Table of Contents

Chapter 9 ........................................................................................................................................................................4
Database and ADO.NET ........................................................................................................................................................4
9.1 Introduction to Database ..............................................................................................................................................4
   Table Definitions ..........................................................................................................................................................4
   DDL and DML ............................................................................................................................................................5
   Indexes, the Primary Key, and the Foreign Key ...........................................................................................................5
   Index Uniqueness .......................................................................................................................................................5
9.2 ADO.NET Concepts .......................................................................................................................................................5
   Introduction to the SQL ...............................................................................................................................................5
   The Basic Syntax for the Select Statement ...............................................................................................................6
   Selecting More Than One Field ................................................................................................................................6
   Sorting the Returned Table .........................................................................................................................................7
   The Wild Card Character ..........................................................................................................................................7
   Specifying a Particular Record ................................................................................................................................7
   Selecting a Range of Records ...................................................................................................................................7
   Matching a Name .......................................................................................................................................................8
   Matching a Partial Text .............................................................................................................................................8
   The Inner Join and Outer Join ...................................................................................................................................8
   Adding, Updating, and Deleting Records ................................................................................................................10
   SQL Statements That Take Parameter Values .......................................................................................................10
   Removing and Creating Database Tables ................................................................................................................10
9.3 Coding the ADO.NET ................................................................................................................................................11
   ADO.NET Concepts ................................................................................................................................................11
   Dataset, Data Table, Data Row, and Data Column .................................................................................................12
   The Connection, Data Command, Data Adapter, and Table Adapter ...................................................................13
   The Phonebook Example ..........................................................................................................................................19
   Retrieving Data from the Data Source ....................................................................................................................20
   Setting Up the Connection and Data Commands .................................................................................................20
   Declaring the Objects Used in the Project ..............................................................................................................21
   Visual interface for Phonebook Maintenance .........................................................................................................19
   Declaring the Objects Used in the Project ................................................................................................................19
   Setting Up the Connection and Data Commands ...................................................................................................20
   Retrieving Data from the Data Source ....................................................................................................................20
   Deleting a Record ....................................................................................................................................................21
   Saving Data to the Data Source ..............................................................................................................................22
   Using the Data Adapter to Maintain the Database .................................................................................................22
   The Data Adapter ....................................................................................................................................................23
   Working with Data Tables ........................................................................................................................................23
   Maintaining the Database by the Data Adapter: the Phonebook Example ............................................................24
   Declaring the Required Objects ................................................................................................................................24
   Setting Up the Data Commands for the Data Adapter ............................................................................................25
   Saving the Data .......................................................................................................................................................26
Chapter 9

Database and ADO.NET

This chapter presents topics on using ADO.NET (ActiveX Data Objects) to work with databases. To provide the reader with proper background to work with ADO.NET, this chapter begins with introducing database concepts. After completing this chapter, you will be able to:

- Articulate basic relational database concepts,
- Construct basic SQL statements,
- Explain concepts pertaining to various ADO.NET objects,
- Write code to maintain data in MS Access databases, including use of data commands and data adapters,
- Write code to bind ADO.NET objects to VB controls,
- Use the dataset designer to create datasets and table adapters,
- Create bound controls using the dataset designer,
- Save data locally,
- Write code to execute stored procedures and obtain results therefrom,
- Write code to browse database schema, and
- Create and use the data relation to handle parent-child data tables in a dataset.

9.1 Introduction to Database

A database is a collection of stored data that are managed by the database management system (DBMS). There are several conceptual models for the database with one being the relational database model. Under this model, a database is a collection of tables, each consisting of rows and columns. Each row represents a record, and a column represents a field, which is the smallest data element that has practical meaning. Examples of fields include the transaction date, Social Security number, and product name. This model is popular because it’s easy to understand and use. Almost all the current commercial database software packages for personal computers are built under this model. Microsoft Access, SQL Server, FoxPro, and Oracle9i are all examples of relational database software.

Table Definitions

Two distinct types of activities are required to build a database: define the database schema and maintain data in the database. In the schema definition phase, before a database comes into existence, you will need to define tables and fields for the database. For example, if you are designing a payroll database, you will include at least an Employee table to keep data pertaining to all employees, and a Paycheck table to record all paychecks paid to employees. For each table in the database, you will further specify the fields and their respective data types. For example, the Employee table will probably have at least the following fields:

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Data Type</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>Employee Number</td>
<td>Long Integer</td>
<td></td>
</tr>
<tr>
<td>Last Name</td>
<td>String</td>
<td>15 characters</td>
</tr>
<tr>
<td>First Name</td>
<td>String</td>
<td>15 characters</td>
</tr>
</tbody>
</table>
In the data maintenance phase, the user enters, updates, and retrieves data from the tables. In many applications, the VB program serves as the front-end interface interacting with the user, while the database engine works as the backbone to handle the actual storage and retrieval of the data.

**DDL and DML**

In the preceding discussion, the language that is used to define tables is referred to as the *data definition language (DDL)*. The result of using the language that defines an overall view of the structure of the database is referred to as the *schema* of the database. The user can further define his (her) own partial views of the database. These definitions are recognized as *subschemata*.

The language that allows the user to manipulate the data is recognized as the *data manipulation language (DML)*. Data manipulations include updating, editing, adding, and deleting records.

**Indexes, the Primary Key, and the Foreign Key**

An index allows the computer to locate a record with a given field value more efficiently. An index can be built on one or several fields. For example, for an employee table, an index can be built on the Employee Number field; another index can be built on the combination of the Last Name, First Name, and Middle Initial fields.

**Index Uniqueness**

An index can be either unique or non-unique. A *unique index* will not allow a new record with the same index value as an existing one to be added. Some indexes by nature should be unique. For example, in the Paycheck table, the Check Number index should be unique. On the other hand, some indexes by nature will never be unique. For example, in the Employees table, if you build an index on Sex, the index will have approximately half of the records with the same field value!

A *primary key* is an index of which each value is *uniquely identified* with a record. By definition, a primary key *must* be a unique index, although not all unique indexes are primary keys. In a way, this distinction is more conceptual than technical. There can be only one primary key in a table. In addition, the primary key of each record must not be missing (recognized as the *entity integrity rule*). Without a value for the primary key, there will be no way to identify the record. The employee number should be the primary key of the Employee table, whereas the check number should be the primary key of the Paycheck table.

The primary key of a table is often also used in another table to establish a relation/reference to the record in that table. For example, an Employee Number field in the Paycheck table serves to indicate to whom the check is paid. The Employee Number in the Paycheck table is the *foreign key* to the Employees table.

**Introduction to the SQL**

The SQL (Structured Query Language) is a standard database query language supported by all relational database software. VB provides ways to interact with the database software that interprets and acts on the SQL. A good understanding of the SQL enables you to code VB programs that can interact with the database software smoothly. The following discussion covers some basics of the SQL that pertain to MS Access. Different software can have its own dialect. Consult the particular software manual for the specific vocabulary used by your software.

The SQL provides several commands that can be used to manipulate data. These include Select, Insert, Update, Delete, and Drop statements, providing the following functions:
Command | Use
--- | ---
Select | To extract data from DB tables
Insert | To add data to a table in the DB
Update | To change existing data in a table in the DB
Delete | To remove records from a DB table
Drop | To remove a table or index from the DB
Create Table | To create a table in the DB

The following discussion provides brief explanations for these commands.

**The Basic Syntax for the Select Statement**

The Select statement of the SQL is used to extract data (and present the results in a tabular form) from the database and has the following basic syntax:

```
Select Field1[, field2][, . . .] From Table [Where Criteria]
[Order by fields to sort];
```

where `field1`, `field2` = the list of fields in the table to be included in the results,
`Table` = the name of the table to be queried,
`Criteria` = the conditions to include records (only records satisfying the criteria will be returned), and
`Fields to sort` = the fields specified here will be used to sort the records.

The statement should conclude with a semicolon (;) although in most cases, its absence will not cause any problem.

Notice that the selection and sort criteria are optional. Here is an example of a simple SQL statement:

```
Select EmployeeNumber From Employee;
```

This statement will return a table that contains a column of all the employee numbers in the Employee table. Note that the EmployeeNumber field name is one word. MS Access allows field names that contain embedded spaces. In such a case, you will need to enclose the name in a pair of brackets to avoid confusing the database engine (server) in interpreting the code. If the field name in the DB table is Employee Number, the preceding statement will have to be changed to:

```
Select [Employee Number] From Employee;
```

Avoid creating table or field names with embedded spaces. Most DBMS do not allow names of this structure. The following discussion assumes that field and table names are not created with embedded spaces.

**Selecting More Than One Field**

To select more than one field, you will list the fields, separated by commas. For example, to select both the Employee Number and Name, you will code:

```
Select EmployeeNumber, LastName, FirstName From Employee;
```

Note that commas are used to separate the field names.
Sorting the Returned Table

You may want the previous results sorted by last name and then by first name. You can use the *Order By* clause:

```
Select EmployeeNumber, LastName, FirstName From Employee Order by LastName, FirstName;
```

Notice that SQL and VB do not have the same syntax rules. For example, in SQL, if you need to continue to the next line, you do not code any line continuation marker.

You can also specify whether the sort is in ascending or descending order by adding the *Asc* or *Desc* keyword after the sort field in the *Order* clause. The following code will sort the last name in descending order and first name in ascending order:

```
Select EmployeeNumber, LastName, FirstName From Employee Order by LastName Desc, FirstName Asc;
```

**Tip**

If your Database happens to have *Desc* (so often used for description) or *Asc* as the field or table names, be sure to enclose them in a pair of brackets when your SQL statement involves them to avoid any problem. *Desc* and *Asc* are reserved words for MS Access. For example you should code the *Desc* field name in your Journal table as follows:

```
Select Date, AccountNo, [Desc] From Journal;
```

The Wild Card Character

If you would like to select all fields in the Employee table while sorting in the same manner as done previously, you can use the wildcard * character to indicate all fields in the table:

```
Select * From Employee Order by LastName, FirstName;
```

Specifying a Particular Record

If you want only the record for employee number 1001, you can specify this criterion by the *Where* clause with the *=` operator:

```
Select * From Employee
    Where EmployeeNumber = 1001;
```

Selecting a Range of Records

Note that the comparison criterion described can also be other relational operators discussed in Chapter 5, Decision. For example, if you want all the records for employee numbers in the range of 1,000 and 2,000, you can code:

```
Select * From Employee
    Where (EmployeeNumber >= 1000) And (EmployeeNumber <= 2000);
```

Notice the use of the logical operator *And*. Specifying the criteria for the SQL is similar to specifying the conditions for an *If* statement in VB. *Or* and *Not* are also recognized logical operators in the SQL. Also, the two pairs of parentheses are used to enhance code clarity; they are not required.
Matching a Name

When the query involves a text string, the string should be enclosed in a pair of either single or double quotes. Because it is easier to handle the single quotes than double quotes in VB code, you should use the single quotes. For example, suppose you would like to select the record for the employee named Charles Smith. The SQL statement should appear:

```sql
Select * From Employee
Where (LastName = 'Smith') And (FirstName = 'Charles');
```

Note that MS Access string comparisons are not case sensitive; that is, the preceding query will produce all records that contain the name Charles Smith, whether they are uppercased, lowercased, or any combination thereof.

Matching a Partial Text

If your search requires a partial match of a text string such as all last names that begin with Sm, you can use the Like operator, with a % specification. For example, the following SQL will select all employees with a last name that begins with Sm, such as Smart, Smiley, and Smith:

```sql
Select * From Employee
Where LastName Like 'Sm%';
```

The % symbol is a wild card specification for string matching. A string of any length and of any characters will be considered a match.

If you want to limit the records to those last names with only five characters (excluding Smiley in the previous example), you will still use the Like operator. Instead of using the % wildcard specification, however, you can use the ? character, which represents a positional parameter that accepts any character as a match. You can code:

```sql
Select * From Employees
Where LastName Like 'Sm???';
```

The three ?s ensure that only those last names with five characters are selected for comparison; then as long as the first two characters are Sm, the record is considered to have satisfied the search criterion. (Note again that MS Access string comparisons are not case sensitive.)

The Inner Join and Outer Join

Suppose in the previous payroll application you would like to retrieve all fields of all checks. In addition, you would like to include the names of employees to whom each check is paid. The Paycheck table should have an Employee Number field to identify the employee being paid; however, it should not have the Employee Name fields because they are already available in the Employee table. The query involves fields that are in two tables: all fields in the Paycheck table, and the Employee Name field in the Employee table. These records should be matched by the Employee Number in the two tables.

**Inner Join**

You can use the Inner Join clause to join the two tables and obtain the desired results:

```sql
Select Employee.LastName, Employee.FirstName, Paycheck.*
From Paycheck Inner Join Employee
```
On Paycheck.EmployeeNumber = Employee.EmployeeNumber;

Notice how the fields are listed. Because the selection involves two tables, the field names should be qualified with the table names. Because the Last Name and First Name fields are in the Employee table, they are coded as:

Employees.LastName, Employees.FirstName

In addition, to indicate all fields in the Paychecks table, you code:

Paycheck.*

If a field name exists in only one table, you can omit the table name. Because LastName and FirstName appear only in the Employee table, you can also code the SQL as:

Select LastName, FirstName, Paycheck.*
From Paycheck Inner Join Employee
On Paycheck.EmployeeNumber = Employee.EmployeeNumber;

For code clarity, it is preferable to qualify the field name with the table name.

If you need to include the Where or Order clause while using the Inner Join clause, you will place it after the preceding lines. For example, if you are interested in only those checks with an amount greater than $10 and would like to sort the result by name, you can code:

Select Employee.LastName, Employee.FirstName, Paycheck.*
From Paychecks Inner Join Employee
On Paycheck.EmployeeNumber = Employee.EmployeeNumber
Where (Paycheck.Amount > 10)
Order By Employee.LastName, Employee.FirstName;

The Inner Join operation will return only those records that have the matching fields specified by the On criterion. In this example, only those checks with matching employee numbers in the Employees table will be returned. Neither employees who have no paychecks nor paychecks without matching employee numbers in the Employees table will be included. If you want either or both of these types of records, you should use the Left Join, Right Join, or Full Join clause.

**Left Join, Right Join, and Full Join**

Left Join will return all rows in the first table with or without matching rows from the second table. For example, the following statement will return all paychecks with or without matching employees.

Select LastName, FirstName, Paycheck.*
From Paycheck Left Join Employee
On Paycheck.EmployeeNumber = Employee.EmployeeNumber;

In contrast, Right Join will return all rows in the second table with or without matching rows in the first table. The following statement will return all employees with or without matching checks.

