a dataset can be designed so it updates the database and refreshes the dataset frequently. That way, concurrency errors are less likely to occur.
- Another way to avoid concurrency errors is to design a program so it retrieves and updates just one row at a time. That way, there's less chance that two users will retrieve and update the same row at the same time.

Figure 2-4 Concurrency and the disconnected data architecture

The ADO.NET data providers and their classes

The *.NET data providers* provide the ADO.NET classes that you use for connecting to and working directly with a database. That's why these classes are sometimes called the *connected classes*. In the topics that follow, you'll learn more about the data providers and the classes that they provide.

The .NET data providers

All .NET data providers include the core classes for creating the four types of objects listed in the first table in figure 2-5. You've already learned the basic functions of these classes, and you'll learn more about these classes throughout this book.

The second table in this figure lists the four data providers that come with the .NET Framework. The SQL Server data provider is designed to provide efficient access to a Microsoft SQL Server database. The OLE DB data provider is a generic data provider that can access any database that supports the industry standard OLE DB interface. The ODBC provider lets you access any database that can work with ODBC, another industry standard database interface. And the Oracle provider lets you access data that's stored in Oracle databases. Although you can use the OLE DB data provider to access a SQL Server database, you shouldn't do that unless you plan on migrating the data to another database since the SQL Server data provider is optimized for accessing SQL Server data.

Besides the providers that come with the .NET Framework, several database vendors have developed .NET data providers that are optimized for use with their databases. For example, .NET data providers are available for the popular MySQL and Sybase databases as well as for a variety of other databases. Before you develop an application using the OLE DB provider, then, you should check with your database vendor to see if a specialized .NET data provider is available.

The third table in this figure lists the names of the classes you use to create objects using the SQL Server, OLE DB, ODBC, and Oracle providers. Notice that these classes use prefixes ("Sql," "OleDb," "Odbc," and "Oracle") to indicate which provider each class belongs to.

When you develop a Visual Basic application that uses ADO.NET, you'll want to add an Imports statement for the namespace that contains the data provider classes at the beginning of each source file that uses those classes. That way, you won't have to qualify the references to these classes. These namespaces are listed in the second table in this figure.

Now that you're familiar with the core classes of the four .NET data providers that come with the .NET Framework, the next four topics describe the classes of the SQL Server data provider in more detail. You should realize, though, that the information presented in these topics applies to the classes of the other data providers as well.

.NET data provider core objects

Object	Description
Connection	Establishes a connection to a database.
Command	Represents an individual SQL statement or stored procedure that can be executed against the database.
Data reader	Provides read-only, forward-only access to the data in a database.
Data adapter	Provides the link between the command and connection objects and a dataset object.

Data providers included with the .NET framework

Provider	Namespace	Description
SQL Server	System.Data.SqlClient	Lets you access SQL Server databases.
OLE DB	System.Data.OleDb	Lets you access any database that supports OLE DB.
ODBC	System.Data.Odbc	Lets you access any database that supports ODBC.
Oracle	System.Data.OracleClient	Lets you access Oracle databases.

Class names for the data providers

Object	SQL Server	OLE DB	ODBC	Oracle
Connection	SqlConnection	OleDbConnection	OdbcConnection	OracleConnection
Command	SqlCommand	OleDbCommand	OdbcCommand	OracleCommand
Data reader	SqlDataReader	OleDbDataReader	OdbcDataReader	OracleDataReader
Data adapter	SqlDataAdapter	OleDbDataAdapter	OdbcDataAdapter	OracleDataAdapter

An Imports statement for the SQL Server data provider namespace

```
Imports System.Data.SqlClient
```

Description

- The *.NET data providers* provide the ADO.NET classes that are responsible for working directly with a database. In addition to the core classes shown above, classes are provided for other functions such as passing parameters to commands and working with transactions.
- To use a .NET data provider in an application, you should add an Imports statement for the appropriate namespace at the beginning of the source file. Otherwise, you'll have to qualify each class you refer to with the SqlClient, OleDb, Odbc, or OracleClient namespace since these namespaces aren't included as references by default.
- Other .NET data providers are available to provide efficient access to non-Microsoft databases such as MySQL, Sybase, PostgreSQL, and IBM DB2.
- All of the ADO.NET objects are implemented by classes in the System.Data namespace of the .NET Framework. However, the specific classes used to implement the connection, command, data reader, and data adapter objects depend on the .NET data provider you use.