Select LastName, FirstName, Paycheck.*
From Paycheck Right Join Employee
On Paycheck.EmployeeNumber = Employee.EmployeeNumber;
Full Join will return all rows in the first table as well as the second table. Rows that have matching checks with employees will appear the same as Inner Join, while checks with no matching employees and employees with no matching checks will appear as separate rows without the missing attributes. Note, however, MS Access does not support Full Join.

Adding, Updating, and Deleting Records

The Select statement selects records from DB tables and returns the results. The SQL also provides commands that allow you to add, update, or delete records from a table. To add a record to a table, you use the Insert statement, which has the following syntax:

```
Insert Into TableName(Field1, Field2,...) Values(Value1, Value2,...);
```

For example, you can use the following statement to add a new employee record to the Employees table:

```
Insert Into Employee(EmployeeNumber, LastName, FirstName) Values(123456789, 'Doe', 'John');
```

To change the contents of a record, use the Update statement, which has the following syntax:

```
Update TableName Set Field1=Value1, Field2=Value2 Where Condition
```

For example, you can use the following statement to change the first name of employee number 123456789:

```
Update Employee Set FirstName = 'Jonathan' Where EmployeeNumber = 123456789
```

To delete a record from a table, use the Delete statement, which has the following syntax:

```
Delete From TableName Where Condition
```

To delete employee number 123456789 from the Employee table, you will code:

```
Delete From Employee Where EmployeeNumber = 123456789
```

SQL Statements That Take Parameter Values

Sometimes when an SQL statement is constructed, the value of a specific item may not be known until the statement is executed. At that time, either the user or the program that invokes the statement will need to provide the parameter value. For example, the following statement will require the user to specify the employee number before it is able to return the data pertaining to that employee:

```
Select [Name], DateOfBirth From Employee Where EmployeeNo = ?;
```

The question mark (?) is used to specify that a parameter value is required.

Removing and Creating Database Tables

The SOL statements introduced so far provide you with the capability to manipulate data in the database. The SQL can also be used to remove or create tables. The syntax to remove a table is:
Drop Table TableName;

For example, to remove the Employee table from a database, you will code:

Drop Table Employee;

The syntax for the `Create Table` statement to create a new table is:

```sql
Create Table TableName (Field1 Type[(Length)]
[, Field2 Type[(Length)]. . .);
```

where `TableName` = name of the new table to be created,
`Field1, Field2 . . .` = name of the nth field in the new table,
`Type` = data type as recognized in the database, and
`Length` = length of the field; required only for the text field; other types have their predefined lengths.

The following table shows the required type keywords for various data type fields:

<table>
<thead>
<tr>
<th>Data Type in MS Access</th>
<th>SQL Type Keywords</th>
<th>Equivalent VB 2008 Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Auto Number</td>
<td>Counter</td>
<td>(N/A; integer)</td>
</tr>
<tr>
<td>Integer</td>
<td>Integer</td>
<td>Short</td>
</tr>
<tr>
<td>Long Integer</td>
<td>Long</td>
<td>Integer</td>
</tr>
<tr>
<td>Single</td>
<td>Single</td>
<td>Single</td>
</tr>
<tr>
<td>Double</td>
<td>Double</td>
<td>Double</td>
</tr>
<tr>
<td>Date/Time</td>
<td>DateTime</td>
<td>Date</td>
</tr>
<tr>
<td>Text</td>
<td>Text</td>
<td>String</td>
</tr>
</tbody>
</table>

The following code will create a new table named Phonebook that contains two fields, Phone and Name:

```sql
Create Table Phonebook (Phone Text(10), Name Text(50));
```

This discussion provides the necessary background to explore ADO.NET. A thorough discussion of the SQL requires an entire book, and is beyond the scope of this chapter. Consult a book on database programming for a complete treatment of the subject or use the keyword, SQL Tutorial, to browse the internet for additional help.

### 9.2 ADO.NET Concepts

ADO.NET (ActiveX Data Objects Dot Net) provides a uniform set of objects that you can use to work with various kinds of databases. This chapter focuses only on the use of ADO.NET to handle the MS Access database as an introduction to database and to the ADO.NET technology.

Figure 9-1 shows an overview of how an MS database (DB) interacts with the outside world. Most likely, your previous experiences with the MS Access DB are through a direct interaction with the MS Access software, the database management system (DBMS). The figure indicates that the MS Access database is managed by the MS Access (program), which also interacts with other software systems (such as ADO.NET). The Connection object serves as a bridge for other ADO.NET objects to connect and interact with the DBMS. Most of the data
manipulations are performed with data command objects through the connection object. In VB.NET, data are stored in data tables, which can belong in, or be independent of, a dataset. Data in datasets and data tables can be built internally by your program or obtained from an external data source (e.g., MS Access DB) through the data adapter or table adapter, which has various data commands at its disposal. As you can infer from the figure, data commands can also be created and used, independent of the data adapter or table adapter.

**Figure 9-1**
ADONET Objects and Their Data Source

![Diagram of ADO.NET Objects and Their Data Source](image)

**Dataset, Data Table, Data Row, and Data Column**

Data within ADO.NET are organized into a hierarchy of objects such as datasets, data tables, data rows, and data columns. A dataset is a memory resident collection of interrelated data tables. Similar to a DB table, a data table has rows and columns. Rows are indexed by numbers. The first row has an index value of zero (0); columns are also referenced by index; and the first column has an index of zero (0), too. Columns can also be referenced with a string that represents the column name. For example, if a column has a name “Credit,” you can reference that column as `Columns(“Credit”).`

Technically, the rows (columns) are called *data rows (columns)*. All datasets, data tables, data rows, and data columns are objects. Each type has its own properties and methods. Although a dataset contains data tables, and data tables are organized by data rows and data columns (therefore appear to have a structural hierarchy), you can create independent data tables and data columns. The following code shows how these objects can be declared:

```vbnet
Dim MyDataset As New Dataset
Dim MyDataTable As New DataTable
Dim MyDataColumn As New DataColumn
Dim MyDataRow As DataRow
```

As discussed in Chapter 4, “Data, Operations and Built-in Functions,” the New keyword in the declaration creates a new object of the object type for the variable declared. Without the New keyword, the variable is not associated with any object, and is recognized only as of the type declared. Recall that an object variable can be referenced only when the object exists. Note that the DataRow object cannot be created with the New keyword. Instead, it is created with the NewRow method, which will be discussed later in this chapter.
The Connection, Data Command, Data Adapter, and Table Adapter

Data commands are used to specify the operations to be performed on the database. The operation requirement must be communicated to the DBMS which actually performs the operation. The connection object is the object that sits between the commands and the DBMS. To perform an operation on a database, the connection object needs to know which DBMS to communicate with and which database to operate on. In the connection object, the DBMS is recognized as the Provider; and the database, the data source. Additional information that the connection object may be required to provide to the DBMS includes passwords to access the database. In short, before any database operation can be performed, you must create a connection object through which you specify the provider (DBMS to use) and the data source (database) as well as any additional information required for the data source (such as the password).

The data adapter is an object that serves as an interface between the dataset (or data table) and the database. It provides methods that can be called to retrieve data from the database and place in the data table as well as to update the database based on row status in the data table. You can think of the data adapter and the data command as the actor on data, while the dataset (data table) as the holder of data.

The table adapter works in a manner similar to the data adapter. As its name implies, it is associated with a DB table. It differs from the data adapter in that several queries can be simultaneously associated with the object, while only one query can be associated with the data adapter. As will be demonstrated in section 9.4, the dataset designer can be used to visually create strongly typed table adapters, alleviating the programmer from many routine coding tasks.

ADO.NET provides two sets of data objects: SQL data objects and OleDB data objects. The former specialize in working with the SQL server; and the latter works with any generic DBMS. Because you will be working with MS Access, you will use only the OleDb data objects. For brevity, the OleDb will be omitted from the reference; e.g., OleDb data adapter will be referenced simply as the data adapter.

9.3 Coding the ADO.NET

This section explains how to code the ADO.NET objects when data commands are used to directly maintain the database. As you will realize in the next section, with the help of the dataset designer some of these coding tasks may not be required. However, it is still very important to gain a thorough understanding of the topics discussed in this section because this knowledge provides the basic background to work with the objects created by the dataset designer.

The Connection Object

To establish a connection with the data source, you create a Connection object and specify the Provider (DBMS) and the data source (database) through its Connection String property. For example, suppose you have a database, Phonebook.accdb in your default folder. The following code will set up the proper connection.

```vbnet
Dim cnnPhonebook As New OleDb.OleDbConnection
cnnPhonebook.ConnectionString = "Provider=Microsoft.ACE.oleDB.12.0; Data Source=Phonebook.accdb"
```

Note how MS Access is specified as the provider in the code. Also, the provider and data source specifications are separated with a semicolon, ";". Note also MS ACE 12.0 works with MS Access 2007. If you are using an older version (e.g., MS Access 2003), the provider should be JET 4.0 and the preceding statement should be changed to:

```vbnet
cnnPhonebook.ConnectionString = "Provider=Microsoft.JET.OleDb.4.0;Data source=
```
Using the Data Command: Adding, Updating, and Delete Records

Using the data command, the operation to be performed on the database is specified through the data command’s command text with an SQL statement. One of the two frequently used methods of the data command is then called. The two methods are listed in the following table:

<table>
<thead>
<tr>
<th>Method</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>ExecuteNonQuery</td>
<td>Execute an SQL query that does not return a result</td>
</tr>
<tr>
<td>ExecuteReader</td>
<td>Execute an SQL query that returns a reader object which contains the results of the query</td>
</tr>
</tbody>
</table>

When the data command is not expected to return any results (e.g., to delete or update records), the ExecuteNonQuery method should be called. On the other hand, when the data command is expected to return results, the ExecuteReader method should be called. In this case, the command returns a data reader, which contains the returned data.

The ExecuteNonQuery Method for Add, Update, and Delete Queries

Add, update, and delete queries do not expect any returned data. To perform such operations, first set up a data command object and specify the desired operation by an SQL statement in the command text and then call the command’s ExecuteNonQuery method. For example, suppose the Phonebook DB in the preceding subsection has a Phonebook table with the following fields:

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>PhoneNo</td>
<td>Text (primary key)</td>
</tr>
<tr>
<td>Name</td>
<td>Text</td>
</tr>
</tbody>
</table>

The following code will add a record to the Phonebook table:

```vbnet
Dim TheSQL As String
Dim cmdPhonebook As OleDb.OleDbCommand
TheSQL = "Insert Into Phonebook(PhoneNo, [Name]) Values('817)272-9999','Doe, John')"
cmdPhonebook = New OleDb.OleDbCommand(TheSQL, cnnPhonebook)
cnnPhonebook.Open()
cmdPhonebook.ExecuteNonQuery()
cnnPhonebook.Close()
```

In the above code, both the command text and the connection are specified for the data command when a new command object is created. Another way to achieve the same is to create the object first and then specify the query and connection. The code should appear as follows:

```vbnet
Dim cmdPhonebook As New OleDb.OleDbCommand 'Note the New keyword here
cmdPhonebook.CommandText = TheSQL
cmdPhonebook.Connection = cnnPhonebook
```

Notice that the connection must be open before the data command’s ExecuteNonQuery method is called. In the SQL statement, the field name, Name is enclosed in a pair of square brackets because Name is a reserved word.
in MS Access. The values for the fields are enclosed in pairs of single quotes because these values are of the Text (String) type.

To update or delete a record entails the same call to the ExecuteNonQuery method. For example replace the SQL text above with the following:

\[ \text{TheSQL} = \text{“Update Phonebook Set [Name] = ‘Wu, Jason’ Where RecordNo = 1;”} \]

Run the project. And the first record in Phonebook will have the name changed to “Wu, Jason.”

Similarly, replace the SQL text with the following:

\[ \text{TheSQL} = \text{“Delete From Phonebook Where RecordNo = 1;”} \]

Run the project. And the first record in Phonebook will be removed.

**Using Command Parameters**

In the above example, the SQL statements for the command text are re-constructed each time the command is called. A more efficient way is to assign the command text with an SQL statement that remains the same for each call. The data that change in each call are then passed to the command as parameters. This entails constructing the SQL statement to take a list of parameters.

To do it this way, several changes should be made. A command should be set up for each type of operation (insert, delete, and update) because each type calls for a different SQL statement and a different parameter list. To avoid repetitive execution of the code, the command text for each command and its parameter list should be set up in the form load event. Also, the parameter values should be assigned before the data command is called to run the ExecuteNonQuery method.

The add method of the parameters collection can be used to create parameters for the data command. This method requires up to four parameters:

\[ \text{Command.Parameters.Add(Name, Data Type, Length, Source Column)} \]

Where \( \text{Name} \) = a character string representing the name of the parameter, \( \text{Data Type} \) = The OleDb data type of the parameter (see table below), \( \text{Size} \) = the size of the parameter in bytes; for example a date field has a size of 8 bytes, and \( \text{Source column} \) = the column name that requires this parameter.

The following table gives a list of selected data types and their corresponding OleDb types.

<table>
<thead>
<tr>
<th>VB Data Type</th>
<th>OleDb Type</th>
<th>Size in bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>BSTR</td>
<td>(varying)</td>
</tr>
<tr>
<td>Boolean</td>
<td>Boolean</td>
<td>2</td>
</tr>
<tr>
<td>Date</td>
<td>Date</td>
<td>8</td>
</tr>
<tr>
<td>Double</td>
<td>Double</td>
<td>8</td>
</tr>
<tr>
<td>Single</td>
<td>Single</td>
<td>4</td>
</tr>
<tr>
<td>Long</td>
<td>BigInt</td>
<td>8</td>
</tr>
<tr>
<td>Integer</td>
<td>Integer</td>
<td>4</td>
</tr>
<tr>
<td>Short</td>
<td>SmallInt</td>
<td>2</td>
</tr>
</tbody>
</table>
Look It Up

Use the keyword, “OleDbType enumeration” to search in the Index tab of the help file for a complete list of available OleDb data type and the corresponding compatible VB data type.