Figure 2-5 The .NET data providers

The SqlConnection class

Before you can access the data in a database, you have to create a connection object that defines the connection to the database. To do that, you use the SqlConnection class presented in figure 2-6.

The most important property of the SqlConnection class is ConnectionString. A *connection string* is a text string that provides the information necessary to establish a connection to a database. That means it includes information such as the name of the database you want to access and the database server that contains it. It can also contain authentication information such as a user ID and password.

The two methods of the SqlConnection class shown in this figure let you open and close the connection. In general, you should leave a connection open only while data is being retrieved or updated. That's why when you use a data adapter, the connection is opened and closed for you. In that case, you don't need to use the Open and Close methods.

The SqlCommand class

To execute a SQL statement against a SQL Server database, you create a SqlCommand object that contains the statement. Figure 2-6 presents the SqlCommand class you use to create this object. Notice that the Connection property of this class associates the command with a SqlConnection object, and the CommandText property contains the SQL statement to be executed.

The CommandType property indicates how the command object should interpret the value of the CommandText property. Instead of specifying a SQL statement for the CommandText property, for example, you can specify the name of a stored procedure. If you specify a SQL statement, you set the value of the CommandType property to CommandType.Text. If you specify the name of a stored procedure, you set it to CommandType.StoredProcedure.

Earlier in this chapter, you learned that you can use a data adapter to execute command objects. In addition, you can execute a command object directly using one of the three Execute methods shown in this figure. If the command contains a Select statement, for example, you can execute it using either ExecuteReader or ExecuteScalar. If you use ExecuteReader, the results are returned as a DataReader object. If you use ExecuteScalar, only the value in the first column and row of the query results is returned. You're most likely to use this method with a Select statement that returns a single summary value or the value of an identify column for a row that was just inserted into the database.

If the command contains an Insert, Update, or Delete statement, you'll use the ExecuteNonQuery method to execute it. This method returns an integer value that indicates the number of rows that were affected by the command. For example, if the command deletes a single row, the ExecuteNonQuery method returns 1.

Common properties and methods of the SqlConnection class

Property	Description
ConnectionString	Contains information that lets you connect to a SQL Server database. The connection string includes information such as the name of the server, the name of the database, and login information.
Method	**Description**
Open	Opens a connection to a database.
Close	Closes a connection to a database.

Common properties and methods of the SqlCommand class

Property	Description
Connection	The SqlConnection object that's used by the command to connect to the database.
CommandText	The text of the SQL command or the name of a stored procedure.
CommandType	A constant in the CommandType enumeration that indicates whether the CommandText property contains a SQL statement (Text) or the name of a stored procedure (StoredProcedure).
Parameters	The collection of parameters used by the command.
Method	**Description**
ExecuteReader	Executes a query and returns the result as a SqlDataReader object.
ExecuteNonQuery	Executes the command and returns an integer representing the number of rows affected.
ExecuteScalar	Executes a query and returns the first column of the first row returned by the query.

Description

- Each command object is associated with a connection object through the command's Connection property. When a command is executed, the information in the ConnectionString property of the connection object is used to connect to the database.
- When you use a data adapter to work with a database, the connection is opened and closed automatically. If that's not what you want, you can use the Open and Close methods of the connection object to open and close the connection.
- You can use the three Execute methods of a command object to execute the SQL statement it contains. You can also execute the SQL statement in a command object using methods of the data adapter. See figure 2-7 for more information.

Figure 2-6 The SqlConnection and SqlCommand classes

The SqlDataAdapter class

As you have learned, the job of a data adapter is to provide a link between a database and a dataset. The four properties of the SqlDataAdapter class listed in figure 2-7, for example, identify the four SQL commands that the data adapter uses to transfer data from the database to the dataset and vice versa. The SelectCommand property identifies the command object that's used to retrieve data from the database. And the DeleteCommand, InsertCommand, and UpdateCommand properties identify the commands that are used to update the database based on changes made to the data in the dataset.