For example, the above code to delete a record will have to be restructured as follows:
Declare a delete command at the class level:

```vbscript
Dim cmdDelete As New OleDb.OleDbCommand
```

Add the following code in the form load event:

```vbscript
cmdDelete.CommandText = “Delete From Phonebook Where PhoneNo = ?;”
’Add a parameter to the parameter list
cmdDelete.Parameters.Add(“PhoneNo”, OleDb.OleDbType.BSTR, 50, “PhoneNo”) 
```

The SQL statement for the command text indicates that the phone number value will be supplied as a parameter. The add method of the command’s parameters object adds a parameter to the parameter list. This parameter is specified to have a name, PhoneNo, to be of the string type, to be of 50 characters long, and to be associated with the source column name, PhoneNo. Finally, the following block of code should be used to delete a record:

```vbscript
’Delete a record
cnnPhonebook.Open
cmdDelete.Parameters(“PhoneNo”).Value = mskPhoneNo.Text
cmdDelete.ExecuteNonQuery() ‘Delete the record from Phonebook DB table
```

As you can see from the code, you no longer need to specify the command text in the procedure that is called to delete a record. Instead, you assign the phone number to be deleted to the PhoneNo parameter. In the same manner, you can modify the code to insert and update the Phonebook DB table. This additional work is left to you.

Retrieving Data from the Data Source

Executing the SQL statements through the data command as illustrated in the preceding subsection expects no returned results from the database. Often, however, executing SQL statements, such as the Select statement, expects data to be returned from the database. In such case, the data command’s ExecuteReader method should be used.

The Data Reader and Its Read Method

The ExecuteReader method returns a data reader object, which consists of a read-only, forward-only stream of data. The data reader’s Read method reads one row of data at a time; that is, the method advances to the next row of the returned data set. The method also returns a value of True if the record exists, and False if no more data is available. If the record exists, you can obtain the field values in the row in several different ways:

- Use a numeric index. For example, Assume the DataReader object is named MyReader. MyReader(0) (or MyReader.GetValue(0)) will then return the value of the first field.
- Use the field name. For example, MyReader(“PhoneNo”) (or MyReader.Item(“PhoneNo”)) will return the value of the field named PhoneNo.
Use one of the DataReader’s data typed accessor methods. Each method allows you to obtain the value in the field’s native data type. For example, the GetInt32 method will allow you to obtain a field of the Integer type. The advantage of using this approach is its speed. Data can be obtained without involving data type conversion; however, you must know the field’s data type before calling the method. The following table shows a selected list of these methods:

<table>
<thead>
<tr>
<th>Method</th>
<th>To access VB Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetDateTime</td>
<td>Date/Time</td>
</tr>
<tr>
<td>GetDouble</td>
<td>Double</td>
</tr>
<tr>
<td>GetFloat</td>
<td>Single</td>
</tr>
<tr>
<td>GetInt16</td>
<td>Short</td>
</tr>
<tr>
<td>GetInt32</td>
<td>Integer</td>
</tr>
<tr>
<td>GetInt64</td>
<td>Long</td>
</tr>
<tr>
<td>GetString</td>
<td>String</td>
</tr>
</tbody>
</table>

All these methods require an index as the parameter that indicates the ordinal position of the data field to access. For example, if you want to retrieve the second field that is an Integer, code

\[\text{TheDataReaderName.GetInt32(1)}\]. As an example, suppose in the preceding Phonebook example, you want to show the returned data in a list box named lstPhonebook. The following code will serve the purpose.

```vba
'Declare ADO.NET objects in the general declaration area
Dim cnnPhonebook As New OleDb.OleDbConnection
Dim cmdSelect As New OleDb.OleDbCommand
Dim rdPhonebook As OleDb.OleDbDataReader 'the data reader

'Place the following code in a procedure to obtain the results
Dim ThePhone As String
Dim TheName As String
Dim lstPhonebook As New System.Windows.Forms.ListBox
lstPhonebook.Items.Add("PhoneNo, [Name] From Phonebook")

'Set up the data command in the Form load event
'Also assume cnnPhonebook has been set up properly
Dim cmdSelect As New OleDb.OleDbCommand
cmdSelect.Connection = cnnPhonebook
cmdSelect.CommandText = "Select PhoneNo, [Name] From Phonebook"
cmdSelect.ExecuteNonQuery()
rdPhonebook = cmdSelect.ExecuteReader()
Do While rdPhonebook.Read() 'Point to next record
    'Obtain the first field (Phone)
    ThePhone = rdPhonebook.GetString(0)
    'Obtain the second field (name) and assign it to the name
    TheName = rdPhonebook.GetString(1)
    'Add data to the list box
    lstPhonebook.Items.Add(TheName & vbTab & ThePhone)
Loop
rdPhonebook.Close
cnnPhonebook.Close
```

The first block of the above code declares the required objects; the second, sets up the data command; and the last, obtains the results. As you can infer, the returned data come from executing the SQL Select statement which is assigned to the data command’s command text property. Because the data reader’s Read method returns a Boolean value, True when the row of data exists, the Do loop will continue until the returned set runs of data. Again, recall that the code `rdPhonebook.GetString(0)` and `PhonebookReader("PhoneNo")` will obtain the same result.

Note that as long as the data reader is open, it retains exclusive use of the connection; therefore, the data reader should be closed as soon as the returned data have been read or loaded. Note also the data reader
(rdPhonebook) is declared without the New keyword. The data reader identifier is needed only to reference the returned data reader which is created by the command’s ExecuteReader method.

**Tip**

Sometimes, a Select query may return some empty data. If the value is to be assigned to a numeric variable, it can cause an error to your code. For example, suppose you have a VB statement:

```vbnet
Amount = rdSales("Amount")
```

Where Amount is of the Double type and rdSales is the data reader. If the returned value turns out to be empty, your code will stop at that line because the empty value is of the DBNull type, which cannot be converted to any numeric data. You should use the TypeOf operator to test the data type first as follows:

```vbnet
If TypeOf(rdSales("Amount")) Is DBNull Then
    Amount = 0
Else
    Amount = rdSales("Amount")
End If
```

**The Load Method of the Data Table**

As an alternative to the above approach, if you want to keep the results in memory, you can use the `load method` of the data table to load the results into the data table for further processing. For example, suppose you want to retrieve all rows from the Phonebook table in the preceding example. You can declare a data table and use its Load method to fill the results into the data table. The following code illustrates this approach:

```vbnet
'declare a data table to load the query results
Dim dtbPhonebook as New DataTable
'other statements of the first two blocks remain the same

'place the following code in the procedure to obtain results
'in place for the third block
cnnPhonebook.Open()
'execute the execute reader method and assign results to the data reader
rdPhonebook = cmdSelect.ExecuteReader
'load the data in the data reader into the data table
dtbPhonebook.Load(rdPhonebook)
rdPhonebook.Close()
cnnPhonebook.Close()
```

In the above routine, the results from the ExecuteReader method are assigned to the data reader, rdPhonebook. The data table’s Load method is then used to load the returned data from the data reader.

**Tip**

When a data table is used to load the results from the data reader, you can actually bypass declaring and referencing a data reader. For example, the above code can be replaced with the following without declaring the data reader:

```vbnet
dtbPhoneBook.Load(cmdSelect.ExecuteReader)
```
The Phonebook Example

This example puts together the discussion in this section so far into a practical application. It can be used to maintain the Phonebook table in the Phonebook database.

Visual interface for Phonebook Maintenance

The program will work in the following manner:

- After the user has entered a phone number into a masked textbox, the program will check whether the record exists in the Phonebook DB table. If it does, the name is displayed in the textbox for name; otherwise, the textbox is cleared.
- When the user clicks the Update button, the program will check whether the record exists and either add a new row or change an existing row in the database, depending on existence status of the record.
- When the user clicks the Delete button, the program will check whether the record exists and delete the record if it exists; otherwise the program will display a “record does not exist” message.
- When the user clicks the Quit button, the program quits.

The following table shows the settings for the properties of the controls in the form:

<table>
<thead>
<tr>
<th>Control</th>
<th>Property</th>
<th>Setting</th>
<th>Remark</th>
</tr>
</thead>
<tbody>
<tr>
<td>Group box</td>
<td>Text</td>
<td>Phonebook Info</td>
<td></td>
</tr>
<tr>
<td>Label</td>
<td>Text</td>
<td>Name (Last, First)</td>
<td></td>
</tr>
<tr>
<td>Label</td>
<td>Text</td>
<td>Phone Number</td>
<td></td>
</tr>
<tr>
<td>Text box</td>
<td>Name</td>
<td>txtName</td>
<td>For the user to enter the name</td>
</tr>
<tr>
<td>Masked textbox</td>
<td>Name</td>
<td>mskPhoneNo</td>
<td>To enter phone number</td>
</tr>
<tr>
<td></td>
<td>Mask</td>
<td>(999) 000-0000</td>
<td></td>
</tr>
<tr>
<td>Button</td>
<td>Name</td>
<td>btnDelete</td>
<td>To delete record (row)</td>
</tr>
<tr>
<td></td>
<td>Text</td>
<td>Delete</td>
<td></td>
</tr>
<tr>
<td>Button</td>
<td>Name</td>
<td>btnSave</td>
<td>To update the database</td>
</tr>
<tr>
<td></td>
<td>Text</td>
<td>Save</td>
<td></td>
</tr>
<tr>
<td>Button</td>
<td>Name</td>
<td>btnQuit</td>
<td>To end the project</td>
</tr>
<tr>
<td></td>
<td>Text</td>
<td>Quit</td>
<td></td>
</tr>
</tbody>
</table>

Declaring the Objects Used in the Project

The ADO.NET objects used in this example include the connection, data commands, and the data reader and are declared at the class level as follows:

```verbatim
' Declare ADO.NET objects in the general declaration area
Dim cnnPhonebook As New OleDb.OleDbConnection
Dim cmdSelect As New OleDb.OleDbCommand
Dim cmdInsert As New OleDb.OleDbCommand
Dim cmdUpdate As New OleDb.OleDbCommand
Dim cmdDelete As New OleDb.OleDbCommand
Dim rdPhonebook As OleDb.OleDbDataReader 'the data reader
```
Setting Up the Connection and Data Commands

For efficiency, SQL statement parameters will be used. As shown above, four data commands are declared so that the SQL statements and their parameters can be set up only once. The data reader (rdPhonebook) will be used to retrieve data returned from the Select command (cmdSelect).

The connection and data commands should be ready for use once the program starts and therefore should be set up in the form load event. For code clarity, a separate procedure is written to set up all the required data commands but is called in the form load event. These procedures appear as follows:

```vbnet
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As System.EventArgs)
Handles MyBase.Load
    cnnPhonebook.ConnectionString = "Provider=Microsoft.ACE.OleDb.12.0;" 
        "data source=Phonebook.accdb"
    SetupCommands()
End Sub

Private Sub SetupCommands()
' Set up Select command and its parameter
    cmdSelect.Connection = cnnPhonebook
    cmdSelect.CommandText = "Select PhoneNo, [Name] From Phonebook Where PhoneNo=?"
    cmdSelect.Parameters.Add("PhoneNo", OleDb.OleDbType.BSTR, 20, "PhoneNo")
' Set up Insert command and its parameter
    cmdInsert.Connection = cnnPhonebook
    cmdInsert.CommandText = "Insert Into Phonebook (PhoneNo, [Name]) Values(?,?)"
    cmdInsert.Parameters.Add("PhoneNo", OleDb.OleDbType.BSTR, 20, "PhoneNo")
    cmdInsert.Parameters.Add("Name", OleDb.OleDbType.BSTR, 20, "Name")
' Set up Update command and its parameter
    cmdUpdate.Connection = cnnPhonebook
    cmdUpdate.CommandText = "Update Phonebook Set [Name]=? Where PhoneNo=?"
    cmdUpdate.Parameters.Add("Name", OleDb.OleDbType.BSTR, 20, "Name")
    cmdUpdate.Parameters.Add("PhoneNo", OleDb.OleDbType.BSTR, 20, "PhoneNo")
' Set up Delete command and its parameter
    cmdDelete.Connection = cnnPhonebook
    cmdDelete.CommandText = "Delete From Phonebook Where PhoneNo=?"
    cmdDelete.Parameters.Add("PhoneNo", OleDb.OleDbType.BSTR, 20, "PhoneNo")
End Sub
```

Retrieving Data from the Data Source

When the user enters a phone number, the program will retrieve the record from the database if the record exists. The routine to retrieve the record is written as a function as follows:

```vbnet
Private Function GetARecord(Optional ByVal Fill As Boolean = True) As Boolean
' The optional parameter will indicate whether to place the name in the text box
' if the record is found
    Dim RecordFound As Boolean
    cmdSelect.Parameters("PhoneNo").Value = mskPhoneNo.Text
    cnnPhonebook.Open()
    rdPhonebook = cmdSelect.ExecuteReader
    If rdPhonebook.Read Then
        RecordFound = True ' Indicate record exists
        If Fill Then
            txtName.Text = rdPhonebook.GetString(1)
        End If
    ElseIf Fill Then
...
This function uses the Select command to retrieve the record using the phone number specified by the user in mskPhoneNo. The function actually serves several purposes. It returns a Boolean value to indicate whether the record actually exists. This value can be used by other routines that need to determine existence of the record. Recall that the data reader’s Read method advances the record pointer by one position and if the record exists, it returns a value True. The variable, RecordFound can then be set to True and returned to the caller at the end of the function.

The function takes a parameter to decide whether to place the returned data on screen. Recall that the Select SQL statement expects two returned fields: PhoneNo and Name. The following statement places the retrieved name in the textbox:

```vbnet
txtName.Text = rdPhonebook.GetString(1)
```

When the user enters the phone number and if the routine finds the record, it should fill the screen with the retrieved name (no need to do the same for the phone number because the entered one and the retrieved one are the same). The mskPhoneNo’s Leave event will be triggered when the user proceeds to the next control and can be coded as follows to retrieve the record:

```vbnet
Private Sub mskPhoneNo_Leave(ByVal sender As Object, ByVal e As System.EventArgs) Handles mskPhoneNo.Leave
    GetARecord(True) 'Retrieve the record and place it on screen if found
End Sub
```

### Deleting a Record

When the user clicks the Delete button, the program will check whether the record specified in mskPhoneNo indeed exists and proceeds to delete the record if it exists; otherwise a message “record does not exist” is displayed. The event procedure appears as follows:

```vbnet
Private Sub btnDelete_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles btnDelete.Click
    Dim RecordExists As Boolean
    'Check if the record exists, but don't display the name
    RecordExists = GetARecord(False)
    If RecordExists Then
        cnnPhonebook.Open()
        'Record exists; delete
        cmdDelete.Parameters("PhoneNo").Value = mskPhoneNo.Text
        cmdDelete.ExecuteNonQuery() 'delete the record
        cnnPhonebook.Close()
    Else
        'Record doesn't exist; send a message
        MsgBox("Record does not exist; can't delete")
    End If
End Sub
```
The above code shows that the GetARecord function is called to get an indication as to whether the record exists. As explained above, this function returns a value True when the specified record exists.

**Saving Data to the Data Source**

When the user clicks the Save button, your code needs to determine whether the phone number already exists in the DB table. If it does, the record should be updated; otherwise, it is a new record and should be added to the database. The complete code to handle the Save button click event appears as follows:

```vbnet
Private Sub btnSave_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles btnSave.Click
Dim RecordExists As Boolean
' check if the record exists; but don't put the name in the textbox
RecordExists = GetARecord(False)
If RecordExists Then
    cnnPhonebook.Open()
    ' record exists; update
    cmdUpdate.Parameters("Name").Value = txtName.Text
    cmdUpdate.Parameters("PhoneNo").Value = mskPhoneNo.Text
    cmdUpdate.ExecuteNonQuery() ' update the record
    cnnPhonebook.Close()
Else
    cnnPhonebook.Open()
    ' record does not exists; add
    cmdInsert.Parameters("PhoneNo").Value = mskPhoneNo.Text
    cmdInsert.Parameters("Name").Value = txtName.Text
    cmdInsert.ExecuteNonQuery() ' Insert the record
    cnnPhonebook.Close()
End If
End Sub
```

**Tip**

If you run a project and encounter the following error,
The 'Microsoft.Ace.oledb.12.0' provider is not registered on the local machine.
verify that the provider is specified correctly. If everything appears to be correct and you are running the project under a 64-bit operating system, do the following to fix the problem:

1. Click Project in the IDE menu and click project Properties (last option in the menu) to open the project properties window.
2. Click Compile on the left frame of the project property window.
3. Click the Advanced Compile Options button.
4. Change the Target CPU box from AnyCPU to x86. Then, click the OK button.

**Using the Data Adapter to Maintain the Database**

The example in the preceding subsection uses data commands to maintain the database. All operations are carried out immediately on the database. An alternative approach is to perform all the operations in memory on a data table first and update the database as a batch later. This involves the use of the data adapter and data table. All the operations are first done on the data table, which keeps the statuses of all the changes made to the rows. Then, the data adapter’s Update method is called to perform all the data maintenance operations as indicated in the row statuses in the data table. The remainder of this subsection explains in detail how this approach is done.
The Data Adapter

The data adapter is an object that serves as an interface between the dataset (or data table) and the data database. The following table lists the three frequently used methods of the data adapter:

<table>
<thead>
<tr>
<th>Method</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fill(dataset or data table)</td>
<td>Fill the dataset or data table with data that are specified by the adapter’s select command</td>
</tr>
<tr>
<td>Update(dataset or data table)</td>
<td>Update the data source based on the row states in the dataset/data table</td>
</tr>
<tr>
<td>FillSchema(Data table, Schema type)</td>
<td>Fill the data table with a schema; where schema type can be either Mapped or Source; in most case Mapped should be specified (look up in the help menu for additional explanation)</td>
</tr>
</tbody>
</table>

The Fill method is used to bring data from the data source to the dataset (or data table). Various operations on the data in the dataset, such as add or modify rows, can then be performed. The data adapter’s Update method is used to update the data source. The data adapter performs various database operations by calling relevant data commands (e.g., Select, Update, and Delete commands) in the object. You will need to set up these commands in the same manner as in the preceding example where data commands were used.

Because each DB table calls for a unique set of data commands to maintain its contents, typically a data adapter is used to interact with a particular DB table; that is, if you want to maintain two DB tables, you should use two data adapters. The data table’s schema is automatically set up when the data adapter’s Fill method is called to fill the data table. Sometimes, however, data may have to be added to the data table without the Fill method being called first. In such cases, the data adapter’s FillSchema method can be used to set up the schema for the data table.

Working with Data Tables

The data table is an object that holds data in memory in tabular form similar to a database table. It consists of rows (data rows) and columns (data columns). Data in the data table can be referenced by specifying the row and column indexes of interest. For example, if dtbPhonebook has PhoneNo and Name columns, then dtbPhonebook.Rows(0)(0) refers to the PhoneNo in the first row. The column reference can also be a string. Thus, dtbPhonebook.Rows(0) ("PhoneNo") refers to the same cell. The following code will place the referenced phone number in the string variable named ThePhoneNo:

```vbnet
Dim ThePhoneNo As String
ThePhoneNo = dtbPhonebook.Rows(0)("PhoneNo")
```

The following code will place the phone number entered in txtPhoneNo in the same cell.

```vbnet
dtbPhonebook.Rows(0)("PhoneNo") = txtPhoneNo.Text
```

The data table object has many useful methods. The following table gives a selected list:

<table>
<thead>
<tr>
<th>Method</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>AcceptChanges()</td>
<td>Update status of all rows and make them current</td>
</tr>
<tr>
<td>Clear()</td>
<td>Remove all rows from the data table</td>
</tr>
<tr>
<td>Load(DataReader)</td>
<td>Load data from a data reader into the data table</td>
</tr>
<tr>
<td>Select(Criterion)</td>
<td>Select rows that match the selection criterion</td>
</tr>
</tbody>
</table>
For example, the following code will remove all rows from dtbPhonebook:

```csharp
dtbPhonebook.Clear()
```

The Select method selects all the rows that match the selection criterion expressed in a character string. The following code will return and display the vendor number of all rows (in dtbPurchase) whose RecordNo are greater than 10:

```csharp
Dim SelectedRows() As DataRow
Dim I As Integer
SelectedRows = dtbPurchase.Select("RecordNo > 10")
For I = 0 To SelectedRows.Length - 1
    MsgBox(SelectedRows.Rows(I)("VendorNo")
Next
```

Note that the array, SelectedRows() in the above code does not contain new rows independent of the data table, dtbPurchase but simply references the same rows in the data table. Any operation that your code performs on the array will be reflected in the corresponding rows in the data table. For example, the following code will remove the first row whose RecordNo is greater than 10 from the data table:

```vbnet
'Remove the first selected row with a RecordNo greater than 10 from dtbPurchase
SelectedRows(0).Delete
```

**Maintaining the Database by the Data Adapter: the Phonebook Example**

To illustrate how the data adapter can be used to maintain the database, the same phonebook example in the previous subsection is used. Keep in mind the key difference of this new example is that all the operations will be performed on the data table first. This program should behave as follows:

- As soon as the program starts, the data adapter will fill a data table with records from the Phonebook DB table.
- After the user has entered a phone number into a masked textbox, the program will check whether the record exists in the data table. If it does, the name is displayed in the textbox; otherwise, the textbox is cleared.
- When the user clicks the Update button, the program will check whether the record exists and either add a new row or change an existing row in the data table, depending on existence status of the record.
- When the user clicks the Delete button, the program will check whether the record exists and delete the record from the data table if it exists; otherwise the program will display a “record does not exist” message.
- When the user clicks the Update DB button, the program will invoke the data adapter’s Update method and also make all rows in the data table current.
- When the user clicks the Quit button, the program quits.

**Declaring the Required Objects**

This project requires a connection, a data adapter, and a data table. They are declared at the module (class) level as follows:

```csharp
Dim cnnPhonebook As New OleDb.OleDbConnection
Dim odaPhonebook As New OleDb.OleDbDataAdapter
Dim dtbPhonebook As New DataTable
```
Setting Up the Data Commands for the Data Adapter

As soon as the program starts, the connection and all data commands for the data adapter should be set up to be ready for use. The code to set up the connection and the adapter’s data commands is similar to the preceding example and is shown below:

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles MyBase.Load
    cnnPhonebook.ConnectionString = "Provider=Microsoft.ACE.OleDb.12.0;Data Source=Phonebook.accdb"
    SetupCommands()
    odaPhonebook.Fill(dtbPhonebook)
End Sub

Private Sub SetupCommands()
    With odaPhonebook
        'Create and set up select command
        .SelectCommand = New OleDb.OleDbCommand
        .SelectCommand.Connection = cnnPhonebook
        .SelectCommand.CommandText = "Select PhoneNo, [Name] From Phonebook"
        'Create and set up Insert command
        .InsertCommand = New OleDb.OleDbCommand
        .InsertCommand.Connection = cnnPhonebook
        .InsertCommand.CommandText = "Insert Into Phonebook (PhoneNo, [Name]) Values(?,?)"
        .InsertCommand.Parameters.Add("PhoneNo", OleDb.OleDbType.BSTR, 20, "PhoneNo")
        .InsertCommand.Parameters.Add("Name", OleDb.OleDbType.BSTR, 40, "Name")
        'Create and set up Update command
        .UpdateCommand = New OleDb.OleDbCommand
        .UpdateCommand.Connection = cnnPhonebook
        .UpdateCommand.CommandText = "Update Phonebook Set [Name]=? Where PhoneNo=?"
        .UpdateCommand.Parameters.Add("Name", OleDb.OleDbType.BSTR, 40, "Name")
        .UpdateCommand.Parameters.Add("PhoneNo", OleDb.OleDbType.BSTR, 20, "PhoneNo")
        'Create and set up Delete command
        .DeleteCommand = New OleDb.OleDbCommand
        .DeleteCommand.Connection = cnnPhonebook
        .DeleteCommand.CommandText = "Delete From Phonebook Where PhoneNo=?"
        .DeleteCommand.Parameters.Add("PhoneNo", OleDb.OleDbType.BSTR, 20, "PhoneNo")
    End With
End Sub
```

As you can see from the code in the SetupCommands sub, the data adapter’s four commands are literally named SelectCommand, InsertCommand, UpdateCommand, and DeleteCommand. These commands are set up exactly the same way as the preceding example. They must be set up before the data adapter’s Update method can be called.

Note that in the form load event procedure there is a statement that invokes the data adapter’s Fill method to fill the data table after a call to set up the data commands. The Fill method uses the data adapter’s Select Command to retrieve data from the database and fill them in the data table. In this example, all records in the Phonebook DB table are placed in the data table for the convenience of updating and verifying the existence of any records. Note also that the data adapter automatically opens and closes the connection. You should not
try to open or leave the connection open before using the data adapter. Nor should you try to close the connection after using the data adapter.

**Saving the Data**

To save data to the database, you save the data to the data table and then call the data adapter’s Update method to perform the operation. However, you typically will call the data adapter’s Update method only once after you have performed all the updates on the data table.

To save the data to the data table, the code should first determine whether the phone number entered exists in the data table. If it does, the existing name should be changed. Otherwise, it is a new record. A new row should be created and added to the data table. You can use the data table’s Select method to select the row that has a matching phone number as specified by the user in the mskPhoneNo. The following code shows how this can be done.

```vbnet
Private Sub btnSave_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles btnSave.Click
' Save data to the data table
Dim FoundRows() As DataRow
FoundRows = dtbPhonebook.Select("PhoneNo='" & mskPhoneNo.Text & "]'"")
If FoundRows.Length > 0 Then
' Recod exists; update the row
FoundRows(0)("Name") = txtName.Text
Else
' Record does not exists; create a new row and add to the table
Dim NewRow As DataRow
NewRow = dtbPhonebook.NewRow
NewRow("PhoneNo") = mskPhoneNo.Text
NewRow("Name") = txtName.Text
'delete the new row to the data table
dtbPhonebook.Rows.Add(NewRow)
End If
End Sub
```

**Deleting A Record**

To delete a record, the code should determine whether the phone number exists by using the data table’s Select method. The matching row is deleted by using the data table’s Delete method.

```vbnet
Private Sub btnDelete_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles btnDelete.Click
' Delete a record from data table
Dim FoundRows() As DataRow
FoundRows = dtbPhonebook.Select("PhoneNo='" & mskPhoneNo.Text & "'")
If FoundRows.Length > 0 Then
' record exists; delete
FoundRows(0).Delete()
Else
MsgBox("Record des not exists; can't delete")
End If
End Sub
```
Updating the Database

Notice that none of the preceding event procedures call the data adapter’s Update method, which should be called only once after all intended updates (including Insert, Update, and Delete) have been performed on the data table. The data table keeps track of the changes you made and maintains several versions for each row: current, original, modified, and deleted. The data adapter’s Update method will determine which data command (Insert, Update, or Delete) to use to update the database by examining the status of the rows. Once the data adapter’s Update method is executed, the database will be updated and the data table’s AcceptChanges method should be called to update the data table, keeping only the current version of each row. The AcceptChanges method should be called only after the database is updated.

```vbnet
Private Sub btnUpdate_Click(ByVal sender As System.Object, ByVal e As System.EventArgs)
    Handles btnUpdate.Click
    'Use the row states in the data table to update the database
    odaPhonebook.Update(dtbPhonebook)
    'Now make all rows in the data table current
    dtbPhonebook.AcceptChanges()
End Sub
```

Tip

Call the data table’s accept changes method only after the data adapter’s update method has been called to update the database. After the data table’s accept changes method is executed, only the current version for each row is kept and the data adapter’s update method will have no clue to determine what to do.

Binding Data Tables to Controls

Data in the data table can be displayed by binding it to a VB control. This entails setting up one or two properties of the control. The following discussion provides a few illustrations.

Binding the Data Table to the Data Grid View

The data grid view is a new control introduced to replace the data grid of previous versions. It displays data in a tabular form and offers many interesting functionalities. The brief example here shows only a simple binding example. To bind the data table to a data grid view, all you have to do is to assign the data table to the data grid view’s data source property. For example, assume you have drawn a data grid view on the form and named it dgvPhonebook. You can have the contents of the data table in the preceding subsection displayed by the following code:

```vbnet
dgvPhonebook.DataSource = dtbPhonebook
```

Note that whatever the user does to the contents in the data grid view will be reflected in the data table by default and a call to the Update method of its associated data adapter will update the database accordingly. If you do not want the user to accidentally alter the database contents, set the data grid view’s read only property to True:

```vbnet
dgvPhonebook.ReadOnly = True
```
Binding the Data Table to Other Controls

The data table can also be bound to other controls such as the combo box or list box. To bind a column to the combo box or list box, assign the data table and its column to the control’s data source and the display member properties, respectively as shown in the code below:

```csharp
cboName.DataSource = dtbPhonebook
cboName.DisplayMember = "Name"
```

The code binds the data table to the combo box and specifies that the Name column be displayed. Typically, a column is bound to a combo box to show a list for the user to select from the available choices, not to enter data. For this purpose, the combo box’s drop down style property should be set to drop down list.

The combo box also has a ValueMember property, which provides the value corresponding to the item selected from the display member. Suppose in this example you also set the combo box’s ValueMember property to the data table’s PhoneNo column as follows:

```csharp
cboName.ValueMember = "PhoneNo"
```

Then, when the user selects a name from the combo box, the control’s SelectedValue property gives the phone number corresponding to the selected name. Binding columns of a data table to a combo box this way can provide a convenient means for the user to specify a foreign key in data entry. For example, in a payroll data entry screen, instead of having the user enter the employee number, you can set up a combo box with its DisplayMember property bound to the employee name column and its ValueMember property bound to the employee number column of a data table. Your program can then obtain the employee number from the SelectedValue property after the user selects the employee name.

**Tip**

Your code can encounter run time errors if you bind the combo box to a data table as suggested in this sub-subsection and attempt to access the SelectedValue property in the control’s SelectedIndexChanged event. This happens because during the binding process, the event is triggered before the SelectedValue property is ready. One way to resolve this problem is to avoid accessing the SelectedValue property in that event. If you have to access the property in that event, one possible solution is to set a class level Boolean variable to True at the beginning of the routine to bind the control to the data table; and set the variable back to False and the end of the routine. Your code can then test the value of this variable in the SelectedIndexChanged event and proceed to access the SelectedValue property only when the Boolean variable is False.

9.4 Using the Dataset Designer

Visual Basic 2008 comes with a powerful dataset designer that allows the programmer to visually create datasets and their associated table adapters. It can also create controls that are bound to data tables. The following discussion deals with these topics.
Creating Datasets and Table Adapters Visually

This subsection illustrates how to use the dataset designer to create datasets and their associated table adapters. Consider the phonebook project example discussed in the last subsection of the preceding section. Suppose you want create the data table and the adapter visually rather than with code. You can use the dataset designer to accomplish this. Follow these steps:

1. Start a new project with a new form.
2. Click Data on the menu bar in the IDE.
3. Select the Add New Data Source the menu option (See figure 9-2).
4. Click the Database icon in the Data Source Configuration Wizard (See Figure 9-2). Then click the Next button.

Figure 9-2
The Data Menu and the Data Source Configuration Wizard

Click the Data Menu and then select Add New Data Source.
The Data Source Configuration Wizard (below) will appear.

Figure 9-3
The Add Connection Dialog Box

If the Data Source box shows something different from Microsoft Access database file, click this Change button to set it.
Click the Browse button to locate the desired database.
5. Click the New Connection button in the Choose Your Data Connection dialog to invoke the Add Connection dialog box. If the Choose Data Source dialog box appears, select Microsoft Access Database File and click the Continue button; otherwise, click the Change button and select Microsoft Access Database File in the Add Connection dialog box if the Data source box shows a different item (Figure 9-3).

6. Click the Browse button in the same dialog box and locate the phonebook database (Figure 9-3).
7. Click the Test Connection button in the same dialog box to verify that the test is successful.
8. Click the OK button in the message box then click the OK button in the Add Connection dialog box.
9. Click the Next button in the Choose your Data Connection dialog.
10. Click No when asked whether to copy the file and modify the connection (See Figure 9-4).
11. Click on the “Yes, save the connection as” check box and click the Next button (See Figure 9-5).
12. Click on the Tables check box in the Choose Your Database Objects dialog (Figure 9-6) then click the Finish button.

Notice that the above steps will create a data table and its associated table adapter for each DB table your database contains; that is, if the database contains 10 tables, there should be 10 data tables in this dataset. Each data table so created has columns corresponding to the DB tables; and its associated table adapter has the four commands (select, insert, update, and delete) set up ready for use.

Figure 9-5
The Save the Connection String Dialog

Make sure this check box is checked and then click the Next button.

You should see PhonebookDataSet.xsd in your Solution Explorer. This is the dataset type (component) created from the above steps. Double click this object. You should see the Phonebook data table including its associated table adapter appear in place of the code window (Figure 9-7).

When you specify a local MS Access file as the connection for your project, you will be asked whether to copy the file to your project’s output directory. Click No if you want the file to reflect the results of the data operations by your
Make sure this Tables check box is checked and then click the Finish button.

You can also specify the dataset name.

After you complete the dialog with the Data Source Configuration Wizard, the Phonebook dataset should appear in your Solution Explorer.

Notice the Data Sources tab appears right below the Solution Explorer. When you click this tab, the Data Sources window will appear in place of the Solution Explorer.
Using Strongly-Typed Datasets and Table Adapters

Suppose you would like to use the dataset and table adapter created with the above steps to replicate the Phonebook project illustrated in the last subsection of the preceding section. With the same visual interface as in that example, the following code should achieve the same objectives:

```vbnet
Dim dtbPhonebook As New PhonebookDataSet.PhonebookDataTable
Dim taPhonebook As New PhonebookDataSetTableAdapters.PhonebookTableAdapter
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles MyBase.Load
    'If you need to specify the location of your DB, use the following sample line
    'taPhonebook.Connection.ConnectionString = "put your connection string here"
    taPhonebook.Fill(dtbPhonebook)
End Sub
Private Sub btnSave_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles btnSave.Click
    Dim FoundRows() As DataRow
    FoundRows = dtbPhonebook.Select("PhoneNo='" & mskPhoneNo.Text & "'")
    If FoundRows.Length > 0 Then
        'Record exists; update the row
        FoundRows(0)("Name") = txtName.Text
    Else
        'Record does not exists; create a new row and add to the table
        Dim NewRow As DataRow
        NewRow = dtbPhonebook.NewRow
        NewRow("PhoneNo") = mskPhoneNo.Text
        NewRow("Name") = txtName.Text
        'add this new row to the data table
        dtbPhonebook.Rows.Add(NewRow)
    End If
End Sub
Private Sub btnDelete_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles btnDelete.Click
    When you double click the Phonebook dataset in the Solution Explorer (above), the contents of the dataset will appear in the main window. This object shows the Phonebook data table and its associated table adapter. Each table in your database will have a data table created here.
```
'Delete a record from data table
Dim FoundRows() As DataRow
FoundRows = dtbPhonebook.Select("'PhoneNo='' & mskPhoneNo.Text & '''")
If FoundRows.Length > 0 Then
    'record exists; delete
    FoundRows(0).Delete()
Else
    MsgBox("Record does not exists; can't delete")
End If
End Sub

'To update the database
Private Sub btnUpdate_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles btnUpdate.Click
    'Use the row states in the data table to update the database
    taPhonebook.Update(dtbPhonebook)
    'Now make all rows in the data table current
    dtbPhonebook.AcceptChanges()
End Sub

The code in this project differs from the preceding example in that:
- dtbPhoenbook is declared to be of the PhonebookDataSet.DataTable type, instead of the generic data table type.
- A table adapter of the PhonebookDataSetTableAdapters.PhonebookTableAdapter is declared in place of the generic data adapter.
- There is no declaration or setup of a connection object.
- There is no setup for the data commands or the table adapter.

The tasks of declaring and setting up the connection as well as commands for the table adapter have been automatically handled by the dataset designer. All the remaining code stays the same except for a name difference: taPhonebook for the table adapter here versus odaPhonebook for the data adapter in the preceding example. As the sample code implies, the table adapter also has the Fill and Update methods, which work in exactly the same way as the data adapter's.

Although you do not need to set up the connection, you can still modify the connection string when you have a need. The table adapter contains a connection object, by which you can make the modification. As shown in the comments in the form load event procedure, you can assign your modification to taPhonebook.Connection.ConnectionString.

Adding Queries to the Table Adapter

The Select query built in for the select command of the table adapter selects all rows from its source DB table; i.e., the query does not include a Where clause to filter records. If you need to select rows based on certain filters (e.g., select rows with names that begin with A or phone number by area code), you can add queries to the table adapter. For example, suppose in the phonebook example you would like to add a query that selects rows by phone number from the phonebook. You can do the following:

1. Double click PhonebookDataset.xsd in the Solution Explorer to make the phonebook dataset structure appears in the main window.
2. Right click PhonebookTableAdapter in the Phonebook data table that appears in the code window area (See Figure 9-7).
3. Select Add Query in the context menu.
4. Click the Next button in the TableAdapter Configuration Wizard dialog box.
5. Click the radio button that shows “SELECT which returns rows.” Then click the Next button.
6. Enter “SELECT PhoneNo, [Name] FROM Phonebook Where PhoneNo Like ?” (without quotes) in the “Specify a SQL SELECT statement” dialog. Then click the Next button.

7. Enter FillByPhoneNo in the Fill a Data Table box and GetDataByPhoneNo in the Return a Data Table box. Then click the Next button.

8. Click the Finish button in the Wizard Results dialog.

These steps will add two additional methods to the PhonebookTableAdapter: the FillByPhoneNo method to fill selected data to the PhonebookDataTable and the GetDataByPhoneNo method to return a data table. The SQL statement in step 6 allows the user to specify a phone number parameter for the returned data table.

To see how to use the FillByPhoneNo method works, first add the following controls to the form:

<table>
<thead>
<tr>
<th>Control</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Text box</td>
<td>txtSearch</td>
</tr>
<tr>
<td>Data grid view</td>
<td>dgvPhonebook</td>
</tr>
<tr>
<td>button</td>
<td>btnSearch</td>
</tr>
</tbody>
</table>

Then, add the following code in the btbSearch_Click event procedure:

```csharp
taPhonebook.FillByPhoneNo(dtbPhonebook, txtSearch.Text)
dgvPhonebook.DataSource = dtbPhonebook
```

If the user enters “(817)%” (without quotes) in the txtSearch text box and clicks the Search button, the data grid view should display all the records containing the 817 area code.

Notice that each table adapter works with only one DB table. If you need to execute a query that involves multiple DB tables, you will still need to use data commands discussed in the preceding section.

**Creating Bound Controls**

The dataset designer can also create bound controls. For example, suppose you want to create the same data entry screen for the preceding phone book project but with the controls bound to the phonebook data table. You can achieve this by doing the following:

1. Create a new form and create the phonebook dataset with its associated table adapter(s) by following steps outlined in the first subsection of this section. Afterwards, make sure the form appears in the main window before proceeding.
2. Click the Data Sources tab that appears below the Solution Explorer window (Figure 9-8).
3. Click the Phonebook node in the Data Sources window. Then click its dropdown arrow (Figure 9-8).
4. Click Details in the drop down list.
5. Click the Expand (+) button on the left of the Phonebook node.
6. Click PhoneNo in the expanded list and then click its dropdown arrow (Figure 9-9).
7. Select MaskedTextBox from the dropdown list. If MaskedTextBox is not in the list, do the following:
   a. Click Customize at the bottom of the dropdown list to invoke the Options dialog box; and then click the checkbox for MaskedTextBox in the dialog box.
   b. Click the OK button in the Options dialog box.
   c. Repeat 6 and 7.

**Figure 9-8**

*The Data Sources Window*
8. Drag the Phonebook node on to the form. A masked textbox for phone number and a textbox for name along with their respective descriptive labels as well as a binding navigator with a row of buttons should appear on the form (Figure 9-10). In addition, a list of components will appear below the form.

Steps 6 and 7 set up the masked text box for the phone number field. In general, if you want a field to be bound to a control other than a text box, you can follow the general approach illustrated in these two steps. You can set the properties for all controls so created. For example, you can change the masked text box’s name to mskPhoneNo and set its mask property to “(999) 000-0000” (without quotes).

You should also see the following code in the code window:

```vbnet
Private Sub PhonebookBindingNavigatorSaveItem_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles PhonebookBindingNavigatorSaveItem.Click
    Me.Validate()
    Me.PhonebookBindingSource.EndEdit()
    Me.TableAdapterManager.UpdateAll(Me.PhonebookDataSet)
End Sub

Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles MyBase.Load
    'TODO: This line of code loads data into the 'PhonebookDataSet.Phonebook' table. You can move, or remove it, as needed.
    Me.PhonebookTableAdapter.Fill(Me.PhonebookDataSet.Phonebook)
End Sub
```

Figure 9-10
As you can see, the code uses the table adapter to fill the phonebook data table when the program starts and save the data to the database when the user clicks the Save button on the binding navigator.

The buttons on the binding navigator perform various data operations:

- The four navigation buttons (move first, move previous, move next, and move last) move to the row in the data table as their names suggest; e.g., the Move First button moves to the first row of the data table. The bound masked text box and text box show the contents of the current row.
- The Add New button adds a new row to the data table.
- The Delete button deletes the current row displayed in the bound controls.
- The Save Data button saves the data in the data table to the database, using the data table’s Update method.

Notice that the way data are maintained here is different from the previous examples. Specifically, in the previous examples, data in the controls are validated before being added to the data table. Here a new row is added to the data table before data in the bound controls are moved into the new row. Therefore, you must ensure that data in the data table are valid before updating the data source. In general, you should validate all rows in the data table before calling the table adapter’s Update method.

Notice also that you can have the selected contents of the DB table bound to a data grid view instead. To do so, select data grid view in step 4 in the above procedure and then drag the Phonebook node to the form.

### 9.5 Additional Topics in ADO.NET

This section discusses a selected list of additional capabilities of the ADO.NET, including saving data locally, browsing database schemata, executing stored procedure, and building relations between data tables in a dataset. You should find these capabilities helpful in various real world applications.

#### Saving Data “Locally”

In some cases, you may find it desirable to save the data in the dataset or data table “locally.” Recall that the dataset and data tables are a memory resident collection of data. These tables may not correspond exactly to their data sources. Keeping the current states of the dataset in the client computer (the one where the dataset resides) may enhance systems efficiency because the user can then proceed without having to reconstruct the dataset (data tables) back to where it was. The dataset can simply be reloaded. In addition, the persisted (saved) dataset can also be used by other applications that understand the structure of the saved data.

The dataset and data tables read and write data (and schemas) as XML documents. (But you do not really have to know the XML to work with the dataset.) Suppose in the previous Phonebook example you want to save the data in the data table locally. You can use its *WriteXML method*:
dtbPhonebook.WriteXML(“C:\Temp\Phonebook.XML”)

Later, you can read it back using its **ReadXML method**:

```csharp
dtbPhonebook.ReadXML(“C:\Temp\Phonebook.XML”)
```

You can also save the data table schema information using the **WriteXMLSchema method**:

```csharp
dtbPhonebook.WriteXMLSchema(“C:\Temp\Phonebook.xsd”)
```

To read the same schema, use the **ReadXMLSchema method**:

```csharp
dtbPhonebook.ReadXMLSchema(“C:\Temp\Phonebook.xsd”)
```

## Browsing Database Table Definitions (Schema)

From time to time, you may need to browse the database for its schema. ADO.NET connection object provides such a capability with the **GetOleDbSchemaTable method**. The following example project is used to illustrate how the method works and will:

- Show all the tables of a database in a list box named lstSchemaTables after the user specifies the database.
- Show (in another list box, named lstSchemaColumns) the field names of a table selected by the user from the preceding list box.
- Display (in a data grid view) the contents of the data table that stores the schema of the database.