To execute the command identified by the SelectCommand property and place the data that's retrieved in a dataset, you use the Fill method. Then, the application can work with the data in the dataset without affecting the data in the database. If the application makes changes to the data in the dataset, it can use the data adapter's Update method to execute the commands identified by the DeleteCommand, InsertCommand, and UpdateCommand properties and post the changes back to the database.

The SqlDataReader class

A data reader provides an efficient way to read the rows in a result set returned by a database query. In fact, when you use a data adapter to retrieve data, the data adapter uses a data reader to read through the rows in the result set and store them in a dataset.

A data reader is similar to other types of readers you may have encountered in the .NET Framework, such as a TextReader, a StreamReader, or an XmlReader. Like these other readers, a data reader lets you read rows but not modify them. In other words, a data reader is read-only. In addition, it only lets you read rows in a forward direction. Once you read the next row, the previous row is unavailable.

Figure 2-7 lists the most important properties and methods of the SqlDataReader class. You use the Read method to read the next row of data in the result set. In most cases, you'll code the Read method in a loop that reads and processes rows until the end of the data reader is reached.

To access a column of data from the current row of a data reader, you use the Item property. To identify the column, you can use either its index value like this:

```
drVendors.Item(0)
```

or its name like this:

```
drVendors.Item("Name")
```

Since Item is the default property, you can also omit it like this:

```
drVendors("Name")
```

Common properties and methods of the SqlDataAdapter class

Property	Description
SelectCommand	A SqlCommand object representing the Select statement or stored procedure used to query the database.
DeleteCommand	A SqlCommand object representing the Delete statement or stored procedure used to delete a row from the database.
InsertCommand	A SqlCommand object representing the Insert statement or stored procedure used to add a row to the database.
UpdateCommand	A SqlCommand object representing the Update statement or stored procedure used to update a row in the database.
Method	**Description**
Fill	Executes the command identified by the SelectCommand property and loads the result into a dataset object.
Update	Executes the commands identified by the DeleteCommand, InsertCommand, and UpdateCommand properties for each row in the dataset that was deleted, added, or updated.

Common properties and methods of the SqlDataReader class

Property	Description
Item	Accesses the column with the specified index or name from the current row.
FieldCount	The number of columns in the current row.
Method	**Description**
Read	Reads the next row. Returns True if there are more rows. Otherwise, returns False.
Close	Closes the data reader.

Description

- When the Fill method of a data adapter is used to retrieve data from a database, the data adapter uses a data reader to load the results into a dataset. If you don't use a dataset, you can work with a data reader directly.
- A data reader provides read-only, forward-only access to the data in a database. Because it doesn't require the overhead of a dataset, it's more efficient than using a data adapter. However, it can't be used to update data. To do that, you have to use other techniques.

Figure 2-7 The SqlDataAdapter and SqlDataReader classes

ADO.NET datasets

Unlike the .NET data providers that provide the connected classes for accessing the data in a database, an ADO.NET dataset provides the *discon-nected classes* for working with the data in a database. In the next two topics, you'll learn how a dataset is organized, and you'll get an overview of the classes you use to define dataset objects.