The program begins with the user specifying the database by entering the filename in a textbox, or by clicking a Browse button that will invoke an Open File dialog to prompt for the filename. Figure 9-11 shows the visual interface at run time. The following table lists the names of the controls used in the code.

<table>
<thead>
<tr>
<th>Control</th>
<th>Name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Text box</td>
<td>txtFileName</td>
<td>To enter or display the specified database</td>
</tr>
<tr>
<td>Button</td>
<td>btnBrowse</td>
<td>To prompt for the filename using the open file dialog</td>
</tr>
<tr>
<td>Data grid view</td>
<td>dgvSchema</td>
<td>To display the contents of returned schema</td>
</tr>
<tr>
<td>List box</td>
<td>lstSchemaTables</td>
<td>To display the names of the tables in the database</td>
</tr>
<tr>
<td>List box</td>
<td>lstSchemaColumns</td>
<td>To display the names of the columns of the selected table</td>
</tr>
<tr>
<td>Open file dialog</td>
<td>cdlOpenFile</td>
<td>For the user to select a file</td>
</tr>
</tbody>
</table>
The GetOleDbSchemaTable method returns a data table with the specified schema information. It has the following syntax:

```
Connection.GetOleDbSchemaTable(Guid, Criteria)
```

where `Connection` is the connection object to perform the schema query;

`Guid` is globally unique identifier that specifies the type of schema query such as Tables (for information pertaining to tables in the database) or Columns; and

`Criteria` is an Object type array specifying the criteria for the query. Different query type will require different array parameters.

The following table shows the criteria list for the Tables and Columns queries:

<table>
<thead>
<tr>
<th>Query Type (Guid)</th>
<th>Criteria</th>
<th>Remark</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tables</td>
<td>TABLE_CATALOG, TABLE_SCHEMA, TABLE_NAME, TABLE_TYPE</td>
<td>To query for the table names in the database, specify “Table” as the TABLE_TYPE. The criteria array will appear as {Nothing, Nothing, Nothing, “Table”}.</td>
</tr>
<tr>
<td>Columns</td>
<td>TABLE_CATALOG, TABLE_SCHEMA, TABLE_NAME, COLUMN_NAME</td>
<td>To query for the column names of a particular table, give the name of the table as the TABLE_NAME. Assume the table name is tblPhonebook. The criteria array will appear as {Nothing, Nothing, “tblPhonebook”, Nothing}.</td>
</tr>
</tbody>
</table>

**Look It Up**

For a complete list of the query types and the criteria corresponding to each type, use the keyword “OpenSchema method” (not “GetOleDbSchemaTable method”) in the Index tab to search the help file. Be sure that the Filter by box in the tab specifies no filter. Double-click the first found topic (at the Host Integration Server: Platform SDK) listed in the Index Results for… area. The page provides a list of queries and the corresponding criteria supported for different systems.
To use the method, you need to open the connection and provide the proper parameter values. Because it will return results in data table type, you will begin by declaring two DataTable variables and a connection variable. They should be declared at the module level as follows:

```
Dim cnnDatabase As New OleDb.OleDbConnection()
Dim tblSchemaTables As DataTable 'To hold data related to tables
Dim tblSchemaColumns As DataTable 'To hold data related to columns
```

The filename (data source) for the connection string comes from what the user specifies in the text box. The code should appear as:

```
cnnDatabase.ConnectionString = "Provider=Microsoft.ACE.OLEDB.12.0; Data Source=" & txtFileName.Text
```

As shown above, the GetOleDbSchemaTable method also requires a parameter that specifies the query criteria, which should be an array of the Object type. The criteria vary depending on the schema query type. For example, if the query is for “Tables,” this criteria (second) parameter expects an array that specifies four criteria as given in the preceding table. To specify these criteria, you can declare an array variable of the Object type as follows:

```
Dim Criteria(3) As Object
```

If you are interested in every kind of “tables” in the schema, you will not need to specify any criteria but just pass the variable as the second parameter. Because you are interested in only the “table” type of the TABLE_TYPE (the fourth criterion), you should specify the criteria as:

```
Criteria(3) = "Table"
```

The code to query for the tables in the database will then appear as:

```
cnnDatabase.Open()
tblSchemaTables = cnnDatabase.GetOleDbSchemaTable(Data.OleDb.OleDbSchemaGuid.Tables, Criteria)
cnnDatabase.Close()
```

**Tip**

As shown in the preceding table, the object array can be specified as {Nothing, Nothing, Nothing, “Table”}; therefore, an alternative way to code the statement to query for the database schema table without using the Criteria variable is:

```
tblSchemaTables = _
    cnnDatabase.GetOleDbSchemaTable(Data.OleDb.OleDbSchemaGuid.Tables, _
    New Object() {Nothing, Nothing, Nothing, "Table"})
```

The returned result is assigned to the tblSchemaTables data table variable. Notice that the connection must be open before the method can be called and is immediately closed after the method is executed. You can then bind the data table to the data grid view and the list box named lstSchemaTables. Similar to the previous example of binding a data column to the combo box, the Table_Name column of the data table is bound to this list box with the code:
The complete code in a sub procedure to show the table names in a database appears as follows:

```vbnet
Dim cnnDatabase As New OleDb.OleDbConnection()
Dim tblSchemaTables As DataTable 'To hold data related tables
Dim tblSchemaColumns As DataTable 'To hold data related to columns
Private Sub ShowDBSchema()
    Dim Criteria(3) As Object
    Criteria(3) = "Table"
    cnnDatabase.ConnectionString = _
        "Provider=Microsoft.ACE.OLEDB.12.0;Data Source=" & txtFileName.Text
    cnnDatabase.Open()
    tblSchemaTables = cnnDatabase.GetOleDbSchemaTable(OleDb.OleDbSchemaGuid.Tables, Criteria)
    cnnDatabase.Close()
    dgvSchema.DataSource = tblSchemaTables
    lstSchemaTables.DataSource = tblSchemaTables
    lstSchemaTables.DisplayMember = "Table_Name"
End Sub
```

Notice how the Guid for the query type is specified. “Tables” is enumerated in the OleDb name space under the OleDbSchemaGuid enumeration. (Enumeration will be discussed in Chapter 12, “Object-Based Programming.”)

Recall that when the user clicks the Browse button, or when the user enters the database name in the text box named txtFileName, the table names in the database should be displayed. Each of these two events should call the ShowDBSchema procedure. The two event procedures should appear as follows:

```vbnet
Private Sub btnBrowse_Click(ByVal sender As System.Object, ByVal e As System.EventArgs)
Handles btnBrowse.Click
    cdlOpenFile.Filter = "MS Access File (*.accdb)|*.accdb"
    If cdlOpenFile.ShowDialog = Windows.Forms.DialogResult.Cancel Then
        Exit Sub
    End If
    txtFileName.Text = cdlOpenFile.FileName
    ShowDBSchema()
End Sub

Private Sub txtFileName_KeyPress(ByVal sender As System.Object, ByVal e As System.Windows.Forms.KeyPressEventArgs)
Handles txtFileName.KeyPress
    If Asc(e.KeyChar) = Keys.Return Then
        'enter key; call ShowDBSchema
        ShowDBSchema()
    End If
End Sub
```

**Try This**

Enter the above code and run the project. After the program is working properly, comment out the statement Criteria(3) = “Table” and run the program again. Examine the contents of the data grid view carefully. You should notice that without the Criteria statement, the data grid contains many “tables” that are not user created. They are created by the system, and are usually of no interest to the user.
Querying for the Column Names of a DB Table

When an item (a DB table name) in the list box, lstSchemaTables, is clicked, the column names in that table should appear in another list box, lstSchemaColumns. Again, the GetOleDbSchemaTable method can be used to obtain the column names. The statement to obtain the result appears as follows:

```
tblSchemaColumns = cnnDatabase.GetOleDbSchemaTable(OleDb.OleDbSchemaGuid.Columns, Criteria)
```

The first parameter (OleDb.OleDbSchemaGuid.Columns) specifies the query type to be “columns.” As mentioned previously in this section, the second parameter varies with the first. As shown in the preceding table, under the “Columns” specification, the second parameter specifies four criteria: TABLE_CATALOG, TABLE_SCHEMA, TABLE_NAME, and COLUMN_NAME. Notice that the third criterion relates to the table name. Because you are interested in only those columns of a particular table, the table name should be specified as the third element of the Criteria parameter. The variable, Criteria should be set up as follows:

```
Dim Criteria(3) As Object
Criteria(2) = lstSchemaTables.Text
```

The code to populate the list box with the field names of the table selected by the user is as follows:

```
Private Sub lstSchemaTables_SelectedIndexChanged(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles lstSchemaTables.SelectedIndexChanged
    Dim Criteria(3) As Object
    Criteria(2) = lstSchemaTables.Text
    cnnDatabase.Open()
    tblSchemaColumns = cnnDatabase.GetOleDbSchemaTable(OleDb.OleDbSchemaGuid.Columns, Criteria)
    cnnDatabase.Close()
    lstSchemaColumns.DataSource = tblSchemaColumns
    lstSchemaColumns.DisplayMember = "Column_Name"
End Sub
```

The returned data table has a column containing the field names. The column name for this column is Column_Name; therefore, “Column_Name” is specified as the display member. Notice that the code is placed in the list box’s SelectedIndexChanged event, which is raised when the user clicks the list box or when the control’s SelectedIndex is assigned a value.

**Executing Stored Procedures**

The data command can be used to execute procedures (e.g., queries) stored in the database. To execute a stored procedure, you need to:

- Set the command’s command text property to the stored procedure name.
- Set the command’s command type property to StoredProcedure.
- Set up parameters for the stored procedure if it requires any parameters. The way to set up the parameters is the same as explained in Section 9.3.
- Call either the command’s ExecuteNonQuery method or ExecuteReader command depending on whether the stored procedure returns any results.

The following code example illustrates how to execute a stored procedure that returns a result set and does not require any parameter value. The results are loaded into a data table called dtbAccountBal.

```
Dim cmdAccountBal As New OleDb.OleDbCommand
```
Dim BalanceReader As OleDb.OleDbDataReader
Dim dtbAccountBal As New DataTable
'Assume cnnAccounting is a valid connection
cmdAccountBal.Connection = cnnAccounting
'qryAccountBal is the Query stored in the database
cmdAccountBal.CommandText = "qryAccountBal"
'Indicate that the command type is stored procedure
cmdAccountBal.CommandType = CommandType.StoredProcedure
cnnAccounting.Open
'Now execute the stored procedure and store the results in BalanceReader
BalanceReader = cmdAccountBal.ExecuteReader
dtbAccountBal.Load(BalanceReader)
BalanceReader.Close
cnnAccounting.Close

**Dataset and Data Relation**

All previous examples involve one data table. In many practical applications, several interrelated data tables may be called for. These data tables can be collected into a dataset in which you can also set up their relations. These data relations enable you to specify the parent-child relationship, just like the database. (Note, however, that the relations in a database are not automatically copied onto the dataset.) Most databases contain many tables that have a one-to-many (parent-child) relationship. For example, each customer can have many sales orders, so the Customer table and the Order table have a one-to-many (parent-child) relationship. Conventionally, to retrieve the related child records given a parent record, the Join (inner join or outer join) operation is performed. The resulting table tends to contain many repeated data elements from the parent record. With the data relation, the GetChildRows method makes it convenient to retrieve related child rows given a parent row without involving the Join operations. This subsection shows how this can be done.

**Creating a Dataset with Multiple Data Tables By Code**

To illustrate, suppose in the Phonebook example, there is also a DB table, CallLog, which logs all calls made and has the following fields:

<table>
<thead>
<tr>
<th>Field</th>
<th>Data Type</th>
<th>Remark</th>
</tr>
</thead>
<tbody>
<tr>
<td>RecordNo</td>
<td>Auto number</td>
<td>Primary key</td>
</tr>
<tr>
<td>PhoneNo</td>
<td>Text</td>
<td>Call made to</td>
</tr>
<tr>
<td>TimeBeg</td>
<td>Date/Time</td>
<td>Time the call begins</td>
</tr>
<tr>
<td>TimeEnd</td>
<td>Date/Time</td>
<td>Time the call ends</td>
</tr>
</tbody>
</table>

The phoneNo field in this table is a foreign key to the Phonebook table. And the two tables have a parent-child one-to-many relationship with the Phonebook table being the parent. You want to create a project that allows you to review the phone calls: when you select a phone number from Phonebook (displayed in a combo box), your program shows all calls made to it.

To follow the approach discussed here, you will read the data in both tables into two data tables, which should belong to the same dataset and create the relation by the phone number. To bring the data into the data tables, two data adapters are needed: one for each table. The following code provides declarations for these objects and should be placed at the class level:

Dim cnnPhonebook As New OleDb.OleDbConnection
Dim odaPhoneBook As New OleDb.OleDbDataAdapter
Dim odaCallLog As New OleDb.OleDbDataAdapter
Dim dtbPhoneBook As DataTable
Dim dtbCallLog As DataTable
Dim dsPhonebook As New DataSet
Dim relPhone2Log As DataRelation

Your code then will proceed to establish the connection, create the two data tables in the dataset, bring the
data into the two data tables, and establish the relation between the two data tables. Suppose you have also
drawn a combo box, cboPhoneNo, on the form to for the user to select a phone number of interest. The
following block of code meets all of these requirements:

' set up connection
cnnPhonebook.ConnectionString = "Provider=Microsoft.ACE.OleDb.12.0;data source= & _
"Phonebook.accdb"
' Set up Select command for adapters
odaPhoneBook.SelectCommand = __
    New OleDb.OleDbCommand("Select PhoneNo, [Name] From Phonebook", cnnPhonebook)
odaCallLog.SelectCommand = __
    New OleDb.OleDbCommand("Select PhoneNo, TimeBeg, TimeEnd From CallLog", cnnPhonebook)
' Create data tables in the data set
dtbPhoneBook = dsPhonebook.Tables.Add("PhoneBook")
dtbCallLog = dsPhonebook.Tables.Add("CallLog")
' Fill both tables
odaPhoneBook.Fill(dsPhonebook.Tables(0))
odaCallLog.Fill(dsPhonebook.Tables(1))
' Set up relation between the two data tables
relPhone2Log = dsPhonebook.Relations.Add("Phone2Log", __
    dtbPhoneBook.Columns("PhoneNo"), dtbCallLog.Columns("PhoneNo"))
cboPhoneNo.DataSource = dtbPhoneBook
cboPhoneNo.DisplayMember = "PhoneNo"

Notice the two data tables are created in the dataset with its Add method, which takes a string parameter for
the name of the data table to be created. The newly created tables are also assigned to two object variables
(dtbPhonebook and dtbCallLog) of the data table type for the convenience of reference. Recall that an
assignment of one object to another object identifier (ObjectB = ObjectA) does not create two copies of
the same object. Instead, both object identifiers refer to the same existing object. That is, in the above code,
dtbPhoneBook and dsPhonebook.Tables(0) (or dsPhonebook.Tables("Phonebook") ) refer to the same object.
The above code also binds the PhoneNo column in dtbPhonebook to the combo box, cboPhone.