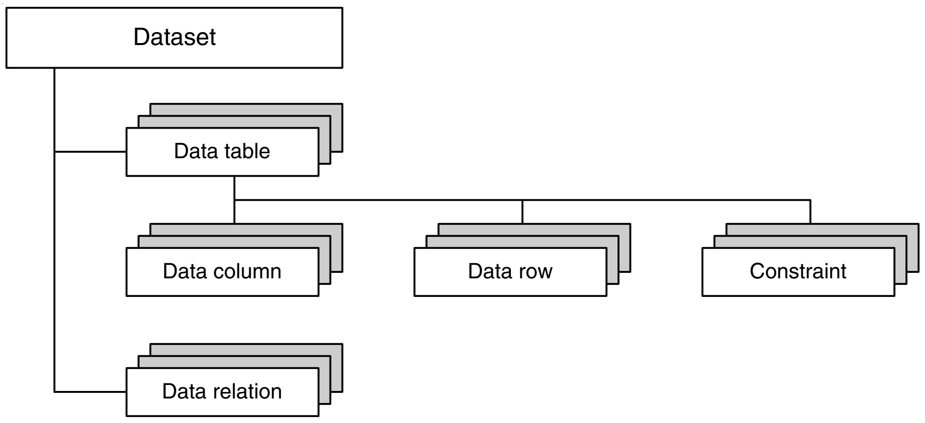
How a dataset is organized

Figure 2-8 illustrates the basic organization of an ADO.NET dataset. The first thing you should notice in this figure is that a dataset is structured much like a relational database. It can contain one or more tables, and each table can contain one or more columns and rows. In addition, each table can contain one or more constraints that can define a unique key within the table or a foreign key of another table in the dataset. If a dataset contains two or more tables, the dataset can also define the relationships between those tables.

Although a dataset is structured much like a relational database, it's important to realize that each table in a dataset corresponds to the result set that's returned from a Select statement, not necessarily to an actual table in a database. For example, a Select statement may join data from several tables in a database to produce a single result set. In this case, the table in the dataset would represent data from each of the tables involved in the join.

You should also know that each group of objects in the diagram in this figure is stored in a collection. All of the columns in a table, for example, are stored in a collection of columns, and all of the rows are stored in a collection of rows. You'll learn more about these collections in the next figure and in later chapters.

The basic dataset object hierarchy



Description

- A dataset object consists of a hierarchy of one or more data table and *data relation* objects.
- A data table object consists of one or more *data column* objects and one or more *data row* objects. The data column objects define the data in each column of the table, including its name, data type, and so on, and the data row objects contain the data for each row in the table.
- A data table can also contain one or more *constraint* objects that are used to maintain the integrity of the data in the table. A unique key constraint ensures that the values in a column, such as the primary key column, are unique. And a foreign key constraint determines how the rows in one table are affected when corresponding rows in a related table are updated or deleted.
- The data relation objects define how the tables in the dataset are related. They are used to manage constraints and to simplify the navigation between related tables.
- All of the objects in a dataset are stored in collections. For example, the data table objects are stored in a data table collection, and the data row objects are stored in a data row collection. You can refer to these collections through the properties of the containing objects.

Figure 2-8 How a dataset is organized

The dataset classes

Figure 2-9 presents some of the properties and methods of the four main classes that you use to work with a dataset: DataSet, DataTable, DataColumn, and DataRow. As you saw in the previous figure, the objects you create from these classes form a hierarchy where each dataset can contain one or more tables and each table can contain one or more rows and one or more columns.

Because of that, a dataset contains a Tables property that provides access to the collection of tables in the dataset. Similarly, a data table contains a Columns property and a Rows property that provide access to the collections of columns and rows in the table. These are the properties you're most likely to use as you work with these objects.

Although they're not shown in this figure, the collections you refer to through the Tables property of a dataset and the Columns and Rows properties of a data table have properties and methods of their own. For instance, each collection has a Count property that you can use to determine how many items are in the collection. To get the number of tables in a dataset named payablesDataSet, for example, you can use code like this:

```
payablesDataSet.Tables.Count()
```

To access a specific item in a collection, you use the Item property. On that property, you specify the index value or name of the item you want to access. To access the Vendors table in payablesDataSet, for example, you can use code like this:

```
payablesDataSet.Tables.Item("Vendors")
```

Since Item is the default property of the collection class, however, you typically omit it like this:

```
payablesDataSet.Tables("Vendors")
```

The code in this figure shows how you can use a For Each…Next statement to loop through the items in a collection. Here, the statement loops through the rows in the Vendors table. To do that, it uses a variable that's declared as a DataRow object. Then, the For Each…Next statement uses this variable to retrieve the value of the Name column in each row. You can use similar code to loop through the columns in a table or the tables in a dataset.

Common properties of the DataSet class

Property	Description
DataSetName	The name of the dataset.
Tables	A collection of the DataTable objects contained in the dataset.
Relations	A collection of the DataRelation objects contained in the dataset.

Common properties and methods of the DataTable class

Property	Description
TableName	The name of the table.
Columns	A collection of the DataColumn objects contained in the data table.
Rows	A collection of the DataRow objects contained in the data table.
Constraints	A collection of the Constraint objects contained in the data table.
Method	**Description**
NewRow	Creates a new row in the table.

Common properties of the DataColumn class

Property	Description
ColumnName	The name of the column.
AllowDBNull	Indicates whether the column allows null values.
AutoIncrement	Indicates whether the column is an auto-increment column, which is similar to an identity column in SQL Server.

Common properties and methods of the DataRow class

Property	Description
Item	Accesses the specified column of the row.
IsNull	Indicates whether the specified column contains a null value.
Method	**Description**
Delete	Deletes a row.

Code that refers to the rows collection in the tables collection of a dataset

```
Dim message As String
For Each dr As DataRow In vendorsDataSet.Tables("Vendors").Rows
    message &= dr.Item("Name") & vbCrLf
Next
MessageBox.Show(message)
```

Description

- You'll use the properties and methods of the dataset classes most often when you work with ADO.NET objects through code. You'll learn more about this in section 4.
- Each collection of objects has properties and methods that you can use to work with the collection.

Figure 2-9 The DataSet, DataTable, DataColumn, and DataRow classes

How ADO.NET applications are structured

As you saw earlier in this chapter, you can use two basic techniques to retrieve and work with the data in a database. With the first technique, you use a data adapter to retrieve the data and store it in a dataset and to update the database with changes made to the dataset. With the second technique, you work with the database by executing command objects directly, and you work with result sets using a data reader. In the next two topics, you'll see how the structures of applications that use these two techniques differ.

How an application that uses datasets is structured

When you develop an application that uses datasets, it typically consists of the two layers shown in figure 2-10. The first layer, called the *presentation layer*, consists of the form classes that display the user interface, plus the dataset classes used by the application. When you use a data source as shown in figure 2-2, for example, the application includes a dataset class that defines the tables and columns in the data source. Then, you can use the objects that are created when you drag the data source to a form for working with the data in the dataset that's created.

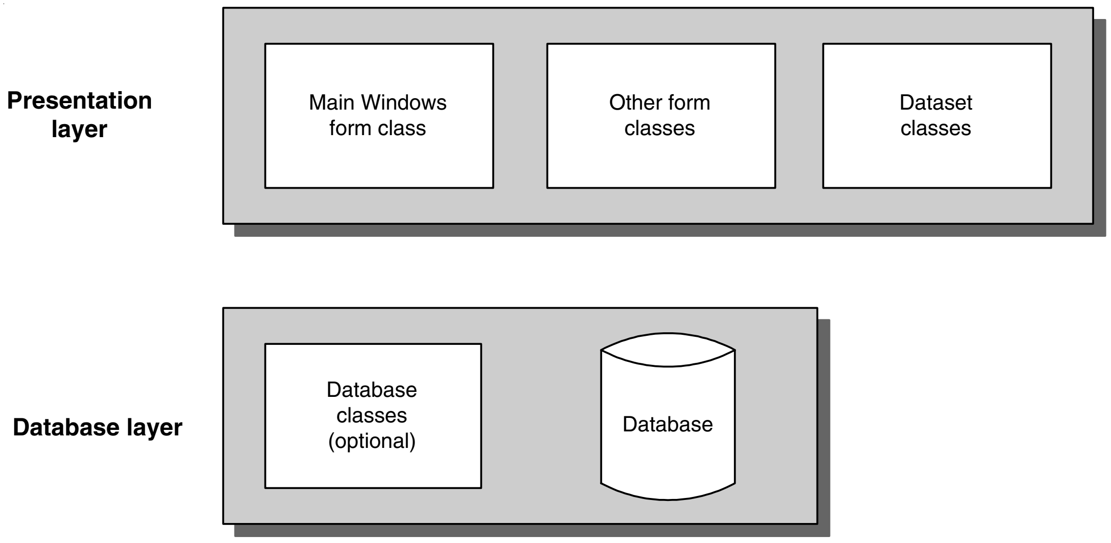
The second layer, called the *database layer*, always includes the database itself. In addition, this layer can include database classes that provide the data access required by the application. These classes typically include methods that connect to the database and retrieve, insert, add, and delete information from the database. Then, the presentation layer can call these methods to access the database, leaving the details of database access to the database classes.