The relation between the two data tables is created with the add method of the dataset’s relations
collection, which has the following syntax:

Relation = Dataset.Relations.Add(Name, Parent Column, Child Column)

Where Relation = the relation object

Name = a string as the name for the relation
Parent column = the column in the parent data table for the relation (similar to the primary key)
Child column = the column in the child data table for the relation (similar to the foreign key)

Using the Relation and the Get Child Rows Method
The relation established in the above code can be used to obtain the child rows with the following syntax:
ChildRows = ParentTable.Rows(Index).GetChildRows(Relation)

The returned child rows should be a data row array. The index gives the position of the particular row, and the relation is the data relation that defines the linkage of the two tables. In our example, when the user makes a PhoneNo selection from the combo box, the program will display all the phone calls made to that number. The code appears as follows:

```vbnet
Private Sub cboPhoneNo_SelectedIndexChanged(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles cboPhoneNo.SelectedIndexChanged
    Dim LogRows() As DataRow
    Dim I As Integer
    If cboPhoneNo.SelectedIndex >= 0 Then
        LogRows = dtbPhoneBook.Rows(cboPhoneNo.SelectedIndex).GetChildRows(relPhone2Log)
        'Code to display the results
    End If
End Sub
```

The returned rows (LogRows) are similar to the results from using the child data table’s select method. Indeed, the following code should produce the same results:

```vbnet
LogRows = dtbCallLog.Select("PhoneNo='" & cboPhoneNo.Text & "'")
```

You can then proceed to handle the returned rows with proper code.

**Summary**

- Two distinct activities are required to create a database. The structure (schema) of the database must be defined first. Data can then be collected, stored, and maintained in the database. The data definition language (DDL) is used to define the database schema, while the data manipulation language (DML) is used to operate and maintain the database.
- A relational database consists of a collection of tables, in which each row represents a record and each column represents an attribute.
- An attribute in a table that uniquely identifies each record is recognized as the primary key. The value of the primary key field of each record must exist and be unique (the entity integrity rule).
- A column that contains the primary key of another table is recognized as the foreign key. The foreign key must be either null or a value that exists in the other table. This rule is recognized as the referential integrity rule.
- The SQL Select statement is used to construct a virtual table from the database; its basic syntax is as follows:
  ```
  Select Field1, Field2, ... From Table1[, Table2,...]
  [Where (condition1) [([And|Or](Condition2)) [([And|Or](Condition3))]]...]
  [Order By Sortfield1 [, Sortfield2, [Sortfield3... ] [Asc|Desc]]]
  ```
  The From clause can also contain Inner Join specification.
- The SQL Insert, Delete, and Update statements are used to add, delete, or update records in database tables.
• The SQL Create Table statement is used to create new tables in a database; and the Drop Table statement, to remove a table from the database.
• ADO.NET consists of a set of objects to handle data.
• The dataset is a memory resident collection of data tables and is independent of the data source, while the data (table) adapter serves as an interface between the data sources and the dataset.
• The data command’s ExecuteNonQuery method executes SQL statements that return no results; and the command’s ExecuteReader method executes statements that return results in a data reader whose contents can be read or loaded into a data table.
• The data (table) adapter provides methods that can be used to extract data from the data source and fill the dataset. The data adapter’s Update method can be used to add, delete, and update records in the data source, based on the status of the data rows in the dataset. In performing the operations, the method uses the proper commands (Select, Insert, Delete, and Update).
• The dataset and the data table each provide a set of methods that can be used to create and modify the contents of the dataset. The dataset’s Tables.Add method adds data tables to the dataset. The data table’s Rows.Add (Rows.Delete) adds (deletes) rows to (from) a data table.
• Changes made to the dataset (add, update, and delete) cause certain rows to have several versions (current, modified, or removed). The dataset’s AcceptChanges method commits these changes, and makes all data rows current. If you mean to have the data source to reflect all changes made in the dataset, call the data (table) adapter’s Update method to update the data source before calling the dataset’s AcceptChanges method.
• Various controls can be bound to the dataset to display its contents. The data grid view can be used for such a purpose to display an entire data table.
• You can use the dataset designer to visually create datasets and their associated table adapters as well as to create controls bound to data tables.
• With the dataset designer, you can also visually add queries to table adapters to suit your needs.
• You can save the dataset in XML using the dataset’s WriteXML method. The saved dataset can be reloaded into a dataset using the ReadXML method. The dataset’s schema can be saved with the dataset’s WriteXMLSchema method and then reloaded with the ReadXMLSchema method.
• You can use the connection’s GetOleDbSchemaTable method to browse the schema of a database.
• Procedures stored in a database can be executed by a data command with proper property settings.
• When a dataset consists of more than one data table, you can use the dataset’s Relations.Add method to set up relations between the tables. The GetChildRows method returns the rows pertaining to a parent row.

Explore and Discover

9-1. Validating Bound Controls. Create the phonebook project with bound controls by following the steps outlined in the subsection of Creating Bound Controls in Section 9.5. Rename the masked text box for PhoneNo as mskPhoneNo. Then enter the following code:

```c
Imports System.Text.RegularExpressions  'Put this line above the Public Class Form1 line
Private Sub mskPhoneNo_Validating(ByVal sender As Object, ByVal e As System.ComponentModel.CancelEventArgs) Handles mskPhoneNo.Validating  
If Regex.IsMatch(mskPhoneNo.Text, "\[(\]\d{3}\[) \d{3}-\d{4}\)") Then 'good data; do nothing
  Else
    If MsgBox("Must have a legitimate phone number; proceed anyway?", MsgBoxStyle.Information Or MsgBoxStyle.Question Or MsgBoxStyle.YesNo) _
      MsgBoxStyle.YesNo
  End If
End Sub
```
Then
'ignore
Else
    e.Cancel = True
End If
End If
End Sub

Click the Add New button on the binding navigation strip. Then enter some incomplete phone number and click the Delete button. The program will display an error message. But you have no way to delete the new row before entering a legitimate phone number. If you comment out the e.Cancel statement in the code, your user will be able to enter a bad phone number and click the Save button to save the data. You will then have no way to ensure that the saved data are all valid. This is why you need to validate data in the data table before saving them to the database.

9-2. Bound Controls for Tables with Parent-Child Relationship. Refer to the Phonebook project in this chapter. Create the Phonebook database with the Phonebook table as specified in the example along with several rows of data. Then create the CallLog table with the following fields:

<table>
<thead>
<tr>
<th>Field name</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>RecordNo</td>
<td>Auto number</td>
</tr>
<tr>
<td>PhoneNo</td>
<td>Text (same mask as the same field in Phonebook)</td>
</tr>
<tr>
<td>TimeBeg</td>
<td>Date/Time; beginning time of a call</td>
</tr>
<tr>
<td>TimeEnd</td>
<td>Date/Time; ending time of a call</td>
</tr>
</tbody>
</table>

Create a relationship between the two tables while working on the database in MS Access:
1. Click the Database Tools menu.
2. Click the Relationship icon.
3. Select both tables on the main window from the Show Table dialog box. (Highlight one table name at a time and click the Add button.) Then click the Close button in the dialog box.
4. Drag PhoneNo from Phonebook to Phonebook in CallLog.
5. Check the Enforce Referential Integrity check box when the Edit Relationships dialog box appears. Then click the Create button.
6. Click Yes when prompted whether to save the relationship when you are closing the Relationships main window.
7. Enter a few rows of data in the CallLog table. Then quit from MS Access.

Create a new form and follow the steps in the subsection of Creating Bound Controls in Section 9.5 to create bound controls for the Phonebook table. You should notice that there is a CallLog node inside the Phonebook node although there is also a separate main CallLog node. Click the dropdown arrow on the CallLog node inside the Phonebook node. Select DataGridView in the list. Then drag this node to the form below the controls bound the Phonebook data table. What do you see?

When you run the program and navigate around the phonebook table, the phone calls pertaining to the phone number of the selected row should appear in the data grid view.

Exercises

9-3. Maintaining the CallLog Table. Create a CallLog table as specified in 9-2 for the Phonebook database. (But do nothing beyond creating the table.) Develop a project to maintain this table.
There should be a button to save the call log. For simplicity, do not provide code to delete any record. The textbox for RecordNo should be left blank for new record. If the user enters a number, use it to retrieve an existing record for editing. If the record is not found, give an error message. Your code should use both the data adapter and the data command. The SQL statements should be constructed on the spot (without the use of any parameter).

9-4. Maintaining the CallLog Table with Parameters. Modify Exercise 9-3 so that the SQL statements are constructed only once and actual values of data to update the CallLog table are passed to commands by parameter. Also, if you use a textbox (or masked textbox) for the user to enter the phone number, replace it with a combo box with its data source property bound to the Phonebook table, its display member bound to Name, and its value member bound to PhoneNo. This should allow the user to easily identify the phone number needed and avoid violating the referential integrity rule.

9-5. Maintaining the CallLog Table by the Table Adapter. Modify Exercise 9-4 by using the Dataset Designer to create the Phonebook dataset and its associated table adapters. Add an Update Database button on the form. Your code should handle all changes (Save or Delete) to data in the data table. Update to the database should be performed only when the Update Database button is clicked.

9-6. Viewing Call Logs by Phone Number. Using the database you have developed in exercise 9-5, develop a project that will allow you to review phone calls by phone number. The form should have a list box that lists all names in the Phonebook table. (Hint: Set the list box’s data source, display member, and value member properties properly. These properties function in the same way as those for the combo box.) There should also be a data grid view. When you select a name in the list, all calls (pertinent rows in the call log table) you have made to this person should appear in the data grid view. The phone number (value member in the list box) should be used as a parameter for the command to retrieve the related calls.

9-7. Viewing Call Logs by Phone Number (Dataset and Relation). Modify Exercise 9-6 so that the project involves the use of a dataset that contains two data tables, which keep the contents of Phonebook and CallLogs. Create the dataset with the dataset designer.

9-8. Maintaining the Product Table by Commands with Parameters. Create an Inventory database with a Product table that has the following fields:

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>ProductID</td>
<td>Product ID</td>
<td>Has a pattern aaa-nn; where a = letter; n = number. Make this field the primary key</td>
</tr>
<tr>
<td>Name</td>
<td>Product Name</td>
<td>Text string</td>
</tr>
<tr>
<td>Description</td>
<td>Description</td>
<td>Text string</td>
</tr>
<tr>
<td>Quantity</td>
<td>Quantity on hand</td>
<td>Maximum is 50,000; no fractional unit</td>
</tr>
<tr>
<td>Price</td>
<td>Unit cost</td>
<td>Values between (inclusive) 20 and 100 with decimal point</td>
</tr>
</tbody>
</table>

Develop a project to maintain this table with Save and Delete buttons. Use both data commands exclusively to perform the necessary operations. The necessary SQL statements should be constructed only once. Actual data should be passed to the data commands as parameters.
9-9. **Maintaining the Product Table by the Data Adapter Only.** Modify Exercise 9-8 so that all maintenance operations are performed through a table adapter. Use the data designer to visually create the dataset and table adapters. There should be a separate Update DB button; and the database is updated only when this button is clicked.

9-10. **Viewing a Selected Product Record: Use of Data Reader.** Develop a project that allows the user to view a selected product record in the Product table (as specified in 9-9). The project should behave as follows:

- As soon as the project starts, a combo box will be populated with the product ID. Use the data reader (but without the data table) to obtain the data from the product table.
- When the user selects a product ID from the combo box, all the product fields for that product are shown in the textboxes on the form. This record should be obtained through a data reader.

For efficiency, the product ID should be passed as a parameter to the data command that executes the ExecuteReader method.

9-11. **Viewing a Selected Product Record: Use of Data Reader and Data Table.** Modify Exercise 9-10 so that the combo box is bound to the data table which loads the data reader’s contents after obtaining the product ID and Name from the Product table (*Hint:* Set the combo box’s data source, display member, and value member properties).

9-12. **Maintaining the Customer Table by Data Commands with Parameters.** Create an Accounts Receivable database for a credit card company with a Customer table that has the following fields:

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>CustomerNo</td>
<td>Customer Number</td>
<td>Primary key; a four digit number ranging from 2001 to 5000; long integer</td>
</tr>
<tr>
<td>Name</td>
<td>Customer Name</td>
<td>Company or individual name; Text string; maximum 60 characters</td>
</tr>
<tr>
<td>Address</td>
<td>Street address</td>
<td>Text string; maximum 255 characters</td>
</tr>
<tr>
<td>City</td>
<td>City</td>
<td>Text string; maximum 60 characters</td>
</tr>
<tr>
<td>State</td>
<td>State code</td>
<td>Text string; 2 characters</td>
</tr>
<tr>
<td>Zip code</td>
<td>Zip code</td>
<td>5 digit zip code; long integer</td>
</tr>
<tr>
<td>BegBal</td>
<td>Beginning balance</td>
<td>Account balance at the beginning of the period; 0 when account is created</td>
</tr>
<tr>
<td>BalDate</td>
<td>Date balance computed/updated</td>
<td>Date/time; when a new customer balance is compute, the latest date of transaction is saved here</td>
</tr>
</tbody>
</table>

Develop a project to maintain this table with Save and Delete buttons. Use data commands exclusively to perform the necessary operations. The necessary SQL statements should be constructed only once. Actual data should be passed to the data commands as parameters.

9-13. **Maintaining the Customer Table by the Table Adapter.** Modify Exercise 9-12 so that all maintenance operations are performed through the dataset and table adapter created visually with the dataset designer. There should be a separate Update DB button to handle all changes done to the data table.

9-14. **Viewing a Selected Customer Record: Use of Data Reader.** Develop a project that allows the user to view a selected customer record in the customer table (as specified in 9-12). The project should behave as follows:
- As soon as the project starts, a combo box will be populated with CustomerNo as value member and customer name as display member. Use the data reader along with the data table to obtain the data from the Customer table.
- When the user selects a customer from the combo box, all the customer fields for that particular customer are shown in the textboxes on the form. This record should be obtained through a data reader. For efficiency, the customer number should be passed as a parameter to the data command that executes the ExecuteReader method.

**9-15. Viewing a Selected Customer Record: Use of Data Reader and Data Table.** Modify Exercise 9-14 so that the combo box is bound to the data table which loads the data reader’s contents after obtaining the customer number and name from the customer table. Take advantage of the combo box’s display member and value member properties.

**9-16. Maintaining the Customer Transaction Table by Commands with Parameters.** Add a Transaction table to the Accounts Receivable database (as specified in 9-12) with the following fields:

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>TransactionNo</td>
<td>A sequential number</td>
<td>Primary key; auto number</td>
</tr>
<tr>
<td>TransDate</td>
<td>Date of Transaction</td>
<td>Date/Time</td>
</tr>
<tr>
<td>CustomerNo</td>
<td>Customer Number</td>
<td>The customer who made this transaction</td>
</tr>
<tr>
<td>TypeOfTrans</td>
<td>Payment or charge</td>
<td>1 for charge and -1 for payment; Integer</td>
</tr>
<tr>
<td>ReferenceNo</td>
<td>Either a merchant number or</td>
<td>If this transaction is a payment, this number is a remittance number; otherwise it is a merchant number who charged this customer</td>
</tr>
<tr>
<td></td>
<td>a remittance number</td>
<td></td>
</tr>
<tr>
<td>Amount</td>
<td>Amount of Transaction</td>
<td>double</td>
</tr>
<tr>
<td>ProcessedYet?</td>
<td>Has this amount been included in the customer beginning balance?</td>
<td>Yes/No (Boolean); initially No and changed to Yes when a new customer beginning balance is computed and updated</td>
</tr>
</tbody>
</table>

Also, set up the relationship between the two tables so that the customer number in the transaction table is the foreign key to the customer table. (Hint: Refer to 9-2 for instructions on how to create a relationship between tables.)

Develop a project to maintain the transaction table with Save and Delete buttons. Use data commands exclusively to perform the necessary operations. The necessary SQL statements should be constructed only once. Actual data should be passed to the data commands as parameters. Also, because customer numbers to be entered here must exist in the customer table, include a combo box bound to a data table with the customer name column being the display member and the customer number as the value member. (Hint: Transaction tables are typically large and their entire contents should not be kept in memory. Thus, queries to determine whether a transaction exists should be made directly to the DB table. Use a data command and a data reader to handle this task. The transaction number field should be left blank for a new transaction. If a transaction number is entered, it should be used to retrieve the existing transaction from the table and displayed on the entry screen for update or delete.)

**9-17. Maintaining the Customer Transaction Table by the table Adapter.** Modify Exercise 9-16 so that all maintenance operations are performed through the dataset and table adapters created with the dataset designer. (Hint: As explained in 9-16, you should not fill the contents of the entire transaction table into the data table. Instead, it should keep only new transactions and existing transactions to be changed. Create an
additional query in the table adapter with the transaction number as the parameter. Name the query FillByTransNo.)

9-18. **Maintaining the Customer Table by the Data Adapter Only.** Modify Exercise 9-16 so that data in database are maintained by the use of a data adapter’s Update method. All the data adapter’s insert, update, and delete commands should created by code. Add code to automatically save all the changes to the data table when the user clicks the Quit button before clicking the Update DB button.

9-19. **Viewing All Transactions for a Selected Customer: Use of the Data Reader.** Develop a project that allows the user to view all transactions for a selected customer in the accounts receivable database maintained in the preceding exercises. The project should behave as follows:

- As soon as the project starts, a combo box will be populated with Customer name. Use the data reader to obtain customer numbers and names from the Customer table. (Hint: Retrieve data for both the customer number and customer name columns and load them into a data table. Bind this data table to the combo box with the customer name being the display member and customer number being the value member.)
- When the user selects a customer from the combo box, all fields from the customer records will be displayed in a header area consisting of textboxes; and all the transactions that this customer has made will be displayed in data grid view.

Use the data reader to retrieve data from both DB tables. All data command calls should be done with proper parameters.

9-20. **Obtaining Total Transaction Amounts for All Active Accounts Through Stored Procedure: Use of Data Reader and Data Table.** (Note: This project requires the MS Access Query by Example capabilities not discussed in this chapter.) In the preceding Accounts Receivable database, create a query that computes account balance for all active customers (customers that have transactions) with the formula: BegBal + Sum (TypeOfTrans * Amount). Create a query to compute the sum of TypeOfTrans * Amount grouped by CustomerNo. The query should also include the CustomerNo and Name. Name this query TotalTransAmount.

Develop a VB project that will execute the TotalTransAmount query in the database (Hint: This is a stored procedure.) The results should be displayed in a data grid view.

9-21. **Computing Customer Balances.** Refer to 9-20. Add code to the project so that customer number, name, beginning balance, total transaction amount, and ending balance are displayed in a data grid view. Be aware that some customers may not have any transactions in the period.

9-22. **Maintaining the Chart of Accounts Table by Commands with Parameters.** Create a general ledger database for a retailer (or a company of your choice) with an Account table that has the following fields:

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>AccountNo</td>
<td>General ledger account number</td>
<td>Primary key; a four digit number ranging from 1001 to 5000; long integer</td>
</tr>
<tr>
<td>Name</td>
<td>Account Name</td>
<td>Text string; maximum 60 characters</td>
</tr>
<tr>
<td>Class</td>
<td>Account classification</td>
<td>1=asset; 2=liability; 3=owner’s equity; 4=revenue; 5=expense; long integer</td>
</tr>
<tr>
<td>Normal balance</td>
<td>Which side (debit or credit) does this account normally</td>
<td>1=debit; -1=credit; long integer</td>
</tr>
</tbody>
</table>
Develop a project to maintain this table with Save and Delete buttons. Use data commands exclusively to perform the necessary operations. The necessary SQL statements should be constructed only once. Actual data should be passed to the data commands as parameters.

9-23. Maintaining the Account Table by the Data Adapter Only. Modify Exercise 9-22 so that all maintenance operations are performed through the data adapter. There should be a separate Update DB button to handle all changes done to the data table.

9-24. Maintaining the Chart of Accounts Table by the Table Adapter. Modify 9-22 so that the Chart of Accounting table is maintained by a table adapter created visually with the dataset designer. There should be a separate Update DB button to handle all changes done to the data table.

9-25. Viewing a Selected Account Record: Use of Data Reader. Refer to 9-22. Develop a project that allows the user to view a selected account in the account table. The project should behave as follows:

- As soon as the project starts, a combo box will be populated with account name. Use the data reader to obtain both the account number and account name from the database. *(Hint: Set the comb box’s data source, display member, and value member properties properly.)*
- When the user selects an account from the combo box, all the account fields for that particular account are shown in the textboxes on the form. This record should be obtained through a data reader. For efficiency, the account number corresponding to the selected account name should be passed as a parameter to the data command that executes the ExecuteReader method.

9-26. Maintaining the Journal Entry Tables by Commands with Parameters. Refer to Exercise 9-22. Add a Journal table and an EntryLine table to the general ledger database. The journal table should have the following fields:

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>EntryNo</td>
<td>A sequential number representing the journal entry number</td>
<td>Primary key; auto number</td>
</tr>
<tr>
<td>TransDate</td>
<td>Date of Transaction</td>
<td>Date/Time</td>
</tr>
<tr>
<td>Description</td>
<td>Description of the transaction</td>
<td>String, maximum 255 characters</td>
</tr>
</tbody>
</table>

The entry line table should have the following fields:

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
<th>Remark</th>
</tr>
</thead>
<tbody>
<tr>
<td>LineNo</td>
<td>A sequential number representing the line number</td>
<td>Primary key; auto number</td>
</tr>
<tr>
<td>AccountNo</td>
<td>Account number</td>
<td>Long integer</td>
</tr>
<tr>
<td>Amount</td>
<td>Amount of transaction</td>
<td>Double; a positive number is a debit and a negative number is a credit</td>
</tr>
</tbody>
</table>
Develop a project to maintain journal entries in these two tables. The visual interface should look like the following:

<table>
<thead>
<tr>
<th>Entry No.</th>
<th>Date</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Account No.</td>
<td></td>
<td>Debit amount</td>
</tr>
<tr>
<td>Account No.</td>
<td></td>
<td>Credit amount</td>
</tr>
</tbody>
</table>

For simplicity, assume one debit and one credit for each entry. (Note: If you want to relax this assumption, replace the data grid view for the entry line area.) The data in the header area (entry number, date, and description) should go into the journal table; and entry lines, in the line entry table. The necessary SQL statements should be constructed only once; Actual data should be passed to the data commands as parameters. Also, because account numbers to be entered here must exist in the account table, you may want to replace the account number textboxes with two combo boxes bound to the data table with the account name column being the display member and the account number column being the value member. The account number that corresponds to the account name selected by the user is the one to be saved. (Hint: The journal entry tables can be large and their entire contents should not be kept in memory. Thus, queries to determine whether a journal entry exists should be made directly to the DB table. You might want to use the data reader to handle this task. The entry number field should be left blank for a new transaction. If an entry number is entered, it should be used to retrieve the existing transaction from the table and displayed on the entry screen for update or delete.)

**9-27. Maintaining the Journal Entry Tables (both Journal and Entry Line) by Data Adapters Only.** Modify Exercise 9-26 so that all maintenance operations are performed through the data adapter. There should be a separate Update DB button to handle all changes done to the data table. (*Hint*: As explained in 9-16, you should not fill the contents of the two transaction tables into the data table. Instead, the data tables should keep only new transactions and existing transactions to be changed. Use the data adapter’s FillSchema method to set up the data table’s schemata before the data tables are used.)

**9-28. Maintaining the Journal Entry Tables by Table Adapters.** Modify Exercise 9-26 so that all the data operations are performed through table adapters created visually with the dataset designer. Add code to automatically save all the changes to the data tables when the user clicks the Quit button before clicking the Update DB button. (*Hint*: Use this statement to update the data base: `TableAdapterManager.UpdateAll(GeneralLedgerDataSet). When the update involves more than one interrelated tables, a fixed sequence of table adapter calls must be properly made. The table adapter manager handles these calls automatically.)
9-29. Viewing All Transactions for a Selected Account: Use of Data Reader. Develop a project that allows the user to view all transactions/activities for a selected account in the general ledger database maintained in the preceding exercises. The project should behave as follows:

- As soon as the project starts, a combo box will be populated with account name. Use the data reader to obtain the data from the account table. (Hint: Retrieve data for both the account number and name columns and load them into a data table. Then set the combo box’s data source, display member, and value member properties properly.)
- When the user selects an account from the combo box, all fields of the account will be displayed in a header area consisting of textboxes and all the transactions/activities that this account has will be displayed in a data grid view. Use the data reader to retrieve data from both tables. All data command calls should be done with proper parameters.

9-30. Obtaining Account Transaction Totals Through Stored Procedure: Use of Data Reader and Data Table. Construct an account transaction totals query in the general ledger database used in the preceding exercises. The query should take a date as its parameter, compute total transaction amounts for all accounts up to that date (inclusive), and have three fields: account number, account name, and total account transaction amounts (debit balance shown as positive; credit as negative). Save the query as AccountTransactionTotals.

Develop a VB project that will execute the AccountTransactionTotals query in the database. (Hint: This is a stored procedure.) The results should be loaded into a data table and displayed in a data grid view; i.e., a table showing account number, account name, and total transaction amount (separated into debit or credit columns).

9-31. Obtaining Trial Balance Through Stored Procedure: Add code to 9-30 so that the results displayed in the data grid view is the trial balance for the specified date. Keep in mind that not all accounts have transactions in the specified period.

9-32. Exploring DB schemata with dataset and data relation. Refer to the “Browsing Database Table Definitions” example in this chapter. There when the user selects a database table name, the program makes another query into the database for the columns of that table. Modify the project so that data for both the DB tables and the DB columns are kept in two data tables. Create a relation between the two data tables and use the GetChildRows method to obtain the column names for the database table selected by the user.

9-33. Creating DB Tables. Create a blank database and name it PatientAccountManagement for a clinic. Then develop a project to create the following two tables by code:

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Account Number</td>
<td>Numeric; integer; primary key</td>
</tr>
<tr>
<td>Name</td>
<td>Account holder name (Last First, Init); text</td>
</tr>
<tr>
<td>Insurance ID</td>
<td>Text</td>
</tr>
</tbody>
</table>

Patient table

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Account Number</td>
<td>Numeric; integer</td>
</tr>
<tr>
<td>Patient Sequence Number</td>
<td>Numeric; integer 0 for the account holder</td>
</tr>
<tr>
<td>Name</td>
<td>Patient name</td>
</tr>
<tr>
<td>Sex</td>
<td>M or F</td>
</tr>
</tbody>
</table>
**9-34. Constructing Emails with Varying Greetings and Salutations.** Create a Contact database with an AddressBook table that has the following fields:

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description/remark</th>
</tr>
</thead>
<tbody>
<tr>
<td>RecordNo</td>
<td>Auto number</td>
</tr>
<tr>
<td>Salutation</td>
<td>Dr., Mr. or Ms.</td>
</tr>
<tr>
<td>LastName</td>
<td>Max 24 characters</td>
</tr>
<tr>
<td>FirstName</td>
<td>Max 24 characters</td>
</tr>
<tr>
<td>GroupID</td>
<td>A, B, C, etc (to identify contact type such as friends, relatives, colleagues, and customers etc)</td>
</tr>
</tbody>
</table>

Then develop a project that will produce letters to recipients selected from the address book as follows:

- The body of the email will come from a textbox, txtLetter in which the user enters the text
- Your program will add the first line for greeting, salutation, and name of recipient based on what the user specifies with:
  - A combo box that offers three choices for greeting word, Dear, Hello, or (none)
  - A checkbox to indicate whether to include salutation
  - A group of radio buttons to indicate to use last name only, first name only, or full name
  - A combo box for the user to choose to which group of people to send the letter; your program should retrieve all unique Group IDs from the address book table and populate this control with all these Group IDs.

For example, if the user selects Hello for greeting, unchecks the salutation checkbox, chooses the first name radio button, and selects A in the group ID combo box, a letter will be constructed for each of the recipients in Group A similar to the following:

```
Hello! Jonathan,
(Body of the letter will appear here.)
```

But if the user selects Dear for greeting, checks the salutation checks box, chooses the full name radio button, and selects A in the group ID combo box, the letter will be constructed for the same group similar to the following:

```
Dear Dr. Jonathan Tsay:
(Body of the letter will appear here.)
```

For simplicity, pretend the email has been sent by placing the results in a text file.