Please note, however, that you can't use database classes when you develop an application by using a data source. That's because the table adapter object that you use to work with the database is generated for you as part of the presentation layer. As a result, all of the code in an application like this is typically stored in the presentation layer.

The primary benefit of using data sources and datasets is rapid application development. This is especially useful for developing small, relatively simple applications and for prototyping larger applications. As an application gets more complicated, though, so does the use of data sources and datasets. So at some point, it often makes sense to use a three-layer architecture as shown in the next figure.

In case you aren't familiar with the term *prototyping*, it refers to quickly developing a working model of an application that can be reviewed by the intended users. Then, the users can point out what's wrong with the prototype or what they want changed, and the prototype can be changed accordingly. When the users agree that the prototype does or will do everything that they want, the application can be completely rewritten, often with a three-layer architecture.

The architecture of an application that uses datasets



Description

- When you use a dataset to store the data that's retrieved from a database, the application is typically separated into two layers: the presentation layer and the database layer.
- The *presentation layer* consists of the forms that provide the application's user interface. In addition, this layer contains the dataset classes that define the data that the forms can use.
- The *database layer* consists of the database itself, and it can also include database classes that provide the methods for retrieving and updating the database for the application.
- If you create ADO.NET objects using a data source, you won't be able to use database classes in the database layer. If you create ADO.NET objects through code, you will be able to use database classes in the database layer, although that isn't a common practice.

Figure 2-10 How an application that uses datasets is structured

How an application that uses business classes is structured

If you use commands and data readers to work with the data in a database instead of using data sources and datasets, you can structure a database application as shown in figure 2-11. This *three-layer architecture* includes a *middle layer* that acts as an interface between the presentation and database layers. The middle layer typically includes classes that correspond to business entities (for example, vendors and invoices). When the classes represent *business objects*, they are commonly called *business classes*.

When you use a three-layer architecture like this, all of the code that's related to database access is stored in classes in the database layer. Then, a form class in the presentation layer can (1) call the methods of the database classes to retrieve data from the database, (2) store the retrieved data in the related business objects in the middle layer, and (3) display the data in these business objects on the form. Similarly, when the user adds, updates, or deletes data in a form, the form class in the presentation layer can (1) change the data in the related business objects, and (2) call the methods of a database class to save the changes to the database.

Although this approach to application development may seem complicated, using an architecture like this has some distinct advantages. First, it is usually easier to debug and maintain a three-layer application because you have complete control over the code. In contrast, you are forced to rely on a large amount of generated code when you use data sources and data sets.

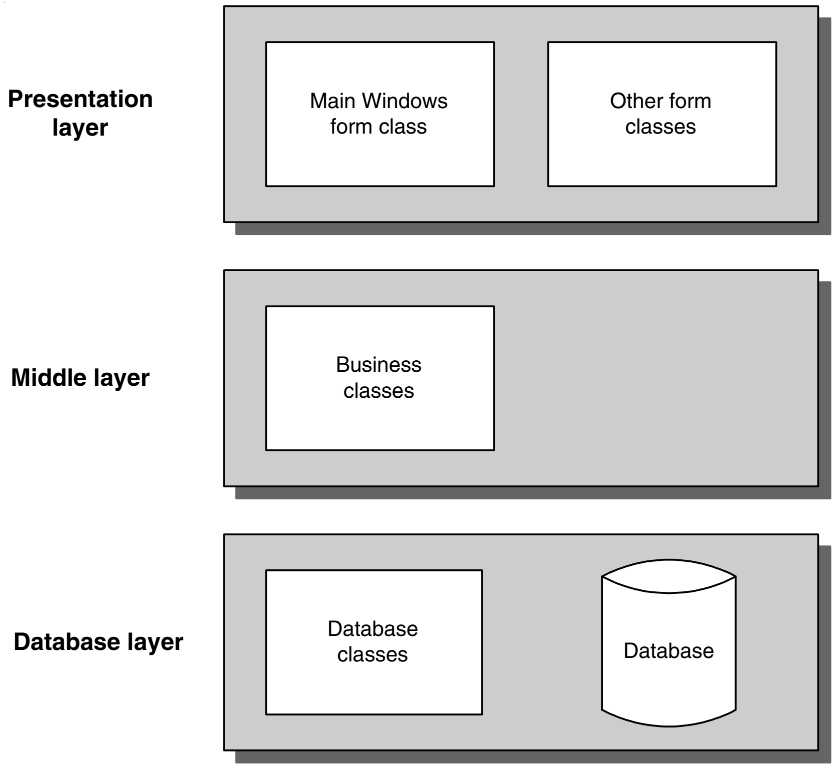
Second, a three-layer architecture allows classes to be shared among applications. In particular, the classes that make up the database and middle layers can be placed in *class libraries* that can be used by more than on project.

Third, a three-layer architecture allows application development to be spread among members of a development team. For instance, one group of developers can work on the database layer, another group on the middle layer, and a third group on the presentation layer.

Fourth, you can run different layers of an application on different servers to improve performance. In that case, a three-layer architecture is often referred to as a *three-tier architecture*. But often, these terms are used interchangeably without implying how the layers are implemented in terms of hardware.

Finally, using a three-layer architecture makes it possible to use object data sources. As you will see in chapter 9, object data sources are a new feature of Visual Studio 2005 that lets you use database classes to get the data you need for an application, store that data in business objects, and still get some of the benefits that are associated with the use of data sources.

The architecture of an application that uses business classes



Description

- To simplify development and maintenance, many applications use a *three-layer architecture* that includes a middle layer.
- The classes in the *middle layer*, sometimes called the *business rules layer*, act as an interface between the classes in the presentation and database layers. These classes can represent business entities, such as vendors or invoices, or they can implement business rules, such as discount or credit policies.
- When the classes in the middle layer represent business entities, the classes can be referred to as *business classes*, and the objects that are created from these classes can be referred to as *business objects*.
- When you use business classes, you don't use datasets. Instead, you use commands and data readers to work directly with the database, and the data you retrieve from the database is stored in business objects.
- Often, the classes that make up the database layer and the middle layer are implemented in *class libraries* that can be shared among applications.

Figure 2-11 How an application that uses business classes is structured

Perspective

One of the goals of this chapter was to introduce you to the two ways that you can develop ADO.NET applications: with datasets and without datasets. The other goal was to introduce you to some of the classes that you'll be using as you develop ADO.NET applications.

With that as background, the next three chapters in this section will show you how to build significant database applications by using data sources and datasets. As you will see, this approach is especially useful for developing simple applications or prototyping larger applications. By chapter 5, though, you'll start to see some of the limitations of this approach.

Then, the five chapters in section 2 will show you how to build database applications without using datasets. This approach lets you use the three-layer architecture, which makes it easier to debug and maintain a complicated application and also lets you reuse the code in the business and database classes. By the time you finish section 2, you'll know how to use both approaches to application development, and you'll have a good feel for when you should use each approach.

Terms

ADO.NET
ActiveX Data Objects
dataset
data table
data adapter
command
connection
disconnected data architecture
data source
Data Sources window
Component Designer tray
table adapter
data reader
concurrency
concurrency control
optimistic concurrency
concurrency exception
"last in wins"
.NET data provider
connected classes
connection string
disconnected class
data relation
data column
data row
constraint
presentation layer
database layer
prototyping
three-layer architecture
middle layer
business rules layer
business class
business object
class library
three-tier architecture